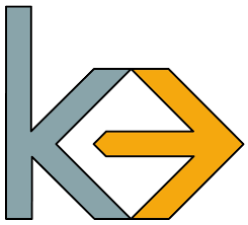
Approximating Policies for Hearthstone via Reinforcement Learning

Master-Thesis von Daniel Haftstein
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Christian Wirth



Approximating Policies for Hearthstone via Reinforcement Learning

Vorgelegte Master-Thesis von Daniel Haftstein

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Christian Wirth

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. März 2017

(D. Haftstein)

Notations

a	Action
s	State
$\delta(s'\|s,a)$	Probability of transition to s' when selecting action a in state s
$\pi(a\|s)$	Probability to select action a in state s when following the stochastic policy π
γ	Discount factor
ϵ	Probability for a random action when following the ϵ-*greedy* policy
$r(s)$	Reward for reaching state s
$\hat{V}^{\pi}(s,\omega)$	Approximate value for state s when following policy π with weight ω
$\hat{Q}^{\pi}(s,a,\omega)$	Approximate value for s and a under policy π with weight ω

Abstract

Markov Decision Processes (MDPs) are a popular model to research decision-making problems in environments where the outcome of an action may be affected by random. The core problem of a MDP is to find a strategy for the decision maker to optimize his reward while interacting with an environment. Common solution approaches to MDPs are provided by *Reinforcement Learning* algorithms which adapt the decision maker's strategy based on exploration of unknown interactions and exploitation of gained knowledge.

Games provide a challenging and popular field of research for such problems because they limit the environment to a model with well defined rules. *Monte Carlo tree search* algorithms are commonly used in games to make decisions based on random samples of future moves. They perform well in situations where random simulations are easy to execute but the calculation of the exact value of a game state is too complex.

In this thesis, we compare the performance of an existing *Monte Carlo tree search* agent on the game *Hearthstone: Heroes of Warcraft* against a new agent which bases on an approximated state value function. We implement a *policy iteration* framework with a value approximation architecture for *Hearthstone* using *reinforcement learning* to receive this function.

We find that an approximating policy is capable to compete with a *Monte Carlo tree search* approach to a certain degree, even without any lookahead or tree search involved. However, the play strength of such a policy depends highly on the distribution of training data and the selected feature set for the approximating architecture.

Contents

1 Introduction

Artificial Intelligence research and games create a mutually beneficial relationship. Games on the one hand, are commonly used for entertainment whereas *Artificial Intelligence* can be applied to multiple domains, not only games. As games are mostly well-defined and small environments, they provide a great area for research of *Artifical Intelligence* because solutions can be be adapted to other domains like stock-market-trading or automated controlling of machines. The general question, either for games or related categories like controlling problems, is which action a decision maker should choose at a given time in order to maximize his performance in the environment.
A common model for this decision making problem are *Markov Decision Processes* (MDPs) which represent the problem as a formal structure of *states*, *actions* and transitions between states, leading to a *reward*. They can be solved or approximated using methods of *Dynamic Programming* (DP) or *reinforcement learning*. Both approaches can be used to find an optimal or nearby optimal solution which allows the decision maker to maximize his reward in the MDP.
Instead of learning from a set of predefined input-output pairs with expected input and output, as used by *supervised learning*, *reinforcement learning* uses the agent's interaction with the environment to learn more or less rewarding actions. Exact solutions to a MDP decision problem are not always possible, especially for environments with a large space of possible states and actions. For this reason approximate methods are used, which reduce the size of the search space while still maintaining a good performance if chosen well.
A game with such a large space and complex state representations is *Hearthstone: Heroes of Warcraft* where the decision problem has been solved quite well using a *Monte Carlo tree search* [1]. One disadvantage of this approach is the inability to re-use observed informations between different matches, therefore the *Monte Carlo* approach is feasible when *evaluating* but unable to *learn* from it's observations.
This work focuses on the elaboration of an approximating *policy* for *Hearthstone*, which will be learned using *reinforcement learning* algorithms. We search for a value function which is capable to estimate the "quality" of a game state. By *exploiting* this value function, the decision making *agent* could select a valuable action by only evaluation the direct next state, instead of using deep rollouts like a *Monte Carlo tree search*.

1.1 Goal

The goal of this thesis is to develop an approximating policy for the game *Hearthstone* via reinforcement learning which can be used by an agent to play the game. This requires an approximate value function to score states of the game as the entire state space of *Hearthstone* is too large for an exact representation. We investigate the upcoming difficulties which emerge when designing an approximate feature set for a complex environment. We evaluate how such an approximation can be optimized and how well it performs when compared against other state-of-the-art solutions.

1.2 Structure

We already introduced the general topic of this thesis in Section 1 and defined our goal in Section 1.1. In this section, we give an overview over the overall structure and a short insight in each of the following seven sections.

After this introduction, we present a deeper introduction to the *reinforcement learning* topic in Section 2. We use the formal model of a *Markov decision process* in Section 2.2 to formalize the general learning problem. Section 2.3 introduces the *policy iteration* process as abstract model to generate a series of improving *policies*, followed by Section 2.4 where we have a deeper look at the *exploration vs exploitation* dilemma.

Algorithms for *reinforcement learning* are introduced in Section 2.5, starting with the class of *TD* algorithms as general approach to *reinforcement learning* problems. A more advanced approach for *policy iteration* is then shown in Section 2.6, where we have a deeper look at the *Least Squares Policy Iteration* algorithm. In the same context, we have a look at approaches to reduce the amount of required training samples, namely *importance sampling* in Section 2.7 as approach to re-use samples from different probability distributions and *gradient based policy updates* in Section 2.8 to modify the update rate of learned weights.

After we summarized the *reinforcement learning* techniques for our further works, we use Section 3 to introduce the game *Hearthstone* as problem domain for our work. We have a deeper look at the game mechanics and define the logic entities and properties in the game's domain.

Our main work starts in Section 4, where we use the definitions and algorithms of Section 2 and the domain knowledge from Section 3 to apply *reinforcement learning* to *Hearthstone*. Therefore, we introduce the action space for a *Hearthstone* agent in Section 4.1 and have a deeper look at the complexity of action sequences. In order to handle the large state space of the game, we develop *features* for *Hearthstone* game properties to receive an *approximating value function* in Section 4.2. We will also discuss further problems like randomness in Section 4.4 or the decision when to end the turn in Section 4.5. Finally describe our learning approach for *Hearthstone* with different setups in Section 4.7.

We evaluate our approach in Section 5 where we test the solutions from the previous section in different experiments. These experiments are based on a *Hearthstone* simulation framework for *Monte Carlo tree search* algorithms, which we describe in Section 5.1. We implement *policies* for different agents in Section 5.3. The impact of different learning parameters on the trained agent's performance is then evaluated in Section 5.4, Section 5.5 and Section 5.6.
In Section 5.7 we test our approximation features for a game state and evaluate the required sample size to train them. After all these experiments, we summarize our results in Section 5.10 where we test the best agents from the previous experiments against each other in a *round robin* tournament.

After our experiments, we compare our work to related projects in Section 6 while focusing on differences and similarities to them.

Last but not least, we conclude our work in Section 7, where we summarize our overall results and the experienced issues and solutions.

Ideas which were out-of-scope of this thesis but valuable for future work are explained in Section 8.

2 Reinforcement Learning

Reinforcement learning covers the task to learn from sequential interaction with an *environment* to achieve a goal. It is a very intuitive way to describe learning because we increase our experience for various tasks in a similar manner. Let's take the task to buy groceries from a store as example. The first time one has to buy an entire shopping list for one week of supplies in an unknown store may take a while. We are unlikely to find all products immediately and will probably miss cheaper or better alternatives while selecting our goods. If we repeat this task over several weeks, our performance will increase. We learn where our most important products are located or what alternatives are available if we need a higher quality or want to save money on a cheaper alternative. Maybe we observe sales promotions and use them to minimize our spent money even more. We could spot times where the store is less populated to decrease our entire shopping time.
There are way more examples, for example to learn how to type faster on a keyboard simply by using it for a long time period. Or driving the same car over several years, we would become better at estimating

how our environment reacts on our actions, allowing us to reach our goal faster and saver.

Using the formal methods of *reinforcement learning*, we can apply this learning approach to different problems as long as they meet different criteria. We will use the rest of this chapter to introduce the common formalizations which are used to define a *reinforcement learning* problem by using definitions and examples from R. Sutton and A. Barto in the book *Reinforcement Learning: An introduction* [2], which summarizes many aspects of *reinforcement learning*.

The entity which learns from the environment and makes the decisions is called *agent*. Environment and agent interact with each other in discrete time-steps and on each time-step, the agent performs an action on the environment which leads to a new state s' and a reward $r(s) \in \mathbb{R}$. The goal of the agent is to maximize his accumulated reward (the return) while interacting with the environment over a sequence of actions. In many cases, this leads to a trade-off between long-term and short-term return.

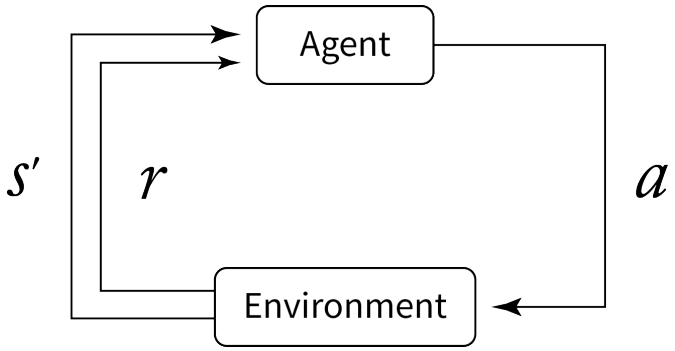


Figure 1: The agent-environment model.

Figure 1 shows the conceptual interaction between the agent and the environment. The agent selects one action a at each time-step for the current state s. The environment updates the state to the new state s' and the agent receives an reward r for the transition to the new state.
This definition is very general and does not make any assumptions about the states, actions or the environment.
One powerful feature of *reinforcement learning* is the ability to learn from the environment without understanding the entire environment or even creating an analytic solution. The typical input for learning tasks are so called *samples* which consist only of *states*, *actions* and *rewards*. A sample must not necessarily be a triple of these values but can also contain more information, like the second-next state and action. Samples may be drawn from the original environment, from a model/simulation of the environment or from another agent than the learning one.

This distinguishes *reinforcement learning* from *supervised learning* where samples are provided by an external supervisor. *Supervised learning* is not always applicable to interaction problems because we have to find samples that are correct and also cover as much of the interaction space as possible. Taking the previously used grocery store example again: We can easily define criteria for optimal shopping, like spending as less money as possible and/or acquiring the entire shopping list in the least possible amount of time. However, how would a supervisor create good samples for this problem without solving the entire problem analytically? If we apply *reinforcement learning* approaches, we could start with any agent, even a random one and learn from our experience while repeating the task multiple times. It is not certain that the agent would always pick the optimal action, it could even select bad actions in many cases. However over a longer series, the agent will start to perform better and better. We can compare this to our own learning experience while driving a car. We will mistakes while learning to drive, maybe even crash the car. However over a longer time we become better and better in driving. Maybe we form

some inefficient habits in this process but our overall performance is very likely to be better than our initial performance when we are confronted with the task for the first time.

2.1 Markov Property

The state signal is the information which the agent receives to make a decision for one state. This must not be limited to typical sensor input but can also be more abstract or derived representations like a velocity property for a moving entity, measured on two different timestamps. A state signal is *Markov* or has the *Markov property* if it delivers all relevant information for the next state transition from s to s'. We will use the definition from [2] again.
For example, the board position of a checkers game with the board positions of all pieces is a state signal with the *Markov property*. It does not reveal the next move of the opponent or the complete history of all move sequences played in the match. However, it contains everything relevant for the future progress of the game.
The transition function $\delta(s'|a,s)$ denotes the probability to reach state s' when performing action a in state s. Given a series of states and actions $s_1, a_1, s_2, a_2, \ldots, s, a$, we define a state signal as *Markov* if the transition probability depends only on the current state s and the action a.

$$\delta\left(s'|a,s\right) = \delta\left(s'|s_1,a_1,s_2,a_2,\ldots,s,a\right) \tag{1}$$

Therefore, a state signal is a *Markov* state and has the *Markov property* if Equation (1) is true.
The environment dynamics are defined by the transition probabilities $\delta(s'|a,s)$ and the reward function $r(s)$. These *one-step* dynamics allow to predict the next state and reward only from the current state. Therefore an agent that uses a *Markov* state to select the next action can perform as good as an agent that keeps track of the entire state and action history.

2.2 Markov Decision Process

A *Markov Decision Process* (MDP) is a reinforcement learning task, that satisfies the *Markov Property* and is defined by the space of possible states and actions combined with the one-step dynamics of the environment. Similar to the previous sections, we stick to the definitions used in [2]. In our further work, we assume that the MDP has a *finite* amount of time steps / a *finite* horizon and a *discrete* space of states and actions.
We already introduced the transition probability for one time step in Equation (1). Actions on each state are selected by an agent using a *policy* π. We use the definition $\pi(a|s)$ for a stochastic policy. $\pi(a|s)$ is the probability for the agent to select action a in state s under policy π.
The *policy* describes the action selection of an *agent*, for example the choice to perform a bet in a given state of poker. A stochastic policy expresses a randomness in these decisions, therefore the agent may select a different action in the same state. On the other hand, the *transition probability* defines the impact of the *environment* on the state change. Depending on $\delta(s'|a,s)$, the agent is not guaranteed to end in the same state s' when performing the same action in two similar states.
The goal of the reinforcement learning task for a finite MDP is to find a policy $\pi(a|s)$ which maximizes the total reward while the agent advances through the state transitions. Given a series of reached states, $s_1, \ldots, s_n$ with s_n being a *terminal* state, the total reward or *return* is denoted with R.

$$R = \sum_{t=1}^{n} \gamma^t r\left(s_i\right) \tag{2}$$

The *discount factor* $\gamma \in [0,1)$ in Equation (2) models the importance difference between long-term and short-term return of the agent and is part of the learning problem. A discount factor close to zero describes a lower importance of future rewards as Equation (2) converges faster to 0 for larger sequences.

On the other hand, a discount factor close to one favors future rewards. Discounting is very important for learning problems with an infinite horizon. If the reward is not discounted and the action sequence is infinite then the return of all policies with an average reward greater than zero would sum up to infinite. If we discount the problem with $\gamma < 1$, then the sum from Equation (2) will not converge to infinite and the return can be used as optimization criteria for different policies.

We can express the *value* of a state with the expected return, when the agent is in state s and follows policy π on all further transitions.

$$V^{\pi}(s) = r(s) + \sum_{a \in A} \pi(a|s) \sum_{s'} \delta\left(s'|a,s\right) \gamma V^{\pi}\left(s'\right) \tag{3}$$

The recursive definition of the value function from Equation (3) is a functional equation, also known as *Bellman equation* [3]. Solving this function would return the best possible value for the decision problem. If solved, we can also use the solution to select the best action a for a state, therefore the solution would return a policy π. However, an exact solution requires to solve the problem for all possible states which is problematic or simply not possible for larger MDPs or continuous state spaces. Without an exact solution, Equation (3) has an error between the left and right side, the *Bellman error*.

Instead of calculating the exact value function $V^{\pi}(s)$, a linear approximation for the value function can be used to define an *approximated value function* $\hat{V}^{\pi}(s,\omega)$. It is represented by k basis functions/features for a state $\phi(s)$ and a weight vector ω.

$$\hat{V}^{\pi}(s,\omega) = \sum_{j=1}^{k} \phi_j(s)\omega_j \tag{4}$$

Using the value function as a measurement, we can compare policies by their performance. The best or optimal policy is denoted by π^*. The policy π^* is the optimal policy, if $V^{\pi^*}(s) \geq V^{\pi}(s)$ for any other policy π.

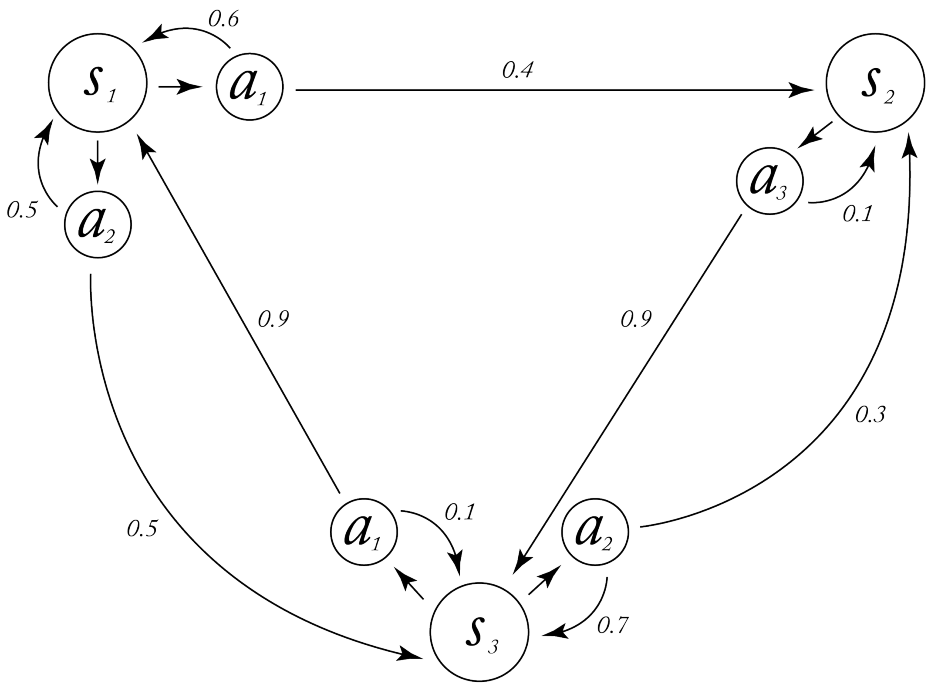


Figure 2: Example for a MDP with three states and three actions.

Figure 2 shows an example for a MDP with three states, s_1, s_2, s_3 and three actions a_1, a_2, a_3. Weights on the edges denote the transition probabilities. For example taking action a_1 in state s_1 has a probability of 0.4 to end up in s_2 and 0.6 to end up in S_1. Therefore $\delta(S_2|a_1, S_1) = 0.4$ and $\delta(s_1|a_1, s_1) = 0.6$. A simple reward function could be $r(s_n) = n$.
A simple policy for this MDP is given in the following equation.

$$\pi_{mdp}(a|s) = \begin{cases} a = a_1 & 0.9 \\ a = a_2 & 0.1 \\ a = a_3 & 1.0 \end{cases} \tag{5}$$

The policy π_{mdp} from Equation (5) will therefore select action a_1 if available or a_3 in s_2 with a chance of 0.9 or the other action, if any.

2.3 Policy Iteration

Policy iteration [4] is an iterative process which generates a sequence of improving policies and value functions. Each iteration consists of two steps: *policy evaluation* and *policy improvement*.

In the *Policy evaluation* step the value function $V^{\pi_j}(s)$ for the current policy π_j is updated by solving or approximating the linear system from Equation (3). The goal of this step is to estimate the value of each state under the current policy. Commonly, the value function is not updated for all possible states but from a number of observed interactions from the policy and the environment. The exact process depends on the selected algorithm. A *Monte Carlo* approach would evaluate the policy until it reaches a final reward, and then update the estimate of the initial state. On the other hand, a *temporal difference* (TD) approach would update the estimate after each step, based on *samples*.
After this value is known, we can decide for each state if the return can be improved by selecting a different action, therefore altering the policy.
This happens in the *policy improvement* step where the value function is used to update the policy to the improved policy π_{j+1}.

$$\pi_{j+1}(a|s) = \begin{cases} \textbf{if} \quad \arg\max_{a \in A(s)} \sum_{s'} \delta(s'|s,a) V^{\pi_j}(s') & 1 \\ \text{else} & 0 \end{cases} \tag{6}$$

The policy π_{j+1} from Equation (6) is at least as good or better as the previous policy π_j. It is also a *greedy* policy because it will always select the action with the highest expected value.
If this process is repeated, the policies converge to or oscillate near the optimal policy π^*.

$$\pi_0 \xrightarrow{e} V^{\pi_0} \xrightarrow{i} \pi_1 \xrightarrow{e} V^{\pi_1} \xrightarrow{i} \dots \pi^*$$

We use the MDP from Figure 2 with the stochastic policy from Equation (5) as example for one step of policy iteration. Given the policy probabilities, we can evaluate the value function $V^{\pi_{mdp}}(s)$. We will use a discount factor of $\gamma = 0.7$.

$$\begin{aligned} V^{\pi_{mdp}}(S_1) =& 1 + 0.9\,(0.6 * 0.7 * V^{\pi_{mdp}}(S_1) + 0.4 * 0.7 * V^{\pi_{mdp}}(S_2)) \\ &+ 0.1\,(0.5 * 0.7 * V^{\pi_{mdp}}(S_1) + 0.5 * 0.7 * V^{\pi_{mdp}}(S_3)) \\ V^{\pi_{mdp}}(S_2) =& 2 + 0.9\,(0.9 * 0.7 * V^{\pi_{mdp}}(S_1) + 0.1 * 0.7 * V^{\pi_{mdp}}(S_3)) \\ &+ 0.1\,(0.3 * 0.7 * V^{\pi_{mdp}}(S_2) + 0.7 * 0.7 * V^{\pi_{mdp}}(S_3)) \\ V^{\pi_{mdp}}(S_3) =& 3 + 1.0\,(0.9 * 0.7 * V^{\pi_{mdp}}(S_3) + 0.1 * 0.7 * V^{\pi_{mdp}}(S_2)) \end{aligned} \tag{7}$$

Solving the linear system from Equation (7) gives the solutions $V^{\pi_{mdp}}(S_1) = \frac{7332370}{1537707} \approx 4.768$, $V^{\pi_{mdp}}(S_2) = \frac{9009370}{1537707} \approx 5.859$ and $V^{\pi_{mdp}}(S_3) = \frac{14172370}{1537707} \approx 9.217$. Using these updated values, we can improve the policy to use the new value function in a *greedy* manner. Taking state S_1 as example, the expected value for a_1 is $0.4 * V^{\pi_{mdp}}(S_2) + 0.6 * V^{\pi_{mdp}}(S_1) \approx 5.2044$ and the expected value for $a_2 \approx 6.9925$. Therefore the improved policy would select a_2 in S_1 with a probability of 1 instead of 0.9.

$$\pi_{mdp+1}(a|s) = \begin{cases} s = S_1 \wedge a = a_2 & 1 \\ s = S_3 \wedge a = a_2 & 1 \\ s = S_3 \wedge a = a_3 & 1 \\ \text{else} & 0 \end{cases} \tag{8}$$

Repeating this process returns the new improved policy $\pi_{mdp+1}(a|s)$. This policy would always select a_2 if possible, which is easy to see on the example because choosing a_2 in S_1 or S_3 has a good probability to end in S_3 with a reward of 3.

Each time the policy is updated and made *greedy* for the current value function, the value function will become incorrect for the new policy because $\pi(a|s)$ changes, therefore altering the value of the sum in Equation (3). On the other hand, each time the value function is updated, the policy is likely to no longer be greedy as other states may lead to a higher value. This is easy to observe on the previous example where $V^{\pi_{mdp}}(S_1)$ would not return the same value for all states as $V^{\pi_{mdp+1}}(S_1)$. If a solution is found where the policy and value function do no longer change, this is the theoretical point where the value function and the policy are optimal.

An exact optimal solution is only possible in smaller action spaces, where exact solutions to the Bellman equations can be computed and the value function and policy can be represented in an exact tabular form. However, in many reinforcement learning tasks, not all possible states of the environment are available or observed but samples for single transitions are used instead. The policy will then be evaluated for all observed transitions.

Approximate policy iteration is also an option to generalize over large state/action spaces. In this case, an approximate architecture as the one defined in Equation (4) is used instead of the exact value function $V(s)$. *Approximate policy iteration* is considered as sound algorithm and will also create a series of policies which differ from the optimal policy by an error $||\hat{V}^{\pi} - V^{\pi}||_2$ which is bounded by the approximation error [5].

We can classify different methods of reinforcement learning by the way how the policy is updated. An *off-policy* algorithm evaluates the policy without following the policy in the evaluation step. For example, the information about the environment can be gathered by random moves or any other policy. In the *policy evaluation* step, the values are treated as if they came from the policy to improved. *Q-learning* [6] is a typical example for an *off-policy* algorithm. On the other hand, an *on-policy* algorithm samples the environment with it's current policy and then improves the policy with these values. SARSA [7] is an example for an *on-policy* learning algorithm.

2.4 Exploitation vs. Exploration

The exploitation vs exploration dilemma is a trade-off between *exploiting* the current knowledge to maximize the own performance in favor of *exploring* and make probably worse decisions but gain new knowledge about the environment. In terms of *policy iteration* we can say, that *exploiting* means to follow the current policy π_j, with the goal to maximize the total reward. This comes with the cost of missing states that may lead to a high reward but are not covered by the current policy. Therefore, the next

evaluated value function $V^{\pi_j}(s)$ is unlikely to differ much from the current value function. On the other hand if we choose to *explore*, we will knowingly select actions which are not *greedy*, thus are likely to end in a state with less reward than other possible states.
This will reduce the return in the actual iteration but improves the value function $V^{\pi_j}(s)$ for the next *policy improvement* step.

The stochastic ϵ-*greedy* policy [2] as defined in Equation (9) is one way to balance exploration and exploitation.

$$\pi_i(a|s) = \begin{cases} \textbf{if} \quad \arg\max_{a \in A(s)} \sum_{s'} \delta(s'|s,a) V(s') & 1 - \epsilon_i + \frac{\epsilon_i}{|A|} \\ \text{else} & \frac{\epsilon_i}{|A|} \end{cases} \tag{9}$$

A *greedy* policy will always select the action which has the highest expected return for the following state. The ϵ-*greedy* policy does the same but keeps a chance to pick a random action from the set of possible actions A. This chance is defined by the parameter $\epsilon \in [0,1]$, which is the amount of randomly selected actions. An agent with $\epsilon = 0$ will always select a greedy action, therefore only *exploit* existing knowledge. On the other hand, an agent with $\epsilon = 1$ will always select a random action and is therefore likely to perform worse but gain more knowledge of the action values, therefore it *explores* the environments characteristics. ϵ is not required to be constant over a series of improved policies but may be adapted to maximize the total reward and therefore speed up the improvement process. A very simple approach is to use a linear interpolation as defined in Equation (10) for k iterations. It starts with a high ϵ^a in the first iterations and ends a low ϵ^b. This increases the exploration in the first iterations, where not much information about the environment is available and moves on to a higher exploitation when the precision of the value function raises in the later iterations.

$$\epsilon_j = (k-j)\epsilon^a + (1-k-j)\epsilon^b \tag{10}$$

More elaborated algorithms to select ϵ like ϵ_n-*greedy* [8] are a broad field of research. The most common problem used for researching the exploitation vs exploration dilemma is the *Multiarmed Bandit Problem* [9] where an agent has to decide which slot-machines (*one-armed bandits*) he wants to play in which order and how many times he wants to play each machine. The agent has no initial knowledge about the reward function of the machines and some may offer a higher reward than others. The agent has to balance between exploring for machines with a higher reward and exploiting machines with a known high reward. As we show later in our experiments at Section 5.5, the change of ϵ had little impact on our results, therefore we did not investigate further approaches in our work.

2.5 Temporal Difference Learning

Temporal difference (TD) learning is a "combination of Monte Carlo ideas and dynamic programming (DP) ideas" [2]. While following a policy π, TD updates the estimated value $V^{\pi}(s)$. Learning samples are drawn from the environment without the need for a model of the environment, similar to a Monte Carlo approach. The estimated values for each state are updated based on previous estimations, similar to DP.

Dynamic programming bases on the Bellman Equation and requires an exact model of the environment. The problem is divided in smaller subproblems and each of them is solved on it's own. When a new estimation is required, the known solutions to the subproblems are combined to obtain the best solution for the global problem.
On the other hand, a *Monte Carlo* approach does not require a model of the environment but is only able to update after an entire sequence passed and the final reward is reached.

TD methods are able to update the estimates after each time step like *dynamic programming* approaches but work on *sampled* observations of the environment like a *Monte Carlo* approach. Therefore a TD method updates the value function based on an estimate after each state-action value.
Goal of this update is to obtain a value function $\hat{V}^{\pi}(s,\omega)$ which is as close as possible to the value function $V^{\pi}(s)$.

$$\mathrm{MSE}(\omega) = \frac{1}{n}\sum_{j=1}^{n}\left(\hat{V}^{\pi}(s,\omega) - V^{\pi}(s)\right)^2 \tag{11}$$

For environments with large or continuous state spaces, an exact solution for the minimum of Equation (11) is not possible. TD uses a numerical approach via *stochastic gradient descent*. This approach requires a parameter α to adjust the step size, otherwise the stochastic gradient descent could overvalue the new local minimum for the next weight ω_{j+1}.
The simplest TD method, *TD(0)* from the family of *TD(λ)* algorithms, introduced by R. S. Sutton [10] uses the update rule from Equation (12) with α being the step size parameter γ being the discount factor. With $\lambda = 0$, only the most recent reward is taken into account.

$$V(s) \leftarrow V(s) + \alpha\left[r\left(s'\right) + \gamma V\left(s'\right) - V(s)\right] \tag{12}$$

We can say that this method is a *bootstrapping* approach because the value function is updated partially based on an estimate and not based on the final reward. This reduces the variance of the value function on each update. The parameter α is a step size factor which controls the update rate.
However, as the update rule will only propagate rewards through one state at a time, learning with this method can take long.
A modification of this approach is the *TD(λ)* algorithm which uses an additional vector to backpropagate rewards over the sampled gradient trajectory. The ratio of this backpropagation is defined with the parameter $\lambda \in [0,1]$. A value of $\lambda = 1$ will use the entire sequence of transitions, similar to a *Monte Carlo* approach but with the drawback of a higher statistical variance on the updated value function. On the other hand, a value of $\lambda = 0$ works like the one-step update from Equation (12).
We did not use the TD algorithm itself, as the parameters λ and α have to be adapted for each training task. Also there are other algorithms which obtain a higher learning rate and omit the additional parameter while still using the same approach as TD. One of them is the *least squares policy iteration* algorithm, which we cover in the next section.

2.6 Least Squares Policy Iteration

One mayor enhancement to the family of TD algorithms is the Least-squares temporal-difference learning algorithm (LSTD) introduced by S. Bradtke and A. Barto [11]. LSTD focuses on the calculation of the parameter ω for an approximated state value function, based on a linear architecture as defined Equation (4). LSTD requires no step size factor and reaches a better learning rate with a higher computational cost.
Least-squares policy iteration (LSPI), introduced by M. Lagoudakis and R. Par [12], is a reinforcement learning method which works without a model and off-policy. LSPI is based on the idea of the LSDT algorithm and provides a least squares solution to minimize the L_2-error between the approximated and the acual value function. Instead of using the *state* value function, LSPI learns an approximate *state-action* value function $\hat{Q}^{\pi}(s,a,\omega)$, comparable to Equation (3) and Equation (4). As we do not make use of *state-action* values (see Section 4.2), we focus on the least-squares approach and adapt the algorithms for *state* values. We will also use the algorithm mainly in an *on-policy* manner but implement an approach for sample re-use in Section 2.7 to use it *off-policy*. LSPI introduces the algorithm LSTD*Q* which learns the state-action value function $\hat{Q}^{\pi}(s,a,w)$. We adapted the algorithm for state values but used the

same update rules as the original implementation.

Input : D : Source of samples (s, a, r, s')
ϕ : Basis functions
γ : Discount factor

Output: ω^{π}

$\tilde{\mathbf{A}} \leftarrow \mathbf{0}$
$\tilde{b} \leftarrow \mathbf{0}$

foreach $(s, a, r, s') \in D$ **do**
  $\tilde{\mathbf{A}} \leftarrow \tilde{\mathbf{A}} + \phi(s)(\phi(s) - \gamma\phi(s'))^T$
  $\tilde{b} \leftarrow \tilde{b} + \phi(s)\, r$
end
$\omega^{\pi} \leftarrow \tilde{\mathbf{A}}^{-1}\tilde{b}$
return ω^{π}

Algorithm 1: Modified LSTDQ algorithm, adapted from [12], for *state* values instead of *state-action* values.

Algorithm 1 represents an adapted version which works on *state* values instead. $\tilde{\mathbf{A}}$ is a squared matrix with the same size as the number of basis functions. The source of samples D starts with an initial policy and updates to the next policy after each iteration. This policy is used to sample the environment for the improvement step, making it an *on-policy* approach. The sample distribution of each iteration j will therefore be biased by the probability distribution of $\pi_j(a|s)$.

In terms of *policy iteration*, Algorithm 1 executes one policy improvement step, depending on the sample source D.

The algorithm LSPI performs approximate policy iteration, using the LSTDQ algorithm.

Input : D : Source of samples (s, a, r, s')
ϕ : Basis functions
γ : Discount factor
ϵ_s : Stopping criterion
ω_0 : Parameter for the initial approximation architecture
Output: ω

$\omega' \leftarrow \omega_0$

repeat
  $\omega \leftarrow \omega'$
  $\omega' \leftarrow \mathbf{LSTD}Q_m(D, \phi, \gamma, \omega)$
until $(\|\omega - \omega'\| < \epsilon_s)$

return ω

Algorithm 2: Modified LSPI algorithm, adapted from [12], which uses Algorithm 1 instead of LSTDQ.

The adapted version which uses Algorithm 1 is shown in Algorithm 2. As we see, the parameter ω is passed as parameter to Algorithm 1 and updated on each iteration. This refers to the update of the sample source D with the current approximate policy, represented by ω. The stopping parameter ϵ_s defines the minimum distance between two updates. If the change at the weight parameter in one iteration is less than a given value, the algorithm will stop. An alternative to this parameter is a fixed amount of iterations, which guarantees the algorithm to terminate. If the amount of samples is too low so that the algorithm learns an entirely different policy in each iteration or the policy oscillates between two

close-to-optimal policies, the algorithm would never terminate.

As the sample source depends on the current policy $\pi_j(a|s)$, the sample distribution in each training iteration is biased by it's probability distribution. If the state space is large and the policy does little to no exploration, the training is likely to suffer from overfitting. A common way to handle this kind of issue is to either *explicit* re-use samples from previous iterations which we describe in Section 2.7, or to re-use the results *implicitly* by updating the previous weights only by an amount, instead of overriding them entirely, which we cover in Section 2.8.

2.7 Importance Sampling

Importance sampling is a variance reduction method that allows to use training samples which are drawn from a different probability distribution [13]. We make use of importance sampling to re-use samples from previous iterations, while performing policy iteration as defined in Algorithm 2. In this case the algorithm would become and *off-policy* approach because not all samples are drawn from the improved policy.
Let D be a source of samples which draws samples from the environment while following the current stochastic policy $\pi_i(a|s)$. If we would use all samples from a previous iteration, the distribution of samples would differ because $\pi_i(a|s) \neq \pi_j(a|s)$ for two iterations i and j. We assign weights to samples from different iterations in order to compensate for the different probability distribution.
$w_{k,n}$ is the probability of the n-th sample in iteration k, relative to all iterations. $w_{0,n} = 1$ because iteration 0 has only the probability distribution from $\pi_0(a|s)$.

$$w_k = \frac{\pi_k(a|s)}{\sum_{i=1}^{k} \pi_k(a|s)} \tag{13}$$

In order to obtain a distribution which sums up to one, the weights are normalized over all samples.

$$w = \frac{w_{k,n}}{\sum_{i=1}^{j} w_{k,i}} \tag{14}$$

Using these weights, we introduce a modified version of Algorithm 1. The source of samples D is expected to draw new samples from the current policy π_j and also return all samples from the previous iterations. We assume the gain of new samples being constant on each iteration. For simplicity, we use the same algorithm with only one loop. Nevertheless, it should be noted that the calculation of w requires an additional iteration over all samples.

Input : D : Source of samples (s, a, r, s')
ϕ : Basis functions
γ : Discount factor

Output: ω^π

$\tilde{\mathbf{A}} \leftarrow \mathbf{0}$
$\tilde{b} \leftarrow \mathbf{0}$

foreach $(s, a, r, s') \in D$ **do**
  $w \leftarrow$ as defined in Equation (14)
  $\tilde{\mathbf{A}} \leftarrow \tilde{\mathbf{A}} + w\phi(s)(\phi(s) - \gamma\phi(s'))^T$
  $\tilde{b} \leftarrow \tilde{b} + w\phi(s)r$
end
$\omega^\pi \leftarrow \tilde{\mathbf{A}}^{-1}\tilde{b}$
return ω^π

Algorithm 3: Modified LSTDQ algorithm from [12] for *state* values instead of *state-action* values.

2.8 Gradient Based Policy Updates

Another way to keep an amount of the previous learned information is to use a gradient based update rule for the policy/weights.
We use an additional parameter $\alpha \in [0,1]$ to control the update rate for the new weights, similar to the step-size parameter of TD algorithms.

$$\omega' \leftarrow \alpha \mathbf{LSTD}Q_m(D,k,\phi,\gamma,\omega) + (1-\alpha)\omega \tag{15}$$

Equation (15) shows the modified update rule from Algorithm 2. $\alpha = 1.0$ is exactly the behavior of Algorithm 2, whereas $\alpha = 0.0$ will never update the new parameters at all.

3 Hearthstone: Heroes of Warcraft

In this chapter we describe the game *Hearthstone: Heroes of Warcraft* or short *Hearthstone* [14]. It is an online collectible card game, released on March 2014 by *Blizzard Entertainment*. The game is free to play with easy to learn game mechanics and is part of the *Warcraft* universe. At 2016 Blizzard announced that *Hearthstone* reached a playerbase of 50 million registered accounts [1].
Hearthstone is played in two-player matches, where both players play against each other. Each player starts the game with 30 health and the goal of each player is to reduce the health of his opponent to zero or below, causing him to win the game and his opponent to lose the game. If both players fall below zero health points, both will loose the game. This makes the game a zero-sum game.
One match is played in rounds, where both players act alternatingly. Therefore, only one player is able to perform actions at a time whereas the other player can not interact with the game. Each turn lasts for a maximum of 75 seconds, after this time the turn will end automatically. A player has always the option to end his turn earlier, which is the case when there are no other options left to play or the player decides not to play any more actions. After one turn ends, the passive player becomes the active player and the active player becomes the observing player.

Before a match starts, both players have to select one deck of 30 cards. Each deck is coupled to one type of hero which represents the player as a character. Hearthstone offers nine heroes from the *Warcraft* universe as choice: Druid, Hunter, Mage, Paladin, Priest, Rogue, Shaman, Warlock and Warrior. Each hero type introduces a set of cards which are unique for the hero. Typically these class-cards reflect the nature of the hero type, like spell cards for the mage or beast cards for the hunter. In addition to these class-cards, *Hearthstone* offers so-called neutral cards, which can be selected by each hero for his deck. Each hero comes with a unique ability which can be activated once per turn and which costs two mana by default.

Mana crystals are the currency which is used to pay cards to play. Each player starts with one mana crystal and gains an additional one on each turn. Spent mana crystals do not disappear but are refreshed on the next turn. Therefore the available amount of mana increases by one on each turn for each player with a maximum of 10 mana crystals. The mana cost of each card is a rough indicator for the card's strength or value. A cheap card can be played earlier in the game but will have less effect than a card with higher cost.

After both players selected their decks but before the first round of a match starts, the game passes through a mulligan phase. The first player is offered three cards and the second player four cards, all randomly selected from their decks. Each player can then choose which cards of them he wants to keep for the game. Cards the players do not keep are shuffled back in their decks and replaced with other randomly selected cards. The second player will also receive a special card, named *The Coin*, in addition

[1] `http://investor.activision.com/releasedetail.cfm?ReleaseID=966955` (Accessed Jan 31. 2017)

to his four starting cards. This card is used to compensate for the disadvantage to go second and allows the player to gain an extra mana crystal in one turn on his choice. Card Order of both decks and the opponent's hand are hidden to each player.

Figure 3: A typical game situation in *Hearthstone*.

Figure 3 shows a game situation in *Hearthstone* where the player plays as *Warrior* (the bottom player) and the opponent as *Mage* (the top player). The player's hero has a weapon equipped which allows him to attack with his hero and dealing 3 damage. Both players have a minion on the board. The player's minion has 1 attack, 4 health and a special ability which is indicated by the small lighting at the bottom of the minion. The opponent's minion has a 3 attack, 2 health and a *deathrattle* effect, which is indicated by the skull icon. The player is on turn and has 3 total mana crystals which he used entirely. Therefore he can't play any more cards as the remaining hand cards have a cost of 4, 3, 7 and 2. The game indicates with a green border around the player's hero, that the hero is still able to perform his attack action. After this action, the player won't be able to perform further actions, except ending his ply.

3.1 Card types

Hearthstone has four main categories of cards with their own mechanics: Minions, Spells, Weapons and Secrets.
Minions are the most important type of cards in *Hearthstone*. Once a minion is played, it will stay on the player's board and allows the player to perform one attack action per turn with this minion. A minion can't act in the turn where it is played. Similar to the player, a minion has a health value but also an attack value. When attacking, the player can attack the opponent's hero or his minions. When one

minion attacks another minion, both loose health by the amount of the other minions attack value. If a minion has zero or less health, it dies and is removed from the game. A player can not have more than 7 minions on his board at the same time. Minions may belong to a subtype, like Beast, Dragon or others. Some cards allow special interactions between these types, for example a minion may increase the attack of other friendly minions with the same type.

Figure 4: Minion example: The *Stonetusk Boar*[2].

Deciding which target to attack is one core element in *Hearthstone*. The player has to make a trade-off between killing the opponents minions and attacking the opponent hero. Removing enemy minions protects the hero from incoming damage in the next turn and may also protect the own minions if they survive to the next turn. Attacking the opponents hero tries to win the game and forces the opponent to remove own minions or play even more aggressive than one self. The drawback of leaving the opponents minions on the board is, that the opponent may find a better way to deal with the board and may create a situation which is more favorable for himself on the next turn. Managing these trades and risks is one ability that is considered hard to master and requires a lot of experience.

Spells on the other hand have unique effects like dealing an amount of instant damage to a target, healing a minion or hero, buffing a minion or drawing cards.

Figure 5: Spell example: The *Frostbolt*[3].

[2] Image source: `http://hearthstone.gamepedia.com/Stonetusk_Boar`, licensed under CC BY-NC-SA 3.0 by Curse, Inc.

Weapons give the player the opportunity to attack with his hero but with the drawback that his hero looses life if he attacks enemy minions. Weapons have a limited amount of uses before they break and are removed from the game. They can be used in the same turn they are played. A player can not equip more than one weapon at the same time. Playing a second weapon destroys the first one.

Figure 6: Weapon example: The *Truesilver Champion*[4].

3.2 Abilities

Cards can have additional abilities, most of them with unique effects depending on the card. We used these abilities as additional features for our value functions (see Table 2).

A card with *battlecry* will trigger an additional effect when the card comes into play. Typical examples are the drawing of an extra card or to deal instant damage to a target of the player's choice.

On the other hand, cards with *deathrattle* have additional effects when the card leaves the game, e.g. when a minion dies or a weapon is destroyed. The effects may be similar to the battlecry effect but won't allow the selection of an extra target. Examples are to draw an extra card, to summon a replacement minion or to deal damage to all enemy targets.

Combo cards trigger an additional effect when they are played after any other card in the same turn. The effects are similar to battlecry in most of the time but can also be different, like spawning a minion with extra attack damage.

Minions with the *charge* ability can attack in the turn they are played instead of the need to wait one turn.

Taunt minions restrict the enemy when selecting his attack targets. It is not possible to target any entity without taunt if the opponent has a taunt minion on the board. This mechanic does not apply to spells or direct effects like battlecries or combos.

Divine Shield is a one-time effect which absorbs one hit with any amount of damage. After one hit, the shield is removed.

[3] Image source: `http://hearthstone.gamepedia.com/Frostbolt`, licensed under CC BY-NC-SA 3.0 by Curse, Inc.
[4] Image source: `http://hearthstone.gamepedia.com/Truesilver_Champion`, licensed under CC BY-NC-SA 3.0 by Curse, Inc.

Windfury allows a minion or weapon to attack two times in one turn. The targets can be chosen different and each attack will be handled as unique action, dealing the full damage but also taking the full damage back if the target is a minion with attack value itself.

Minions with *stealth* can not be targeted. The can still be the target of indirect effects, e.g. targeting all enemy minions. The effect is removed after the minion attacked for the first time. If a minion has *taunt* and *stealth* for the same time, the *taunt* effect does not apply.

Immune protects an entity from any kind of damage but does not protect against negative spells like reducing the attack damage.

Spell Immune minions can not be targeted with effects or spells. They can still take damage from attacks or indirect damage like from spells which target every minion from the opponent.

Frozen entities can not perform any actions until the player to whom they belong performed an entire ply. Therefore, this is a negative effect.

Silence is also a negative effect, which removes any ability from a minion.

Overload is a negative effect that hits the player. Some cards cost overload when played which is kind of an additional cost. Each point of overload will lock a mana crystal in the players next turn. For example: If the player has three mana crystals and plays a card with an overload value of two, he would have four mana crystals in the next turn but two of them locked. Therefore he would have an effective mana of two in the next turn.

4 Reinforcement Learning applied to Hearthstone

To obtain an approximate policy for *Hearthstone*, we make use of the models and algorithms we introduced in Section 2. To do so, we handle the game environment as a MDP where the first player is the agent which plays against an opponent and tries to win the game. Possible actions for one agent can be derived from the game mechanics whereas the state value approximation requires a manual selection of reasonable features. This chapter covers our design decisions and the occurred problems we found while developing a learning algorithm for an approximate policy in *Hearthstone*.

Hearthstone can be easily models as a MDP, because it is played in independent matches. We can therefore describe the game as an *episodic* MDP and one episode being one match. The environment state is the actual state of the game, excluding hidden informations for the agent, and actions are the possible game choices available to the agent. If one action, for example playing a card, triggers multiple events in the game, the agent will perceive this as a single state transaction.
The reward function is also simple to define because of the zero-sum nature of the game. As long as both players play against each other, both have (theoretically) the chance to win the game. Goal of the agent should be to win the game, therefore we reward the game state where he won the game with a value of 1. If the agent looses the game, the reward will be -1. A draw is rewarded with 0. All other states of the game are also rewarded with 0.

4.1 Hearthstone Agents

An agent for the game *Hearthstone* has to select one of the available actions the game offers him or has to wait for the end of the opponents turn. The agent is also able to observe the current gamestate s, except the hidden information like the opponent's hand or the order of both decks. Wo omit the actions for deck building in this thesis and focus on the actions which are executed during a match between two players.

In the first stage of the game, the agent has to perform the *mulligan* phase. In this phase, each player is revealed his initial hand cards and can choose which of them he wants to discard. Discarded cards are replaced with randomly selected cards from the player's deck. After this phase the game begins and the active player selects one of his available actions until his turn ends, which is indicated by selecting the *EndPly* action. The *EndPly* action is always available to the agent and we will use it instead of a time-limit to end the turn. Other available actions depend on the current game state.
Some examples for other actions:

- *Play Minion Wild Pyromancer at board position 0*
- *Play Minion Elven Archer at board position 2, targeting the opponent's hero*
- *Use Warlock Hero Power*
- *Attack with Bloodfen Raptor from board position 0, targeting Enemy Minion Wild Boar at board position 3*
- *Cast Spell Fireball, targeting the opponent's hero*
- *Cast Spell Wild Growth*
- *Equip Weapon Fiery War Axe*
- *Attack with Hero, targeting Enemy Minion Argent Protector*
- *EndPly*

A ply in *Hearthstone* consists of single actions which alter the game state. The complexity of *Hearthstone* arises as these actions can be combined in multiple variations. It is also possible for an action to impose further possible actions which were not available in the initial state.
One example is the attack order of minions on the board. The agent can use any minion but the available targets could change, for example if all enemy *taunt* minion die which would reveal all targets without the *taunt* ability.
Several effects like *deathrattle* or spells allow the player to draw extra cards in his ply. In this case, the amount of possible actions could increase, depending on the drawn card. If we model such actions as a game tree, a node with a card draw would have an outdegree up to the 30 possible cards in a player's deck.
It is also not possible to split minion actions and card play actions, as these could be interlaced with each other. If all seven slots on his board are occupied by minions, he would not be able to place another minion on the board. On the other hand, if the agent sacrifices a minion first, he would not be able to use a spell on this minion which could increase his health and/or attack.

4.2 Value Function Features

Hearthstone has a very large space of possible states which makes an exact tabular representation for a policy impractical. We will use an approximate linear architecture as defined in Equation (4) to represent the value of a state.
This representation requires a set of basis functions/features $\phi_j(s)$ which project the game state to numeric values. These functions are fixed and won't change during learning iterations. The parameters ω_j are used as weights for the value functions and will represent the learned policy.
The core question is, how to select the basis functions to get a model which is close to the real gamestate but also cheap enough in computation time.
In this chapter, we will evaluate different sets of value functions against each other with the goal to find a set of functions that can be used for a good state approximation. We focused on state value

functions instead of state-action value functions because it is also hard to find a value function for actions like minions attack without simulating the outcome of the action. A typical example is a minion with deathrattle: *Deal 2 damage to all other minions*. If such a minion dies because he was attacker/defender, the side-effects to the rest of the game state can be huge. The simulation framework makes it very easy and cheap to clone a game state and apply the event on the clone, therefore we used a state function approach.

As the features function as the agent's view on the environment, we implemented only features for non hidden informations. Features are used in a symmetric approach, therefore a feature is evaluated from the perspective of both players. We define this set of feature as $\phi^{p=c \wedge i}$ with $p = c$ being the perspective of the current active player and $p = i$ the perspective of the inactive player. By concatenating both sets of functions, we define $\phi = \left(\phi^{p=c}, \phi^{p=i}\right)$.

For example, if ${\phi_1}^p$ is the only value function, which returns the health of the player's hero, ϕ would be $\left({\phi_1}^{p=c}, {\phi_1}^{p=i}\right)$. In a game, where player one has 14 health, player two has 13 health and player one is on turn, the policy of player one would evaluate to the base values of $(14, 13)$. If player two was on turn instead, the policy would evaluate to the base values of $(13, 14)$. The advantage of such a model is, that it makes the parameters ω of both players policy easy to compare. A reasonable policy for both players is expected to have $\omega_1 \geq 0$ and $\omega_2 \leq 0$ because it is obviously good to minimize the health of the opponent while maximizing the own health.

A different leading sign of ω_j for two policies can be an indicator, that two agents learned a different meaning for this value function. This must not be a mistake in every case but the model makes it more easy to spot and compare such differences.

We will now define the value functions for ϕ^p. The first set of functions is directly bound to values available from the player's hero.

Value Function	**Description**
Hand size	The amount of cards in the player's hand (0-10). The amount of cards will increase the player's available actions if he has enough mana to pay the card. On the other hand, keeping too many cards can also be an indicator that a player does not use his available cards on the full potential or that the player has too many cards he can't play yet because of insufficient mana.
Board size	The amount of minions on the players board (0-7). It is obviously bad to have zero minions. On the other hand, playing all cards to the board makes them vulnerable to board clear effects.
Hero class	This is a set of functions, one for each hero class. If the hero is of the specific class, the function returns 1 or 0 otherwise. As the hero is static in most of the games, this function does not do very much. But if used with squared value functions (see Section 4.3), it allows the agent to use a policy with respect to his and the opponents hero class.
Deck size	The amount of remaining cards in the player's deck. This is related to the hand size because a large deck with no card remaining on the hand may be an indicator for overplay or missing card draw. On the other hand an empty deck is also bad because the player can not draw anymore. Therefore, this value function may be used to balance the worth or cost of a card draw.
Health	The remaining health points of the player's hero. Obviously this is the win condition but must be balanced with the own and the opponent's board strength.

Mana	The remaining available mana in the player's turn (0-10). If a player does not make use of his mana, it is generally considered bad because he left resources unused. There may be cases where a player wants to save a card for later or has no fitting card for the current turn.
Max mana	The maximum number of mana crystals the player has in the current turn (1-10). Some cards may offer additional (empty) mana crystals which can be captured by this value function.
Used mana	The amount of mana crystals, the player used in the current turn.
Total used mana	The total amount of mana crystals, the player spent in the total game.
Overload	The amount of locked mana crystals, caused from *overload* effects
Fatique	The amount of damage, the player took the last time he tried to draw a card. If the deck is empty, the player will receive increasing damage for each missing card he tries to draw. Fatique starts with one damage, therefore this value function has an initial value of zero.
Armor	Armor protects the player and is used in favor of the player's health when he receives damage.
Attack	The current attack value from the hero. This is not the value of the player's weapon but a temporary attack value which may be given by spell effects and adds to the weapon attack.
Max health	The maximum amount of health the hero can have. This is 30 for all common heroes but there are two replacement heroes which can exchange the current hero due to card effects. Therefore this is expected to be a minor value function.
Frozen	A binary function which returns 1 if the hero is frozen and 0 otherwise. A frozen hero can not attack with his weapon or attack value.
Immune	Similar to frozen, this value function returns 1 if the hero is immune and 0 otherwise.
Attacks	The number of remaining attacks of the hero. This is 1 by default and can only be used if the hero has an effective attack value greater than zero.
Weapon attack	The attack value of an equipped weapon, or 0 if no weapon is equipped.
Weapon durability	The durability of the equipped weapon or 0 if no weapon is equipped.
Weapon windfury	This value function returns 1 if the weapon has windfury (two attacks per turn) or 0 otherwise.

Table 1: Value functions for player attributes.

The value functions from Table 1 do not cover minions on the board. A lot of minions have unique effects which makes it very hard to generalize over a set of minions while taking account of their special abilities and effects. We decided not to learn value functions for each type of minion or for each board position because this would have required a much larger amount of training samples and also reduces the policies capability to generalize over different games. For this reason, we used aggregation functions to estimate the total value of the entire board. Special effects from the minions are covered indirectly because they would affect a following state s' and change the result of $\phi(s')$. For example if the player has two minions with attack two and a health of one. One minion has deathrattle: *Draw a card*. The minion value functions would not cover the deathrattle effect but if the agent has to decide which minion he sacrifices on a taunt minion, the action which kills the deathrattle minion is likely to have a higher value because the player would have an additional card in the following state.

Value Function	Description
Mana Cost	The basic cost of the card. The mana cost is an indicator for the strength or value of a minion, independent from it's attack or health as it also includes unique card effects.
Health	The current health of a minion, including changes through spell effects or abilities and taken damage.
Attack	The current attack value of a minion, including changes through spell effects or abilities.
Attacks	The number of remaining attacks for this minion.
Taunt	1 If the minion has taunt, 0 otherwise.
Windfury	1 If the minion has windfury, 0 otherwise.
Divine Shield	1 If the minion has divine shield, 0 otherwise.
Frozen	1 If the minion is frozen, 0 otherwise.
Stealth	1 If the minion has stealth, 0 otherwise.
Immune	1 If the minion is immune to spells and hero powers, 0 otherwise.
Charge	1 If the minion has charge, 0 otherwise.

Table 2: Value functions for minion attributes.

Table 2 lists the value functions for a single minion. The values of these functions are aggregated for each existing minion on the player's board. We used the *min*, *max* and *sum* as aggregation functions. *Sum* is useful to get the total value of the board, whereas *min* and *max* can be used to balance out the value of too weak or strong minions.

Different combinations of these value functions are evaluated in Section 5.7.

4.3 Squared Features

As in Equation (4), we use a linear combination of the features for an approximate value function. However, there may be cases where features depend on each other. For example, it is considered bad to have low health because this means the player is closer to loosing the game. It is also considered good to have a minion with *taunt* in most cases. However, if the player has lots of health, he may choose to play another minion in favor of the *taunt* minion, because the other minion could have more attack for the same mana cost. If the player is close to death, he may favor the *taunt* minion to protect himself. This dependency can not be expressed by adding both value functions with only one tuning parameter for each without lowering the parameter for the minion attack. If $\phi_t(s)$ is the sum of the player's taunt minions, $\phi_a(s)$ is the sum of the minion's attack values and $\phi_h(s)$ is the health of the player's hero, we can introduce $phi_{th}(s) = \phi_t(s)\phi_h(s)$ as a combination of both features. The policy can now use ω_{th}, ω_h, ω_a and ω_t to express a point where playing the taunt minion is favorable to an attack minion when the player's health is low.
We can generalize this approach and define the set of squared value functions ϕ^s.

$$\phi^s = (\phi_1, \ldots, \phi_n, \phi_1\phi_1, \phi_1\phi_2, \phi_2\phi_3 \ldots, \phi_1\phi_n, \ldots, \phi_n\phi_n) \tag{16}$$

Equation (16) shows the feature combination for ϕ^s. As $\phi_1\phi_2 = \phi_2\phi_1$, we only need half of all possible combinations. We evaluated the squared value functions in Section 5.8.

4.4 Randomness

Hearthstone includes random events which may alter the game state. Therefore $\delta(s'|s,a) < 1$ for some states and actions. This is a huge problem when implementing an ϵ-*greedy* agent which uses the stochastic policy as defined in Equation (9). If an action may result in a lot of different states, for example

playing a minion that causes a card draw from the deck (up to 30 possible cards), the estimation of the action's value gets expensive because the agent needs to evaluate all possible following states.

As simplification, we assumed that $\delta(s'|s,a) = 1$, thus that the environment is *deterministic*. An *ϵ-greedy* agent, as the one described in Section 4.6, would not evaluate all possible states but only one s' for each action a.

Nevertheless, a large amount of actions in *Hearthstone* are in fact *deterministic*, for example playing a minion on the board or attacking targets. We discussed further solutions to this problem in Section 8.

4.5 EndPly Value

The value functions we defined in Section 4.2 do not cover the value of the game state after one player ends his turn (the *EndPly* action). One example for such a case is the usage of the approximated value functions in an *ϵ-greedy* approach like the stochastic policy from Equation (9). If the agent evaluates the game always from the perspective of one player, then the state value after the *EndPly* action $\hat{V}^{\pi}(s', \omega)$ will have be similar value as $\hat{V}^{\pi}(s, \omega)$ because none of the features changed (the active player is no feature). On the other hand, if the agent evaluates the game from the active player's perspective, the following game state would have an entirely different value depending on the opponent's status.

One option to handle this issue is to treat the *EndPly* separate and devalue the following state to negative infinity. The agent would then execute as many greedy actions as possible until no more options than *EndPly* are available. This is the easiest approach and we used it in our experiments as described in Section 5.3.

A more complicated strategy would be to evaluate the entire ply of the opponent, comparable to a *MinMax* approach or a *Monte Carlo tree search*. However to do so, the agent would need to know which cards are in the opponents deck and what cards are in his hand. We did not evaluate further strategies to gain good *EndPly* estimations but this is one mayor point for future improvements, as discussed in Section 8.

4.6 *ϵ-greedy* Agent

Finally, we developed an *ϵ-greedy* agent for *Hearthstone*. The agent uses a stochastic policy as defined in Equation (9). In order to select the next action, all possible actions are evaluated. The agent applies each action as event to a copy of the current game state and calculates the approximate state value as described in Section 4.2. Therefore the agent is only capable to do a one-step lookahead and will treat random actions like deterministic transitions (see Section 4.4).

The agent is parameterized with the current weight parameter ω for the value approximation function and with ϵ to control the randomness of the chosen actions. If the value function scores multiple states with an equal highest score, the agent will select randomly between these actions. Therefore an agent with $\omega = \mathbf{0}$ will behave similar to an agent with $\epsilon = 1.0$.

The *EndPly* action is a special case as it will not alter the game state but change the active player. This may falsify the approximated value of the next state because it will either be evaluated from the perspective of the player who ends the turn or from the other players perspective. In the first case, the state would have the same score, because no feature of the value approximation changed. In the second case, the state value would be evaluated from the opponents perspective, which is also a false value for the player which ends his ply. We discussed solutions to this issue in Section 4.5 and used a fixed value of $-\infty$ as score for the following state. The drawback of this approach is that the agent will execute all possible actions before ending his ply, leaving out the option to pass early.

Input : $A = (a_0, a_1, \ldots, a_n)$: List of possible actions
s : the current state of the game
ω : approximate value function weights
ϵ : randomness parameter
Output: The selected action a

$r \leftarrow$ random number $\in [0, 1]$
if $r < \epsilon$ **then**
  $a \leftarrow$ random element from A
  return a
end

$a_{max} \leftarrow EndPly$
$v_{max} \leftarrow -\infty$
foreach $a_j \in A$ **do**
  $s_{copy} \leftarrow$ copy s
  $s'_{copy} \leftarrow$ apply a to s_{copy} /* assumes $\delta\left(s'_{copy}|s_{copy}, a\right) = 1$ */

  $v \leftarrow \hat{V}^{\pi}\left(s'_{copy}, \omega\right)$
  if $a_j = EndPly$ **then**
    $v = -\infty$ /* could also be embedded in $\hat{V}^{\pi}$ */
  end
  if $v = v_{max}$ **then**
    $v_{max} \leftarrow$ randomly select v or v_{max}
  else if $v > v_{max}$ **then**
    $a_{max} \leftarrow a_j$
    $v_{max} \leftarrow v$
end
return a_{max}

Algorithm 4: Policy of the *ϵ-greedy agent*.

To control the parameter ϵ over multiple iterations, we used the linear interpolation as defined in Equation (10). For simplification, we will denote this with $\epsilon^a = x \rightarrow \epsilon^b = y$ to express the linear interpolation from x in the first iteration to the value of y in the last iteration.
When we evaluate our results, we will always set $\epsilon = 0$ to maximize the agent's policy exploitation for ω_j of the current training iteration.

4.7 Coactive Learning

With the agent from Section 4.6, we set up a learning schedule to train a reasonable policy for the *ϵ-greedy* agent. To obtain a better initial learning result, we considered *supervised training* from existing game samples using recored human games as source to learn the first iteration's policy. However, as we have to use a simulation of the game environment which is not guaranteed to behave 100% similar to the game (see Section 5), we would have to adapt these replays to match our environment. Secondly, the official games has no archive of played games and therefore we had no reliable source of game replays. Some third-party projects like *hsreplay*[5] offer the ability to record games and therefore work like

[5] `https://hsreplay.net/` (Accessed Jan 31. 2017)

a crowd-sourcing archive. However, they don't provide an API to access the collected data and we got no response as we asked for a snapshot of the data for our research.
For this reason, we limited our training to *self-play* which has proven itself as a successful learning approach in *TD-Gammon* [15].
We used a pairwise setting, where two ϵ-*greedy* agents play against each other in each iteration. The first agent is used similar to the one we described previously. The second one functions as the opponent but uses the trained parameters from it's previous iteration. After each iteration we switch both agents and use the other one for training. The training agent uses ϵ_j from the current iteration, whereas the opponent will be restricted to the trained policy from the previous iteration by using $\epsilon = 0$.
Therefore, we train two separate policies, each using half of the training iterations. We denote their weights with $\omega_{p,j}$ with $p \in 1, 2$ being the player number/the policy and j being the training iteration. Similar to the previous setup, $\omega_{1,1} = \mathbf{0}$ and $\omega_{2,1} = \mathbf{0}$. The total series of trained weights (the results from each training iteration) is the series $\omega_{1,2}, \omega_{2,3}, \omega_{1,4}, \ldots$. The second agent's weights are not updated after one iteration. Therefore the second agents ω is the training result from the previous iteration.

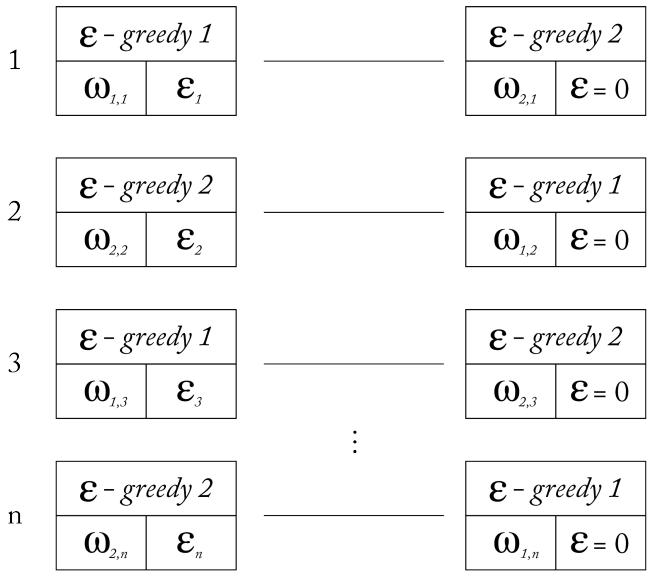


Figure 7: Training schema for two alternating ϵ-*greedy* agents.

The alternating training schema is shown in Figure 7 with two agents, ϵ-*greedy 1* and ϵ-*greedy 2*. As stated above, only one weight is trained per iteration, therefore $\omega_{2,2} = \omega_{2,0}$, $\omega_{1,3} = \omega_{1,2}$ and so on. Which agent is trained last depends on the number of iterations. In this example, we assume n being even, therefore both agents pass through the same amount of training iterations.
One advantage of this learning strategy is the high exploration of both agents in the first iterations. We also receive two instead of one policy in the last two iterations. Similar to the previous approach, one problem could be overfitting, if both policies learn to play well against each other but perform worse against other opponents. To evaluate this possible issue, we test our policies against a third opponent which is independent of the trained policies.

5 Experiments and Evaluation

Blizzard Entertainment keeps *Hearthstone* closed source and does not provide an API to control the game interface. Automated playing the game is even prohibited by the end user license agreement.
Therefore we used the *Hearthstone* simulation library which M. Zopf introduced in his work, where he compared different Monte Carlo approaches on *Hearthstone* [1] for our experiments. The library models

the rules of hearthstone while reverse engineering the original card mechanics. It also offers the ability to play out games between artificial agents and includes Monte Carlo agents for hearthstone which we used as baseline for our evaluations, as described in Section 5.1.

The library is implemented in JAVA and uses an event-driven approach to model the mechanics of *Hearthstone*. Components like the player, minions, cards or the game state itself are entities which can react to events in the system. Hearthstone can be represented very well with an event stack because of the transitive effects that actions may trigger. An example could be an action where the player plays a spell card that kills a minion. That minion has a deathrattle effect which triggers and kills other minions due to an global damage effect. Other minions may react to these death events, e.g. by getting an attack buff if a minion dies.

We extended this simulation library with additional agents as described in Section 4.1 and implemented the coactive learning algorithms from Section 4.7. We used an approximated value function for the agents policy using the features from Section 4.2. The parameter for this function was then trained by an implementation of Algorithm 2 and Algorithm 3.

After defining the features from Section 4.2 for an approximate value function we applied the modified LSPI algorithm as defined in Algorithm 2 to *Hearthstone* in order to train an ϵ-*greedy* agent.

We used a sample source D which streamed new samples from games of two agents on each learning iteration. Samples were represented in the LSPI sample format of (s, a, r, s'). The reward of all samples will be zero, except for the terminal state, where the winner is rewarded with 1 and the loosing agent receives a reward of -1.

Even with the approximated value function in place, the number of states to cover for a generalization is very large. The simplest solution to this problem was to use a large amount of samples on each training iteration. We evaluated the agent's performance with different sample sizes and sets of features for the value function in Section 5.7. Another approach to reduce the required amount of new samples is the gradient based policy update we described in Section 2.8 because it implicitly re-uses the trained information from previous iterations. We evaluated this approach in Section 5.6. Last but not least, we implemented the importance sampling algorithm as described in Section 2.7. The sample source D will then behave as described previously (it creates a fixed amount of new samples using the current policy) but it will also use all samples from previous iterations. The importance sampling results are evaluated in Section 5.9.

5.1 Monte Carlo Tree Search

The simulation framework implements two *Monte Carlo* approaches for *Hearthstone*: UCT [16] and UCB [17].

Both algorithms were implemented with a configurable number of evaluations/simulations instead a time limit for evaluations and also with the tuning parameter $c \in [0, \infty]$. A lower value favors exploitation whereas an algorithm with unlimited c would only explore. One evaluation refers to an entire *Monte Carlo tree search* iteration, consisting of *selection*, *expansion*, *simulation* and *backpropagation*.

M. Zopf already evaluated the impact of the parameter c on both algorithms [1] for the values 0.01, 0.1, 1.1414 and 10.0. His work concludes that a value of 0.01 or 0.1 is to be preferred for UCB with more than 256 simulations the same values worked reasonable with UCT. The difference on UCT was larger for different settings than the impact on UCB.

We wrapped the existing implementation in our agent facade and evaluated the next action using the existing implementation on each step of the game.

We then re-evaluated the parameters with the goal to find a good baseline opponent for our further experiments.

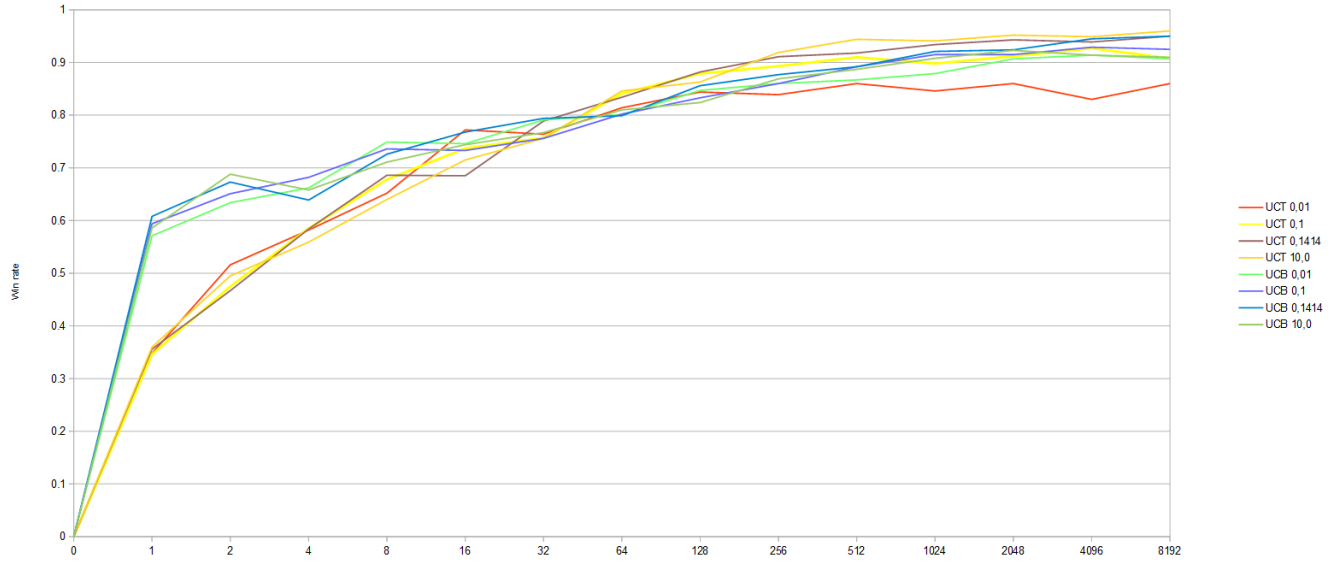


Figure 8: Win rate of the UCT and UCB agent with different amounts of evaluations and $c \in 0.01, 0.1, 0.1414, 10.0$, against a random opponent through 1000 games.

Figure 8 shows the win rate of both algorithms with different settings for c and the number of evaluations against a random agent (see Section 5.3.1). The graphs show the win-rate through 1000 simulated games for each setting.
The UCT algorithm performed better and faster in terms of computation time than the UCB variant as the number of evaluations raised over 32 and for $c = 0.1414$ or $c = 10$. At 256 evaluations, the UCB agents took around 1.6 of the computation time compared to the UCT agents. This number increased with the amount of evaluations. We therefore decided to use only the UCT agent in our further experiments.
We did a second comparison against an UCT opponent with 256 evaluations and $c = 0.1414$ as the results against a random opponent stabilize very fast near a winrate of 0.9. This time, we used a second UCT agent with $c = 0.1414$ and 256 evaluations as opponent, instead of the random agent.

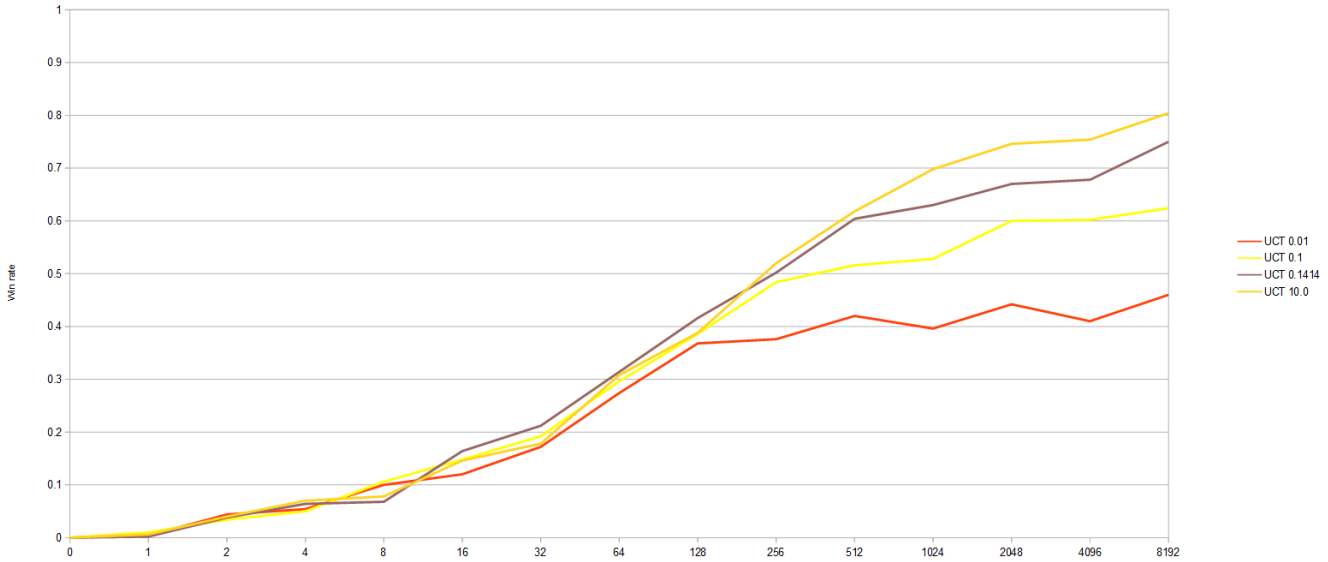


Figure 9: Win rate of the UCT agent with different amounts of evaluations and $c \in 0.01, 0.1, 0.1414, 10.0$, against a second UCT agent with fixed parameters of $c = 0.1414$ and 256 evaluations.

Our re-evaluation of the parameters yield slightly different results than the suggested values. The agent with $c = 10.0$ performed better than the agent with $c = 0.1414$ after 256 iterations. On the other hand, the agent with $c = 0.01$ performed much worse than the other variants. We did not found the exact reason for the derivation in the results. Possible explanations are the different numbers of evaluations and a different parameter c for the opponent. Another explanation could be that the agent evaluates for each action instead once for an entire ply sequence.
We selected the UCT agent with 256 evaluations and $c = 0.1414$ as baseline for our further experiments. At this point the agent achieves a win rate > 0.9 against the random agent. With more iterations, it gets outperformed by the agent with $c = 10.0$ but the higher number of evaluations comes also with a higher computation cost.
As we see in the following experiments, this baseline is feasible to receive a useful amount of information about the game mechanics while learning a good policy. It is easy to switch the training or evaluation to a higher value if more fine tuning to the parameters would be required.

5.2 Training against a static agent

As the simulation framework comes with a *Monte Carlo* implementation and allows a fast simulation of state-action transitions, we can add a second training method in addition to the self-play which we introduced in Section 4.7.
Samples in this variation are drawn from games where an *ϵ-greedy* agent plays against another agent which does not change throughout the iterations. The *ϵ-greedy* agent uses the value parameter ω_n from the actual iteration n and ϵ_n to control his exploration/exploitation rate. ϵ can be fixed throughout all iterations or modified as defined in Equation (10) and below.
The *ϵ-greedy* agent is always initialized with $\omega_1 = \mathbf{0}$, therefore will only select random actions in the first iteration.

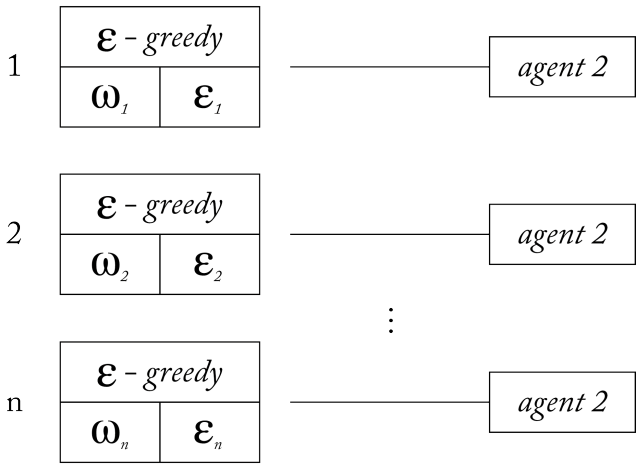


Figure 10: Training schema for the *ϵ-greedy* agent against a fixed opponent.

Figure 10 shows the setup where one *ϵ-greedy* agent plays against an opponent *agent 2* which does not change it's behavior over the iterations. Values of ω and ϵ below the agents are the ones which are *used* by the agent in the current iteration, not the ones which are *trained* in this iteration.

As the opponent stays similar throughout all iterations, we will use this approach mainly to evaluate the learning parameters ϵ and γ and observe the performance of the *ϵ-greedy* agent. The result of each iteration is the parameter γ_j which contains the updated weights for the same improved policy.
One possible disadvantage of this approach is that the trained policy suffers overfitting against this specific opponent and performs worse against other opponents. We use a larger experiment in Section 5.10 with multiple different agents to test if this is the case.

5.3 Agents

We used two additional agents in our experiments. They were implemented as addition to the *Hearthstone* simulation framework and function as described in Section 4.1. The interface for an agent consists of two methods: The first one has to select the cards which should be replaced in the *mulligan* phase, given a list of initial hand cards. The second method has to select one action from a list of possible actions, given the current state of the game and the own player. As mentioned before, we did not implement further card selection logic, therefore all agents discard cards which cost 4 or more mana in the *mulligan* phase.
The action selection logic of each agent is a one to one mapping to a policy $\pi(a|s)$ from our *Hearthstone* model.

5.3.1 Random Agent

The random agent is the simplest agent. It will select a random action until it's ply ends. The list of possible actions will always contain the *EndPly* action. For this reason, the random agent is likely to end his ply earlier than possible. As the game mechanics won't allow some actions like attacking the own hero with the own minions, the random agent is still able to play somewhat reasonable. However, all other actions like selecting targets for spells or choosing the card to play next will be entirely random and may favor the other player directly.

Input : $A = (a_0, a_1, \ldots, a_n)$: List of possible actions
s : the current state of the game
Output: The selected action a

$a \leftarrow$ random element from A
return a

Algorithm 5: Policy of the *random agent*.

5.3.2 UCT/UCB Agent

These agents use the UCT/UCB implementation as described in Section 5.1 to evaluate the next action. On each step, the UCT/UCB algorithm is executed again for the current state of the game and the best action is selected.

Input : $A = (a_0, a_1, \ldots, a_n)$: List of possible actions
s : the current state of the game
Output: The selected action a

$\tilde{A} \leftarrow$ evaluate UCT/UCB action sequence for s `/* Use the existing algorithm as black-box */`

return *first element of* $\tilde{A}$

Algorithm 6: Policy of the *UCT/UCB agent*.

5.4 Discount Factor

The agent is only rewarded at the terminal state of the game, where one player has lost the game (see Section 4). We have to choose a discount factor in a way that the first samples still receive a value for the possible return and are not overrated on the other hand because the state of the game is still uncertain. In our simulations, the games had a sample size around $80-120$. As the discount factor is multiplied on each proceeding state, the value for early samples in a game becomes really small (see Equation (2)).

A very low discount factor close to 0 is unlikely to utilize the value of the early game correctly because the samples would have a very low expected return. On the other hand a discount factor of $\gamma = 1.0$ is probably too high because the early samples can't have an exact return of 1 or -1. This would implicate that the game is already decided on the first actions. Therefore we would learn a false value for the early samples, depending on the sample distribution.

In our first experiment, we tested different settings for the discount factor γ with the *ϵ-greedy* agent against a random opponent. The random opponent plays worse than the UCT agent and most of the *ϵ-greedy* settings, therefore the games are shorter in average. We observed an average sample count per episode in the range of 30 to 40.

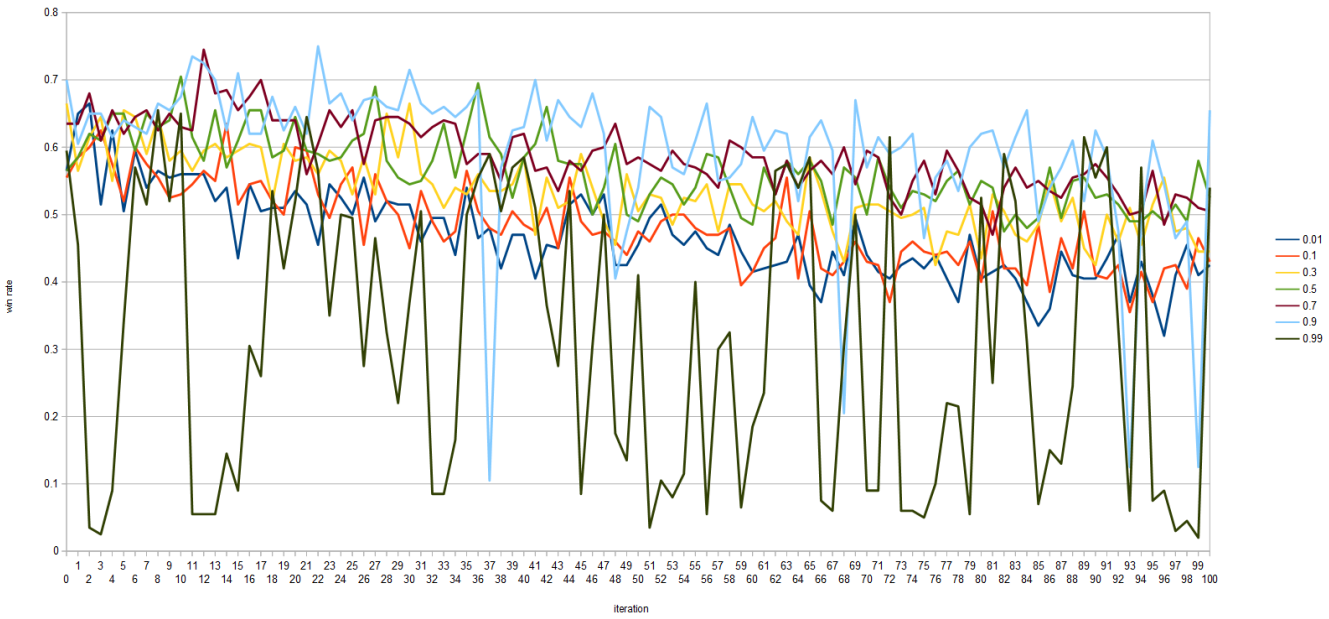


Figure 11: Win rates of an *ϵ-greedy* agents trained with different settings for γ against a random opponent over 200 games per iteration against the default UCT opponent.

Figure 11 shows the results of the first test. We used an update rate of $\alpha = 0.7$ and $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$ over 100 iterations with 100000 samples on each iteration. We evaluated the weights ω_j from each iteration over 200 games against the UCT agent.

The edge cases with $\gamma = 0.01$ and $\gamma = 0.99$ performed as worse as expected. The high discount factor resulted also in a very unstable training result, although as we updated only with $\alpha = 0.7$. We can also observe an overall decrease in the agent's performance over the series of iterations. This can be caused by overfitting of the policy against the random opponent or general overfitting as ϵ decreases and the sample distribution shifts to exploitation of the current policy.

For a better comparison, we re-evaluated the discount factor with the same learning parameters on the pairwise training with two *ϵ-greedy* agents, as described in Section 4.7. Similar to the previous experiment, the resulting policy from each iteration was evaluated against the UCT agent with 256 evaluations and $c = 0.1414$.

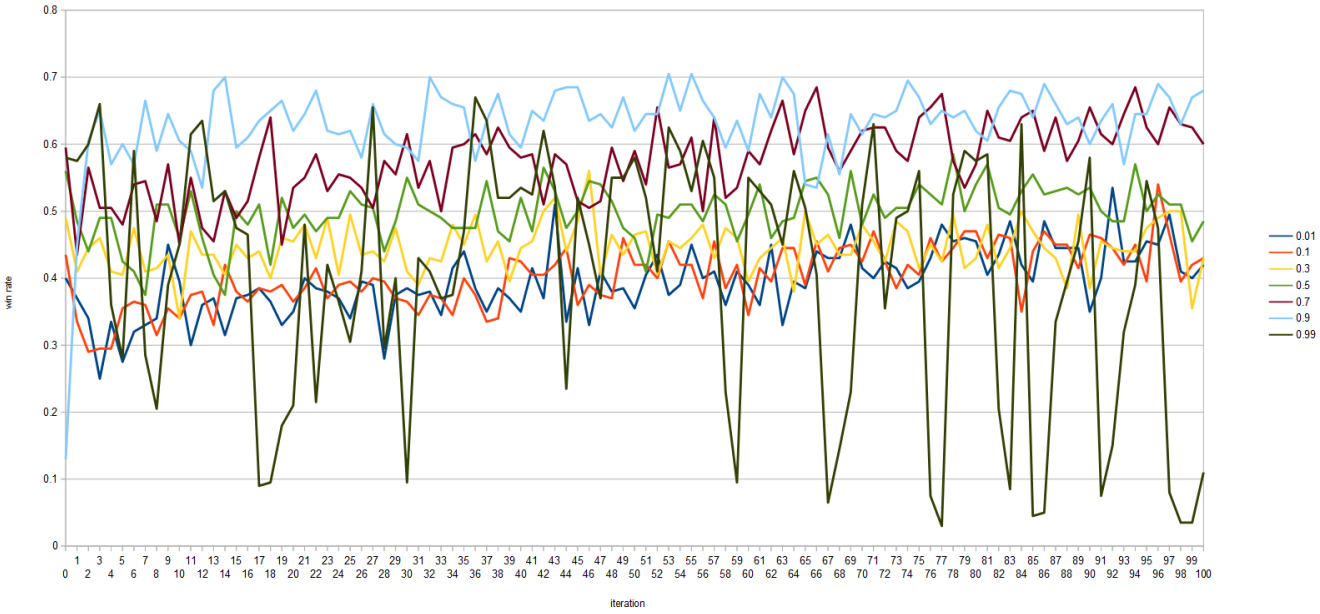


Figure 12: Win rates of two ϵ-*greedy* agents trained pairwise with different settings for γ over 200 games per iteration against the default UCT opponent.

Results with pairwise training are shown in Figure 12. Compared to the training against a random agent, we see little to no decrease of the agent's win rate over the series of iterations. As expected, a discount factor in the range of 0.7 to 0.9 seems to give the best results. We will use $\gamma = 0.7$ for most of the following experiments and some agent's trained with $\gamma = 0.9$ in the last experiment where we do an overall comparison of our different settings.

5.5 Epsilon

Similar to the discount factor, we are going to test different values for ϵ. Ideally, we want enough exploration to obtain a good value function but have enough exploitation to improve the current policy.

For this experiment, we used 100 training iterations, an update rate of $\alpha = 0.7$, 100k samples per iteration and $\gamma = 0.7$ to train two ϵ-*greedy* agents in the pairwise setup. The trained policy from each iteration is then evaluated against the default UCT agent with 256 iterations and $c = 0.1414$. In the first experiment, ϵ is set to a fixed value over all iterations. Afterwards, we will test the linear interpolation from Equation (10).

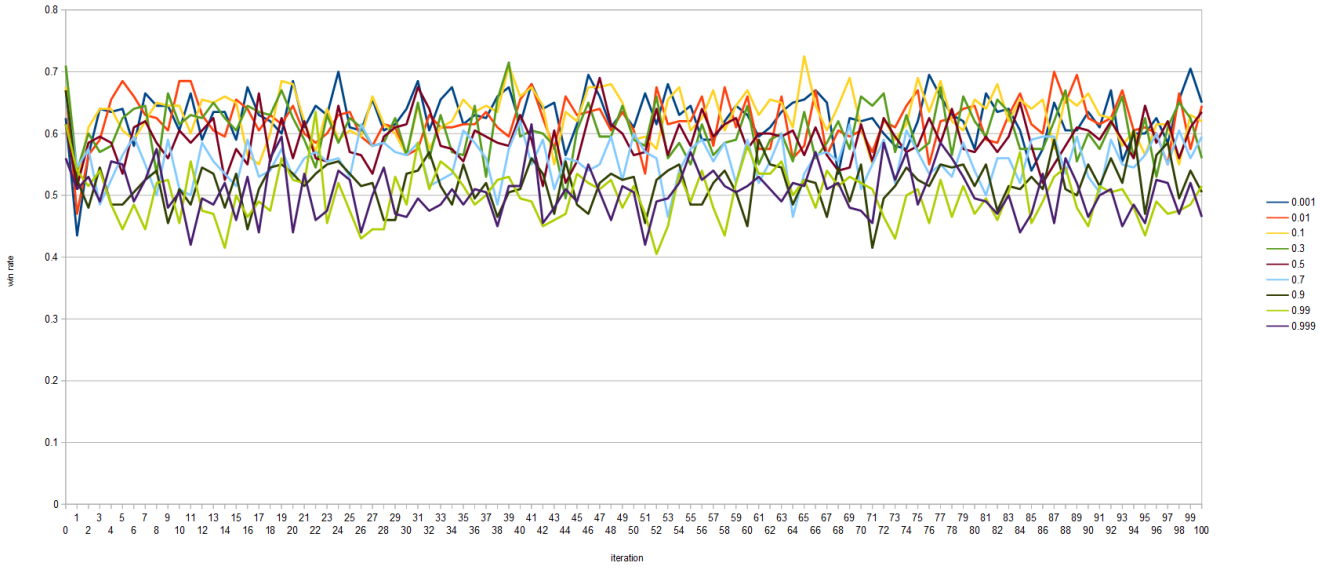


Figure 13: Win rates of two ϵ-*greedy* agents trained pairwise with different settings for ϵ over 200 games per iteration against the default UCT opponent.

Figure 13 shows the results of different settings throughout multiple iterations. As the policy is initialized with $\omega = \mathbf{0}$, the agent will always behave similar to $\epsilon = 1$ in the first iteration. Note, that the tested values are referring to the *training* setting of the specific iteration. As stated in Section 4.6, we will always use $\epsilon = 0$ when we evaluate the ϵ-*greedy* agent against other agents in order to maximize the policy exploitation. It would make little sense to include random actions if the agent uses a value function to estimate it's return.

By comparing the results for different ϵ, it seems that it is favorable to use a low value to exploit the policy early on. We also see little to no improve in the results over the sequence of iterations. Instead, the policies oscillate with little to no change after the first iterations. This strengthens our estimation that there is no single optimal policy for *Hearthstone* but a series of good policies and the learned policy depends on the sample bias of the current policy.

We used a second experiment with similar parameters to test the linear interpolation for ϵ from Equation (10).

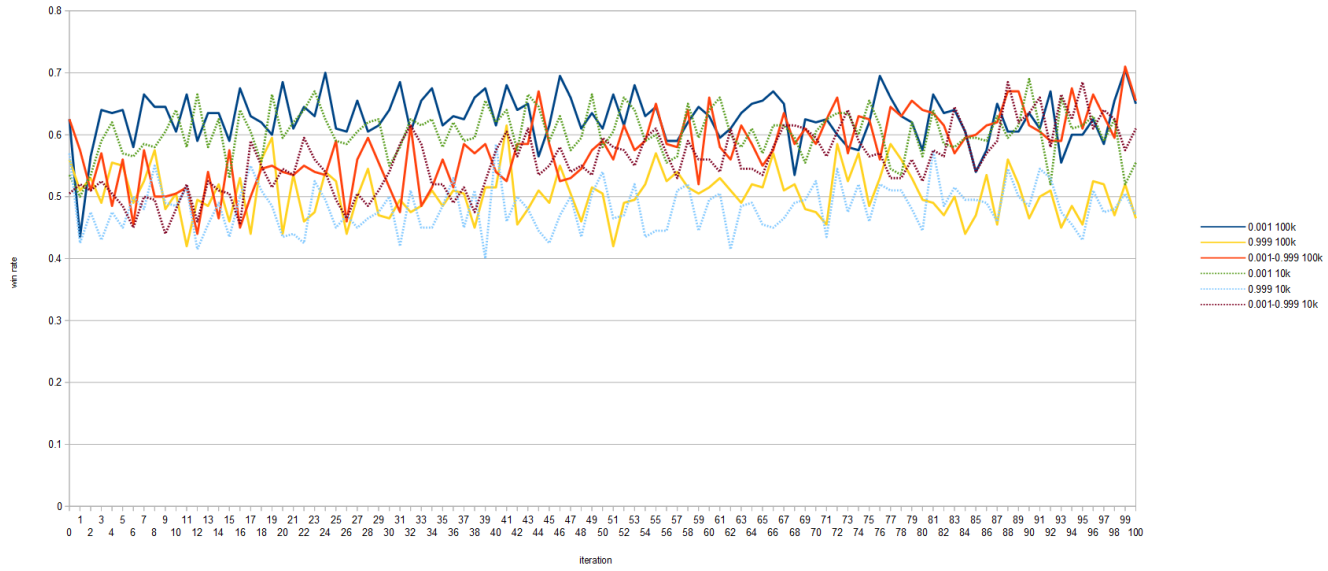


Figure 14: Win rates of two ϵ-*greedy* agents trained pairwise with different settings for ϵ over 200 games per iteration against the default UCT opponent.

Figure 14 shows the results. We used two sample sizes per iteration, 100k and 10k. We added the initial and final value as constant parameter as comparison to the interpolated value. The results are comparable to the previous experiment. We observe that the value of ϵ has no huge impact on the trained results, as long as it is low enough to favor a high exploitation. Also, the 10k sample setting with $\epsilon = 0.001$ performed almost as good as the setting with 100k samples per iteration.

Nevertheless, we decided to use $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$ in most of our experiments, as it results in a policy which is similar to one trained with a constant ϵ but leaves more space for early exploration in case it would be required, for example if we use an entirely different set of value functions where early overfitting would be an issue.

5.6 Gradient Based Policy Updates

We used comparable settings to the previous experiments to evaluate different values for α. The experiment was run with a pairwise training, 100k samples per iteration, $\gamma = 0.7$ and $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$.

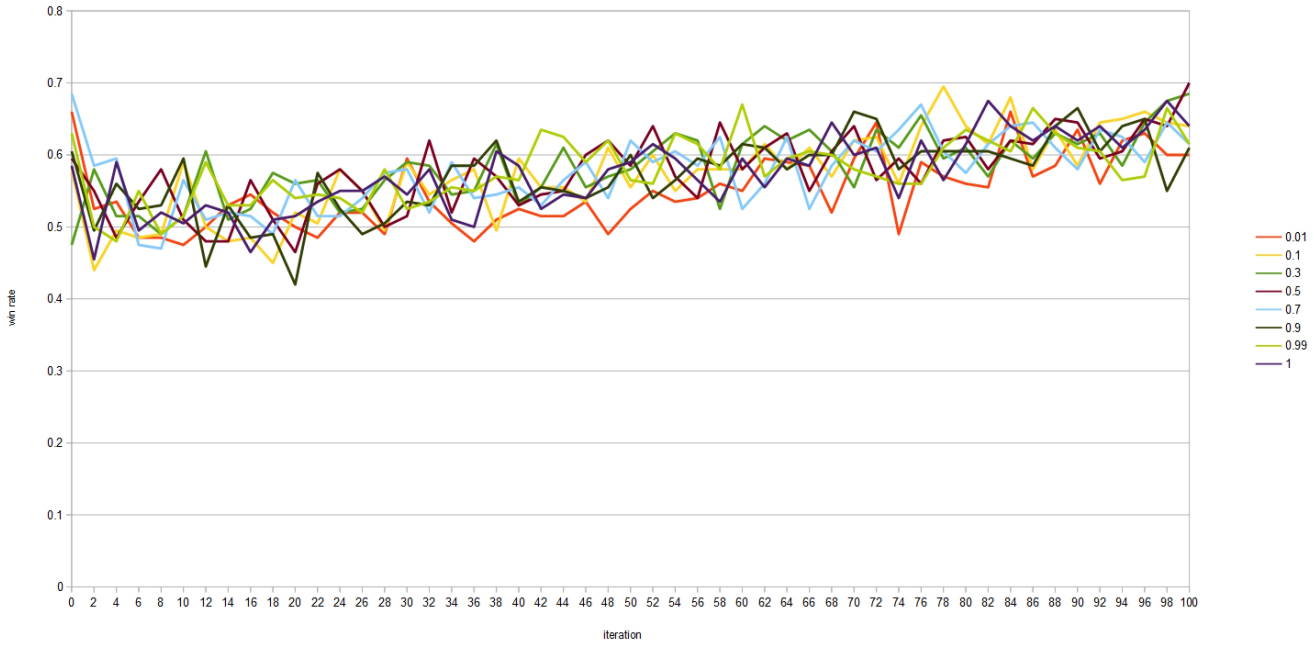


Figure 15: Win rates of two ϵ-*greedy* agents trained pairwise with different settings for α over 200 games per iteration against the default UCT opponent.

The results of this experiment are shown in Figure 15. It seems that the parameter α has little impact on the trained policies. A low value of $\alpha = 0.01$ lowers the update rate and results in a slower improvement of the policy but we see little to no impact in the overall policy improvement when comparing the initial and last iteration. A possible explanation for this observation is that the trained policy is biased by the sample distribution of the current iteration. Even with a very low update rate of 0.01, the policy changes the weight ratios large enough to alter the agent's win rates by up to 10% between two iterations. However, we also observe that all policies oscillate around the same win rate, which is an indicator for multiple *good* policies. This is fully explainable, as we train a general policy for the entire game and the strategy in *Hearthstone* is highly deck and card dependent. Therefore the policy is altered by the amount of observed card and deck combinations, optimizing on different aspects of the game in each iteration.

In the next step, we compared $\alpha = 1.0$ and $\alpha = 0.3$ and 1kk samples instead of 100k samples over 1k iterations.

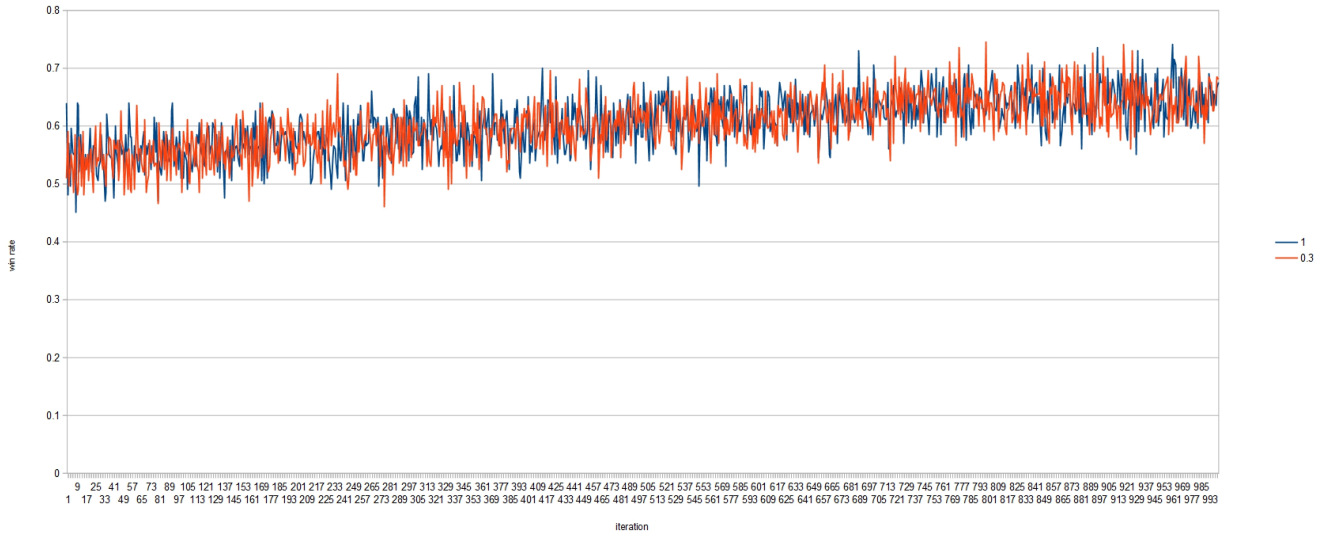


Figure 16: Win rates from ϵ-greedy vs an UCT opponent out of 200 games on each iteration, trained pairwise with $\alpha = 1.0$ and $\alpha = 0.3$, $1kk$ samples per iteration and $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$.

Figure 16 shows the second results. On some iterations, the win-rate drops or increases by almost 20. The overall performance was comparable to the previous experiment and the improvement of the policy was mainly dependent on the linear change of ϵ.

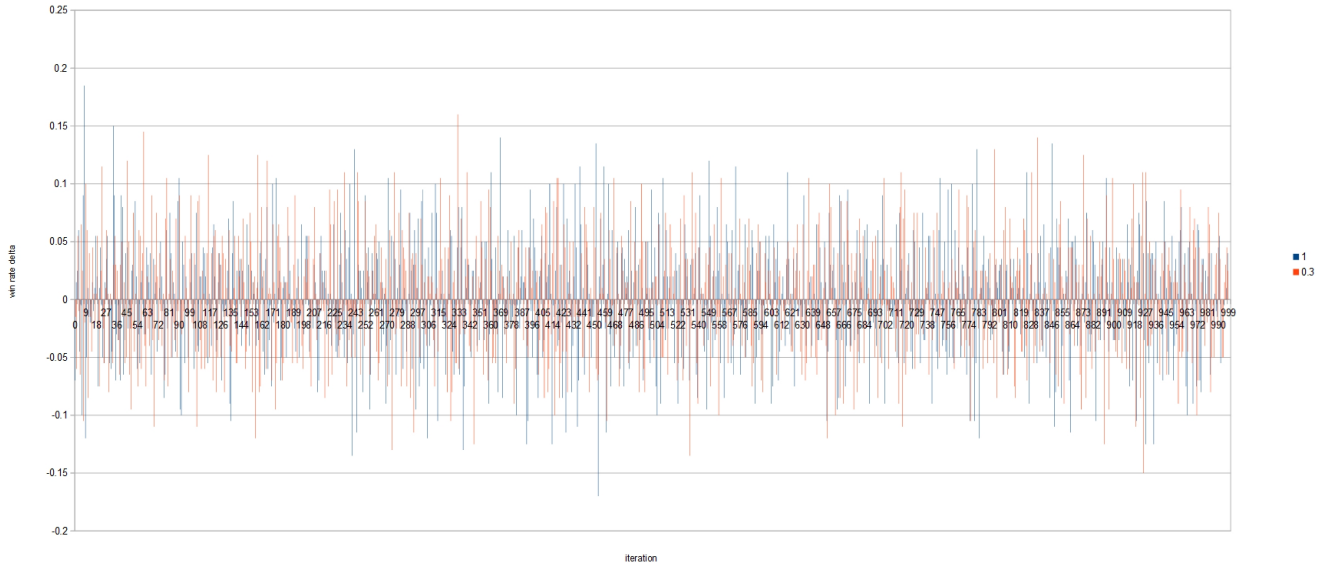


Figure 17: Win rate deltas from Figure 16. The value at iteration t is the win rate difference from t to $t + 2$.

Figure 17 shows the win rate delta from each iteration to the second next one, therefore comparing the win rate deltas for each agent. As we see, the win rate's change keeps almost the same during the entire training. We conclude that the gradient based policy update had little to none impact on the agent's performance. It seems that the trained policy is highly dependent on the observed card combinations and samples. For these observations, a good policy is found almost immediately in each iteration as long as enough samples are available.

5.7 Value Function Comparison

To evaluate the best features for our policy, we defined subsets of these value functions and used each subset to train a policy with it. Table 3 Gives an overview of the sets of value functions.

Name of the Set	Contained functions	Description
ALL	All player value functions from Table 1 and all minion value functions from Table 2	The entire set of value functions which is expected to cover the largest amount of gameplay features. If not stated otherwise, we always used the **ALL**-Set for ϕ^p.
HEALTH	*Player Health*	The baseline set, as the health maps to the win condition. With a lookahead of one, all other actions than direct damaging the enemy or healing the own hero will be selected random.
PLAYER_ONLY	All functions from Table 1	Player-centric play, where no information about the board is available. The *used Mana* and *total used mana* will still be an indicator for the value of a played card.
MINION_ONLY	All functions from Table 2	Completely board-centric play as only minions will count for value. Attacks on the enemy hero will only be performed when no other targets are left.
SIMPLE_GAME	The player value functions *Hand size*, *Board size*, *Deck Size* and *Health* and the minion value functions *Health* and *Attack* with all aggregations	Simplified version of the game, where weapons, abilities and mana efficiency are omitted.
MANA	The player value functions *Health*, *Mana*, *Max mana*, *Used mana* and *Total used mana* and the minion value function *Mana Cost*.	This set of functions focuses entirely on mana advantage and the usage of mana.
NO_ABILITIES	All functions except the minion value functions *Taunt*, *Windfury*, *Divine Shield*, *Frozen*, *Stealth*, *Immune* and *Charge*	Compared with the **ALL**-set, this set evaluates the impact of the common abilities from minions.

Table 3: Overview of the sets of value functions for ϕ^p.

In a first step, we trained each set with two *ϵ-greedy* agents, using 10k, 50k and 100k samples over 50, 30 and 30 iterations, $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$, $\gamma = 0.7$ and $\alpha = 1.0$. We then tested the trained policy for each iteration with 100 games against an UCT-opponent using 128 evaluations and $c = 0.1414$.

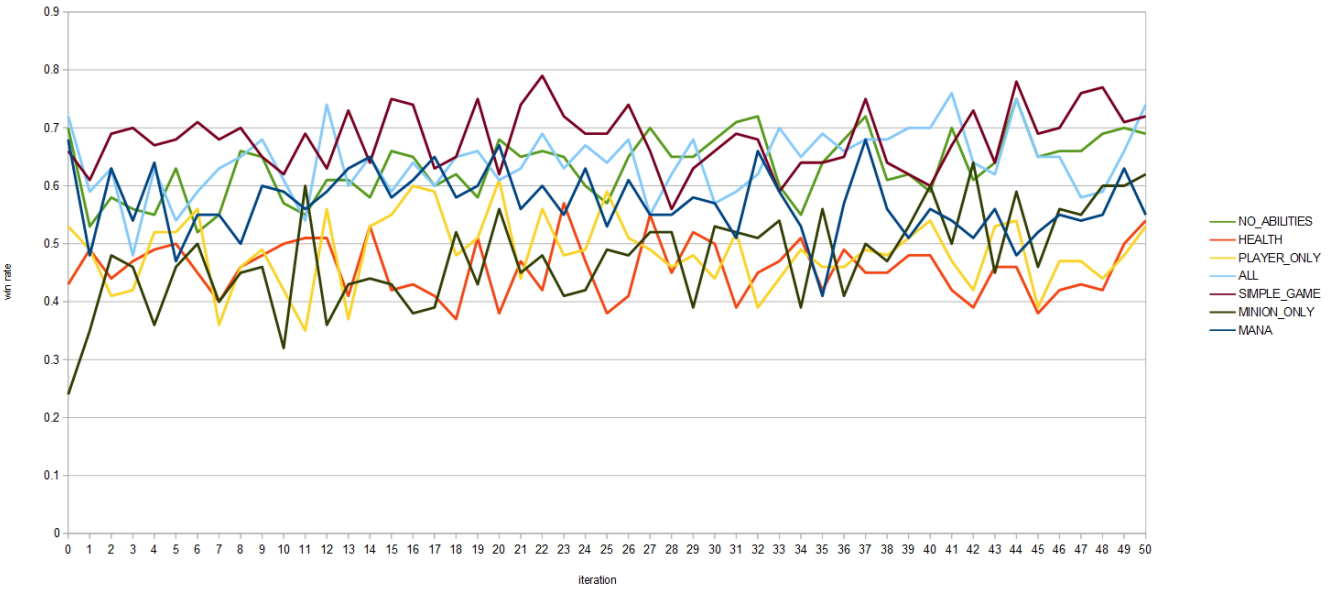


Figure 18: ϵ-greedy vs UCT with different sets of value functions and 10k samples.

Figure 18 shows the results with 10k samples. All sets performed way better than a random opponent, which won only 10% against the same UCT opponent. The **ALL** set did perform well but not best which may be the cause of the small number of used samples. **MINION_ONLY** and **PLAYER_ONLY** performed very similar but both worse than **ALL** which is the union of both. This is also expected, because the minions on the board are the most valuable asset on winning while the player value functions cover resource management and the win condition. **MANA** performed almost similar to **NO_ABILITIES** in the beginning but lost value as ϵ decreases and the policy stabilized. This matches the results from E. Bursztein who used machine learning to predict the next cards of the opponent [18]. They evaluated that *cumulative board mana advantage* and the *cumulative mana advantage* are the most predictive when deciding if a player will win. Both features keep track over the total mana spent on minions and the total mana spent, compared to the same values of the opponent.

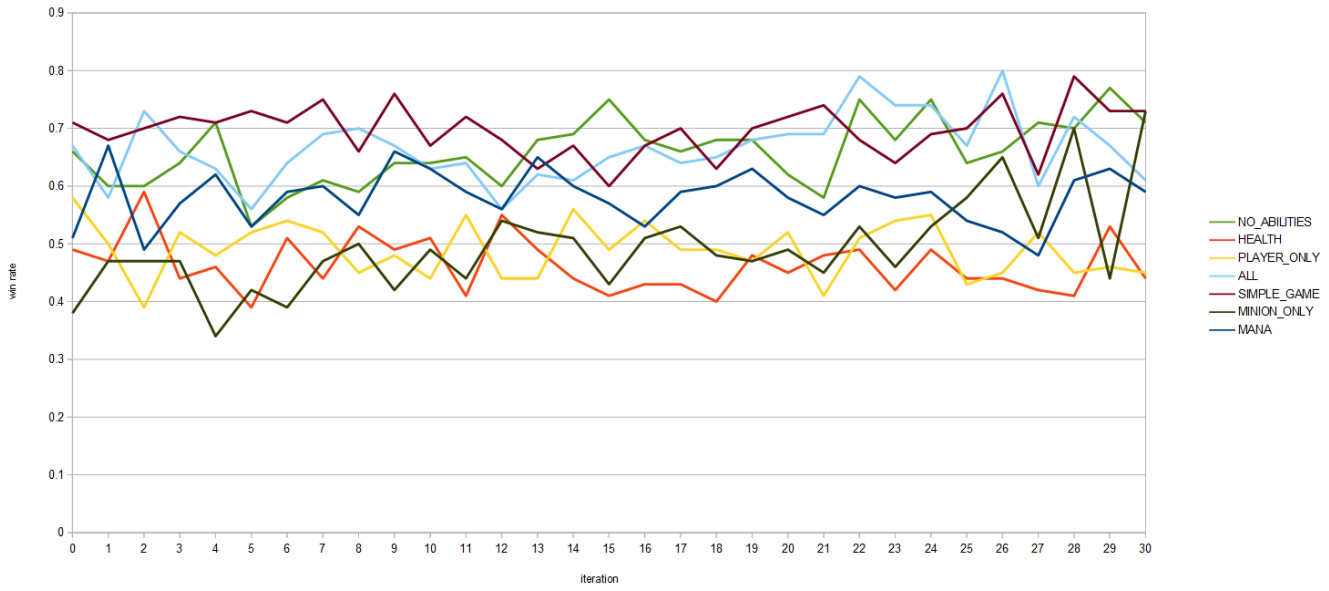


Figure 19: ϵ-greedy vs UCT with different sets of value functions and 100k samples.

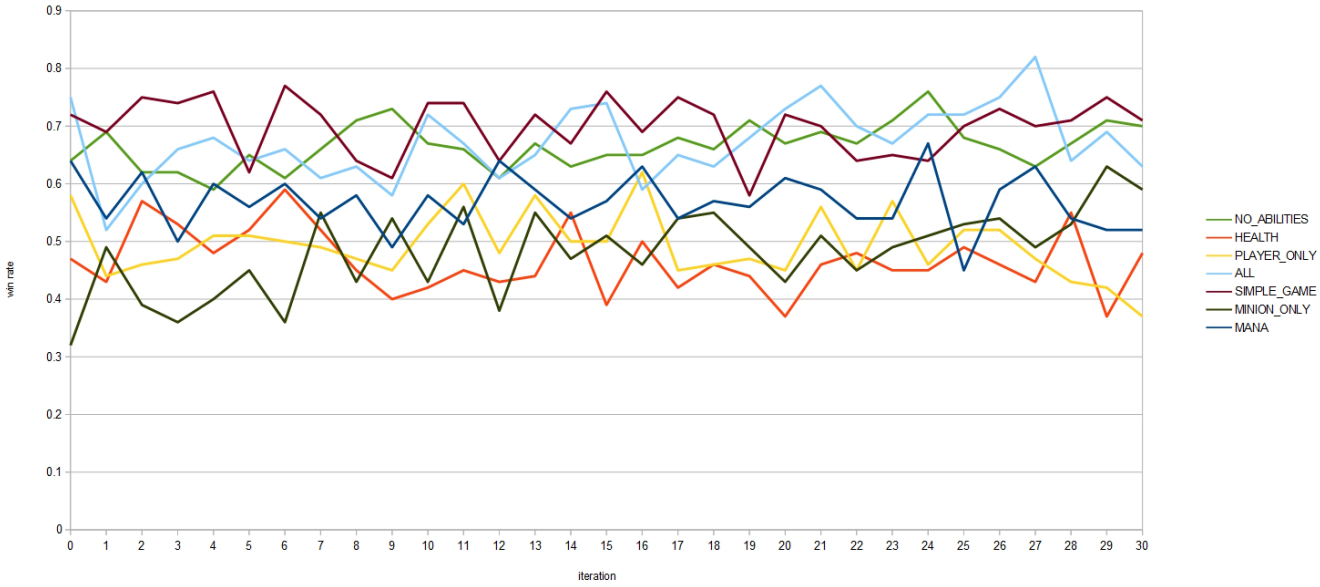


Figure 20: ϵ-greedy vs UCT with different sets of value functions and 1kk samples.

Figure 19 and Figure 20 show the results for 100k and 1kk samples. Compared to Figure 18, the average gain through the iterations is smaller. Even with the high ϵ of 0.999, the amount of samples seems to be high enough to allow to train a reasonable policy. The impact of the higher sample count is easy to spot on the the **ALL** set, which reached a higher win rate as the number of samples raised.

5.8 Squared Features

We will now evaluate the performance of the squared value functions as introduced in Section 4.3 for the **ALL** set. This consists of 124 total features, therefore the squared set has a total of $124 + \frac{124^2}{2} = 7812$ features. We expect to need more training samples for ϕ^s to reach a similar performance compared the previous experiments with the default set.

We started evaluated with 50k and 100k samples in each iteration and common settings for all other parameters.

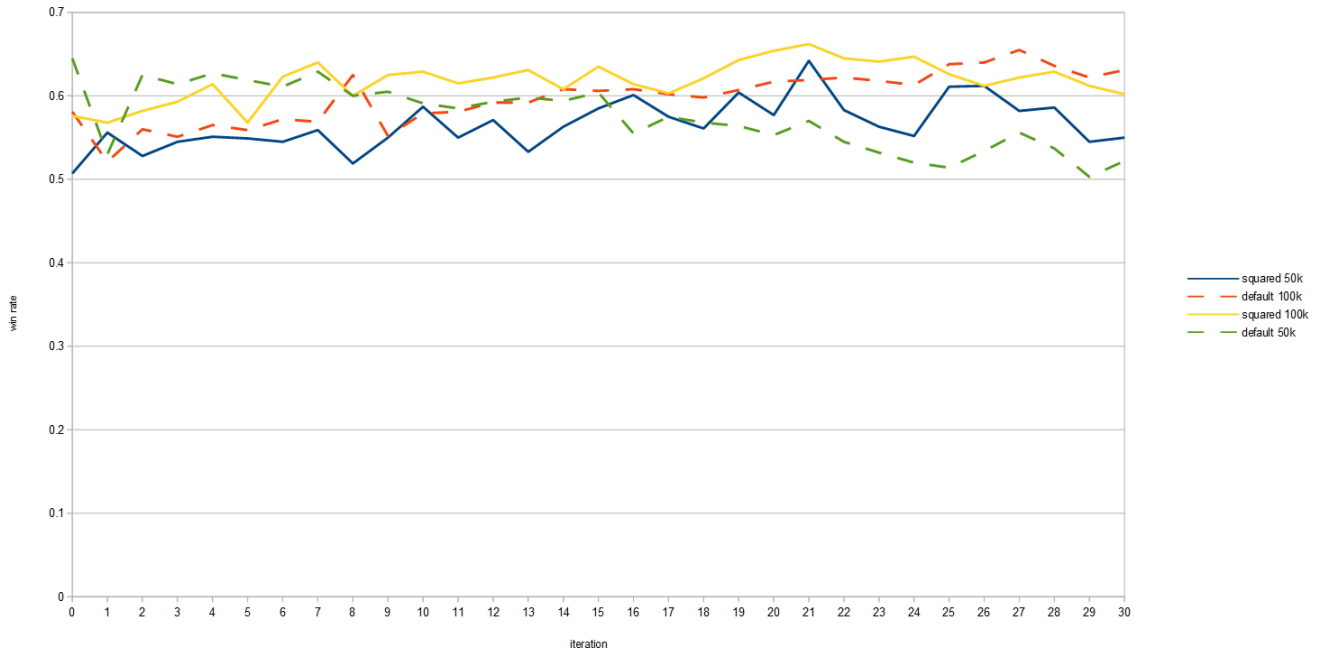


Figure 21: Win rates from ϵ-greedy agent vs an UCT opponent wit 256 evaluations and $c = 0.1414$ out of 1000 games on each iteration, trained with squared value functions (blue) and default value functions (red), $\alpha = 0.7$, $50k$ and $100k$ samples per iteration, ϵ from 0.999 to 0.001 and $\gamma = 0.7$.

Figure 21 shows the results of. The win-rate of the squared set with 50k samples is over the baseline of 0.5 on each iteration but not very stable compared to the default set. This can be explained by the low amount of samples which leads to less stable parameters, as shown in the next figures.

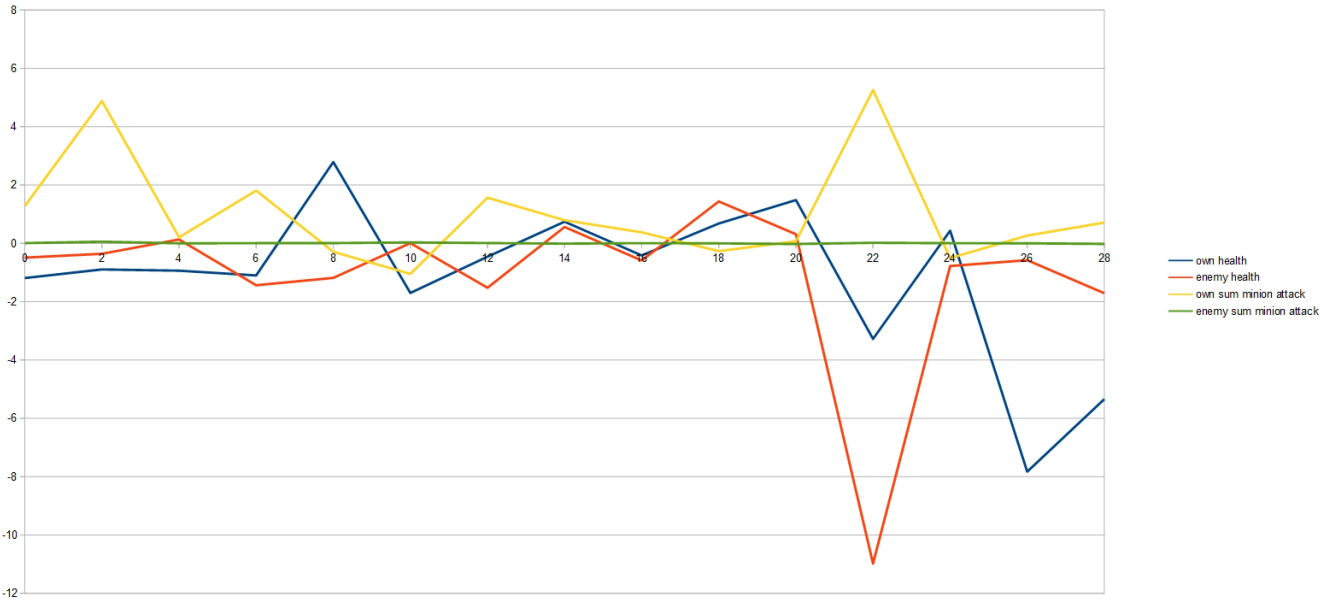


Figure 22: Selected features from the first agent with 50k features of Figure 21, trained with the *squared* set. The second agent was trained in the odd iterations, therefore there are only values at even iterations.

We conclude that 50k or 100k samples are not enough to train stable parameters for the value functions, which is also observable in Figure 22 where we compared the trained parameters ω_j for four selected value functions. All four parameters are unstable and change between being positive and negative. They also fluctuate with their counterpart from the opponent's value. This is only a hand-selected example but was true for almost every parameter in the entire set.

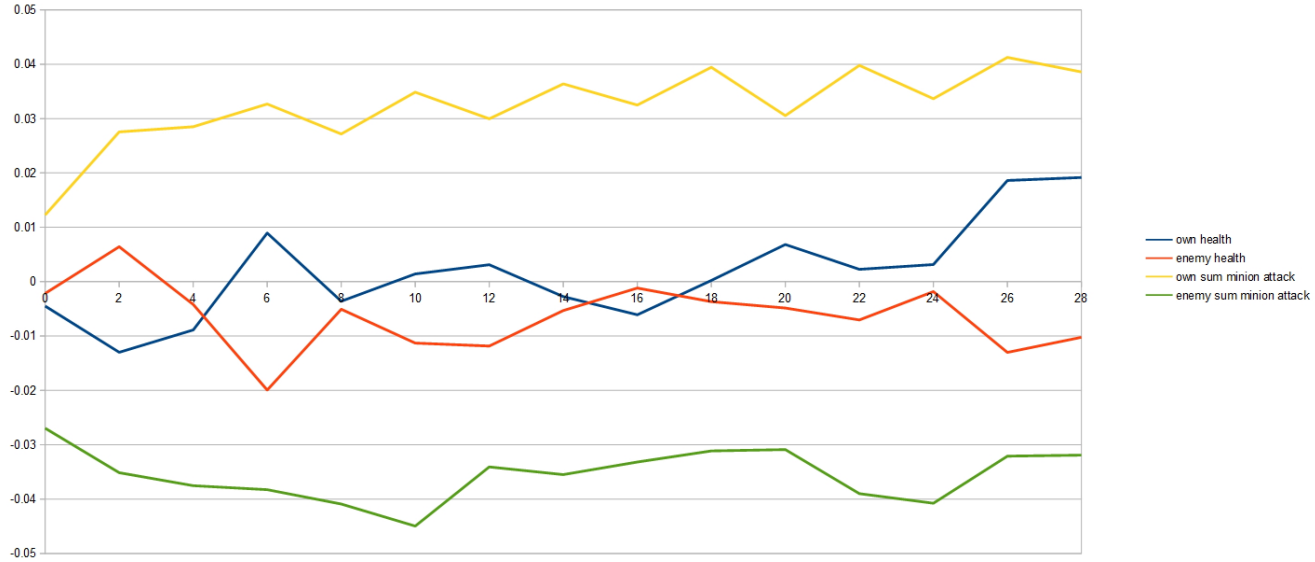


Figure 23: Selected features from the first agent of Figure 21, trained with the *default* set. The second agent trained in the odd iterations, therefore there are only values at even iterations.

The same parameters from the *default* set are much more stable, which is shown in Figure 23. Values of the *own health* and *own sum minion attack* features are rated higher than the values of the opponent with the exception of the begin and the 16th iteration.

5.9 Importance Sampling

The importance sampling approach which we described in Section 2.7 has one huge disadvantage compared to the previous approaches. Namely, samples from previous iterations are re-used and the parameters w_k for the different probability distributions have to be stored with them. Our other approaches did not require a sample re-use and could simply discard each sample after it went through the loop of Algorithm 1.

Each sample consists of two entire *Hearthstone* game states, one action and one reward. Each Hearthstone state consists at least of the 30 cards from each players deck plus extra information like the player class, health and damage values of the minions and the heroes and more. Even if we assume a lower bound of 200 properties/objects per game state (which is by far too low) and 64 bytes per object (which is also too low as a lot of these properties are implemented as fully qualified Java objects with internal ids and additional informations), then only the game states of the sample would require $200 * 2 * 64 = 25.6$kB. Therefore all samples of a 100k sample iteration would take $2.56GB$ of storage. As mentioned, these are just lower bound assumptions, we observed much higher numbers in our experiments. Therefore we used a database to persist samples internally between iterations.
Compared to a simple stream of samples, this method was highly expensive in terms of computation time due to the overhead of persisting and re-loading the samples in each iteration.

In the following experiments, we used an update rate of $\alpha = 0.7$, $\gamma = 0.7$, $\epsilon^a = 0.999 \rightarrow \epsilon^b = 0.001$ and 100 iterations. We always trained with two *ϵ-greedy* agents, therefore the even iterations maps to the first agent and odd iterations to the second agent.

The importance sampling variant used 1000 new samples per iteration. Therefore $1000j$ samples in the jth iteration and $\sum_{n=j}^{100} 1000j = 5050000$ samples in total with an average of $\frac{5050000}{100} = 50500$ samples per iteration.

We compared this series **IS1k** against the default training without sample re-use. For this comparison, we used three different settings for the sample count. One with the same amount of 1000 new samples per iteration **Def1k** which will use around $\frac{1}{5}$ of the amount of samples of **IS1k** in total. We also used a series with the same average of 50500 samples through all iterations **Def50.5k**. The largest series in our comparison is **Def5.05kk** which uses the same amount of samples in each iteration as the importance sampling set uses in total.

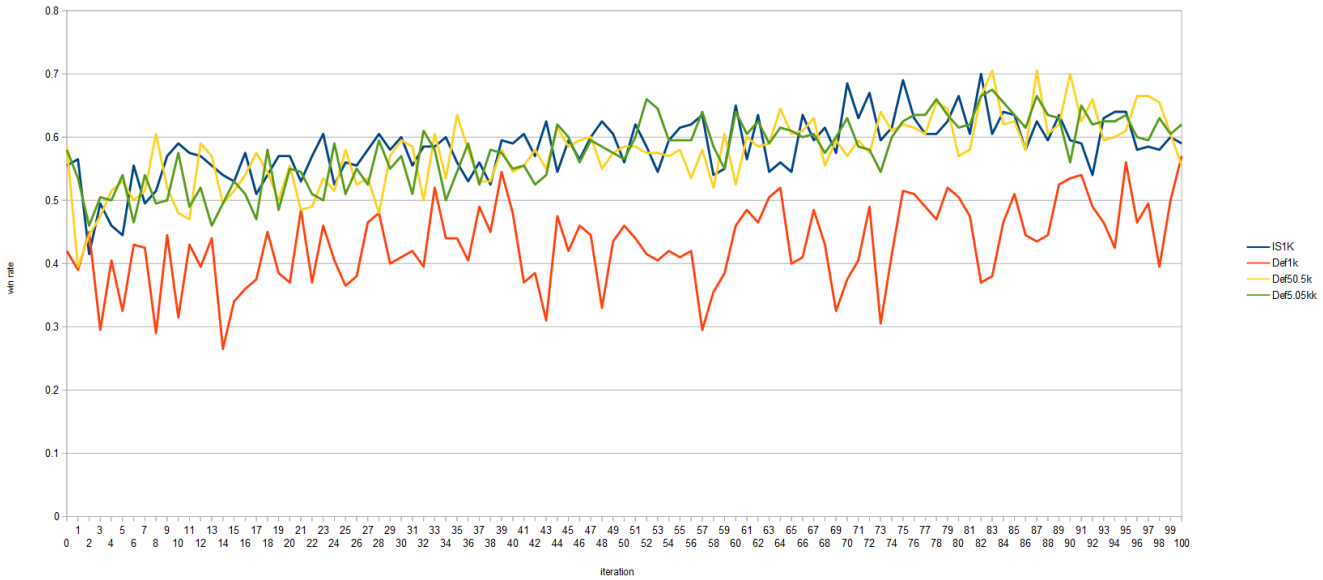


Figure 24: Importance sampling vs default training. Results were tested against the default UCT opponent with 200 games per iteration.

Figure 24 shows the results for one run. We observe, that the re-use of sample leads to a comparable performance than using the average number of samples in each iteration.
We conclude, that the importance sampling approach does not work very well in our environment as it is much cheaper to generate a larger set of samples than to re-use the old ones.

5.10 Final Comparison

So far we have only compared single combinations of two agents in order to evaluate our parameters or algorithms. In our last experiment, we are going to compare a larger amount of agents against each other with the goal to find an overall best solution or spot pairs of strategies, where one agent clearly outperforms another one.
If not stated otherwise, all agents in the final evaluation are *ϵ-greedy* agents with $\epsilon = 0$ and ω being the result of the last training iteration from one of the previous experiments.

Short label	Feature set	Samples/ Iteration	Training iterations	γ	α	ϵ	Training mode
DISC_0_9	ALL	100k	100	0.7	1.0	0.999 → 0.001	pairwise
F_ALL	ALL	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_HEALTH	HEALTH	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_PLAYER_ONLY	PLAYER_ONLY	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_MINION_ONLY	MINION_ONLY	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_SIMPLE_GAME	SIMPLE_GAME	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_MANA	MANA	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
F_NO_ABILITIES	NO_ABILITIES	1kk	30	0.7	1.0	0.999 → 0.001	pairwise
SQ_100k	ALL (squared)	100k	30	0.7	0.7	0.999 → 0.001	pairwise
IS_1k	ALL	1k (re-used)	100	0.7	0.7	0.999 → 0.001	pairwise + Importance Sampling
DEF_5_05k	ALL	50.5k	100	0.7	0.7	0.999 → 0.001	pairwise
VS_UCTk	ALL	100k	100	0.7	0.7	0.999 → 0.001	against **UCT_256**
UCT_256	none (UCT agent)	-	-	-	-	-	-
UCT_1024_10	none (UCT agent)	-	-	-	-	-	-
RAND	none (Random agent)	-	-	-	-	-	-

Table 4: Overview over all agents in the final evaluation.

Table 4 gives an overview over all agents we used. Columns refer to the training parameters which were used to train the final weight parameter ω for this agent. The first agent is **DISC_0_9** which used a discount factor of 0.9 in the first experiment. The second set of agents, labeled with **F_*** are the last ones we used in the evaluation for different sets of value functions from Section 5.7.
Third, the **SQ_100k** agent use the squared value functions which we trained with 100k samples in Section 5.8.
The agents **IS_1k** and **DEF_50_5k** are drawn from the importance sampling experiment of Section 2.7.
To get a better comparison against the pairwise training, we added the **VS_UCT** agent, which was trained against **UCT_256** instead of another *ϵ-greedy* agent.

We also added the UCT agents **UCT_256** with 256 evaluations and $c = 0.1414$, which was our default opponent in the experiments, and **UCT_1024_10** with 1024 and $c = 10.0$ as a stronger opponent (see Section 5.1). The bottom baseline will be the random opponent, **RAND**.

To achieve a fair comparison between all agents, we use the *round-robin tournament*, a matchup algorithm where each of n competitors has to play against each other competitor. This distinguishes the *round-robin tournament* from an *elimination tournament* or *knockout system* where one competitor may drop out early [19].
The scheduling algorithm for a *round robin tournament* assigns each competitor a number. It then splits the competitors in two groups and each group member plays against one member of the other group. After each round, the same competitor in one group is fixed and all other competitors rotate clockwise or counterclockwise. This process is repeated until all competitors played against each other.

```
Round 1:
        1       2       3       4
        5       6       7       8

Round 2:
        1       5       2       3
        6       7       8       4

Round 3:
        1       6       5       2
        7       8       4       3

...

Round 7:
        1       3       4       8
        2       5       6       7
```

Listing 1: Round robin schedule example.

Listing 1 shows an example schedule with 8 competitors. In general, if there are n competitors, a *round-robin tournament* requires $n-1$ rounds with a total of $\frac{n}{2}(n-1)$ pairs. If the number of competitors is odd, one competitor would have no opponent. The default solution in this case is to add a dummy competitor which does not play at all.
We have also to distinguish the term *game*. In the *round-robin tournament* one *game* is the match between two competitors. In the *Hearthstone* framework, *game* refers to a single hearthstone match which is obviously no informative value for the performance of an agent. For each pairing in the *round-robin tournament*, we use a total of 1000 *Hearthstone* games and use the win rate as result.

	DISC_0_9	F_ALL	F_HEALTH	F_PLAYER_ONLY	F_MINION_ONLY	F_SIMPLE_GAME	F_MANA	F_NO_ABILITIES	SQ_100k	IS_1k	DEF_50_5k	VS_UCT	UCT_256	UCT_1024_10	RAND
DISC_0_9	-	0.488	0.793	0.779	0.662	0.540	0.698	0.510	0.636	0.601	0.518	0.492	0.627	0.387	0.981
F_ALL	-	-	0.806	0.763	0.621	0.542	0.670	0.517	0.632	0.620	0.519	0.486	0.670	0.419	0.983
F_HEALTH	-	-	-	0.461	0.326	0.226	0.349	0.217	0.307	0.283	0.203	0.211	0.403	0.205	0.932
F_PLAYER_ONLY	-	-	-	-	0.343	0.277	0.390	0.242	0.354	0.330	0.242	0.261	0.368	0.196	0.932
F_MINION_ONLY	-	-	-	-	-	0.388	0.545	0.395	0.553	0.476	0.385	0.356	0.487	0.285	0.940
F_SIMPLE_GAME	-	-	-	-	-	-	0.598	0.469	0.577	0.548	0.471	0.467	0.663	0.420	0.984
F_MANA	-	-	-	-	-	-	-	0.320	0.455	0.417	0.344	0.336	0.462	0.264	0.960
F_NO_ABILITIES	-	-	-	-	-	-	-	-	0.660	0.593	0.484	0.487	0.631	0.440	0.973
SQ_100k	-	-	-	-	-	-	-	-	-	0.471	0.387	0.377	0.506	0.294	0.971
IS_1k	-	-	-	-	-	-	-	-	-	-	0.396	0.387	0.579	0.343	0.981
DEF_50_5k	-	-	-	-	-	-	-	-	-	-	-	0.475	0.668	0.427	0.982
VS_UCT	-	-	-	-	-	-	-	-	-	-	-	-	0.666	0.430	0.981
UCT_256	-	-	-	-	-	-	-	-	-	-	-	-	-	0.305	0.955
UCT_1024_10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.985
RAND	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 5: Final results of the *round-robin tournament*. Results are stored row-column wise, therefore each cell contains the win rate $\frac{\text{wins row agent}}{\text{wins column agent}}$.

Table 5 shows the results of the *round-robin tournament*. As we assumed in Section 5.7, the selection of features has a huge impact on the agent's performance. However, the quality of these features is more important than the quantity. Taking the win rates of **F_ALL** and **F_SIMPLE_GAME** as example. The second one has only $\frac{1}{6}$ as many features as the first set but the win rates are comparable and the smaller set performed only a bit worse. We make the same observation on the squared set **SQ_100k**, which is comparable to **F_MINION_ONLY**. Sample re-use via importance sampling had also little effect, as we see at the results of the **IS_1k** agent which did not even reach the same win rates as **F_ALL**. One reason for this result could be the little amount of total samples for too many dependent features.
A more surprising result was the agent **VS_UCT**, which we trained against the static agent **UCT_256** in each iteration instead of *self-play* against another ϵ-*greedy* agent. The win rates of this agent were superior to all other ϵ-*greedy*, however the delta to **DISC_0_9** and **F_ALL** was very low. We can therefore conclude that a good policy can also be trained against a strong opponent instead by *self-play*, as the agent performed also well against the other ϵ-*greedy* agents.
Finally, it seems that the ϵ-*greedy* agent with a lookahead of only one state caps it's win rate against **UCT_256** at ≈ 0.67. We had single iterations in our training which achieved around 0.72 but this seems to be the highest possible result with our selection of value functions and training setup. For the stronger UCT opponent **UCT_1024_10**, the win rates fall below 0.45. Nevertheless, this UCT agent requires around 1000 to 2000 times of the computation time the ϵ-*greedy* agent requires to decide for a single action.

6 Related Work

Before we conclude our results and give an outlook for future work, we want to compare our work to related projects with the focus on differences to these. There are many popular projects which applied *reinforcement learning* to different games, each with different focus on the learning algorithm, the actual playing algorithm or the domain problems, introduced by the game's environment.

6.1 TD-Gammon

TD-Gammon was developed 1992 by G. Tesauro [15] and is a popular algorithm which used *TD*-learning to train an artificial neural network to play the game *backgammon*.
Backgammon is a two-player game, where both players move their checkers in opposite directions over the track. The number of allowed steps is obtained by a two dice roll on each ply with the restriction that an entire dice must be used for one checker. Winner is the player which moves all of his checkers first to the opposite side of the board. The game complexity of *backgammon* is increased by two mechanics. First, if a player moves one of his checkers to a position which is occupied by exact one checker of the opponent, the opponent's checker is "hit" and moved back to the start of the opponent's track. It must then re-enter the board, before the opponent may move other checkers. The second main mechanic of *backgammon* is a blocking strategy which prevents the described loss of a checker. If one player stacks two or more of his checkers at the same position, the opponent is not allowed to place a checker at this spot. Both mechanics lead to different complex strategies, for example to form blocking structures to suspend the opponent's progress on the board.
With an amount of 10^20 possible states, a table lookup for *Backgammon* was not feasible. On the other hand, the amount of possible dice combinations (21) and legal moves (≈ 20) on each ply lead to a high branching factor of ≈ 400. Compared with $8-10$ for checkers and $30-40$ for chess, this explains why a brute-force deep search works bad on *backgammon*. A lot of the previous solutions to *TD-Gammon* relied on heuristics and expert knowledge.

TD gammon uses it's neural network as evaluation function and evaluates all possible legal moves on each turn, in order to select the move with the highest expected value. With a two-ply lookahead and a *greedy* use of the value function, the way *TD-Gammon* played was similar to other approaches.

The innovation of *TD-Gammon*'s was a good positional judgment which was independent of human knowledge but self-trained. Abstaining from the use of human experts may seem bad in a first place but it also omits bias or prejudices of human strategies. In fact, *TD-Gammon* has lead to an advance in *backgammon* theory especially in the opening play where it discovered some superior moves compared to the traditional strategies.
TD-Gammon used a multilayer perceptron (MLP) as architecture for the neural network. The MLP works as a nonlinear function approximator, where the input values are single nodes which propagate their value through on or more layers of *hidden* nodes. The output is then aggregated to one ore more nodes in the final layer. Each connection between two nodes is parameterized with a weight to modify the value during the propagation. The internal weights can be updated using *backpropagation*, based on the error of the output compared to an expected result. Therefore MLPs are popular algorithm for *supervised learning*. *TD-Gammon* updated the weights using the *TD(λ)* algorithm with the following update rule.

$$\omega_{t+1} - \omega_t = \alpha (Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_\omega Y_k \tag{17}$$

Y_t is the predicted output vector of the neural network, in the case of *TD-Gammon* it is a four component vector with the estimates of either black or white winning or black or white winning with a gammon (without loosing a checker during the game).
The weights were initialized randomly and *TD-Gammon* learned the game by playing against itself and choosing moves for both players. After several thousand training games, the play strength of *TD-Gammon* reached the level of an intermediate human player. One weakness of *TD-Gammon* was a bad endgame play, compared to a decent positional play in the early and midgame. The endgame of *backgammon* requires a higher lookahead and a more analytic approach which was not covered by the value function with a two-ply lookahead.

We used a similar approach with self-play as described in Section 4.7 but used a linear approximation value function with manually selected features instead of a neural network. It is possible that a similar approach with a MLP would discover different features and other strategies for *Hearthstone*.

6.2 KnightCap

KnightCap by J. Baxter, A. Tridgell and L. Weaver [20] is a chess algorithm which combines *TD(λ)* learning with a *MinMax* game tree in the algorithm TDLeaf(λ). KnightCap continues the ideas from *TD-Gammon* by applying similar methods to the game *chess*. The mayor difference compared to *backgammon* is, that a deep lookahead with large exploration is required to evaluate reliable tactics. An agent for *Backgammon* works well with a shallow evaluation because of the high randomness due to the dice rolls. With a high number of required evaluations, the computation cost of the evaluation function has to be as low as possible. Therefore a neural network like in *TD-Gammon* is unlikely to deliver good results in a feasible time. The chess program *KnightCap* has a large board representation function with a lot of positional features. This value function has been trained with the TDLeaf(λ) algorithm for the usage in a *MinMax* evaluation.
Key idea of *KnightCap* was to define a value function $V(s, \omega)$ which is used as min or max function to score a leaf in a *MinMax* tree. The value function $\tilde{V}_d(s, \omega)$ denotes the evaluation result which is obtained by applying $V(s, \omega)$ to all leaf nodes of a minmax search, starting from s with depth d. The TD(λ) algorithm is then applied to $\tilde{V}_d(s, \omega)$ instead on $V(s, \omega)$, using the following update rule.

$$\omega_{t+1} = \omega_t + \alpha \sum_{t=1}^{N-1} \nabla \tilde{V}_d(s, \omega_t) \left[\sum_{j=t}^{N-1} \lambda^{j-t} \left(\tilde{V}_d\left(s', \omega_t\right) - \tilde{V}_d(s, \omega_t) \right) \right] \tag{18}$$

KnightCap was trained on an online chess platform where it started with an empty evaluation function (all parameters were set to 0) and increased from a rating of 1650 to 2100 over ≈ 300 games. Further improvements were made by adding an opening book for chess and better starting parameters for the chess material values.

It is likely, that our approach would profit from a further lookup or a *MinMax* tree search to select action sequences. However, the huge amount of different orders for action sequences combined with the randomness of *Hearthstone* make such an approach difficult to implement.

6.3 AlphaGo

AlphaGo, developed by the artificial intelligence company *Deepmind Technologies Limited*, is an algorithm for the popular board game *GO* [21]. It became popular in October 2015 as the first computer program to beat a professional GO player without handicap on a full sized (19x19) board. In March 2016, *AlphaGo* received a larger media attention when it played against *Lee Sedol*, who is considered as one of the world strongest professional *GO* players with the highest possible rank of *9 dan*. *AlphaGo* won four out of five matches which was the first time for a *GO* program to beat a *9 dan* professional player without handicap in tournament conditions.
This was an remarkable achievement for *artificial intelligence* as *GO* was considered a hard challenge for *machine learning*. The high branching factor of ≈ 250, a possible deep tree with long games taking ≈ 150 moves and an immense number of possible states made *GO* hard to solve with conventional approaches like *Monte Carlo tree search*, *Alpha-Beta pruning* or search heuristics. In fact, it is very hard to estimate the value of a checker/stone in *GO*, compared to the material values in chess, because it dependents on it's surroundings and all stones have the same value by default.

Prior to *AlphaGo*, the most successful *GO* programs were based on *Monte Carlo tree search*. *AlphaGo* improved this approach with *reinforcement learning* techniques, like training a policy and a value function to estimate good and bad actions without the need for multiple *Monte Carlo* rollouts. The algorithm used neural networks for both functions, which worked well for pattern based tasks like facial recognition and would also perform well to analyze a board situation in go, which is also a pattern problem on 19x19 positions and 3 possible values for each position (white stone, black stone or empty). *AlphaGo* uses two neural networks to reduce the depth and breadth of the *Monte Carlo* search tree.

These networks were trained in a pipeline, which combined *supervised learning* from expert knowledge with *reinforcement learning* and self-play. The *SL policy network* $\pi_\sigma(s|a)$ is used to predict expert moves for a given board position. The network was trained via *supervised learning* from 30 million game positions of human games, played on the KGS Go server (a popular platform to play GO online). Output of $\pi_\sigma(s|a)$ is a *softmax* distribution over possible moves an expert would take in a given board situation. *The SL policy network* reached an accuracy around 55.7 to 0.57.
The second network was then initialized to the *SL policy network* using the same network structure and trained via *policy gradient learning* and self-play where the current version of the network played against a randomly chosen previous iteration of the network to avoid overfitting. This *RL policy network* $\pi_p(s|a)$ won around 80% of the games against the *SL policy network* without any tree search or rollouts. It also won ≈ 85% or the games against *Pachi*, an open source program that uses *Monte Carlo tree search* rollouts and is ranked *2 dan* on the KGS server.

In the last stage of the learning pipeline, a value function $V^\pi(s)$ was trained, to predict the outcome of a game position, if both players follow the policy $\pi_p(s|a)$. An optimal value function for *GO* is unknown, therefore *AlphaGo* used a value function for the best found policy $\pi_P(s|a)$. Similar to the *SL* and *RL* network, the value function was represented as a neural network with a similar structure. The main difference to the policy networks was a single output node for the expected value instead of a distribution

over action probabilities. This network was trained using samples of one game state and the final outcome using a *stochastic gradient descend* to minimize the mean squared error between the predicted and the corresponding value (comparable to TD methods). To avoid overfitting for entire game sequences, this network was trained with 30 million positions from distinct games, sampled from pairwise games of $\pi_p(s|a)$. The value network reached an accuracy comparable to *Monte Carlo rollouts* with $\pi_p(s|a)$ by taking only $\approx \frac{1}{15000}$ of the time.

AlphaGo combined these networks in a *Monte Carlo tree search* algorithm. During the expansion of the tree, the algorithm selects actions which maximize the resulting value, ranked with the prior probability and visit count of the edge. Leaf nodes are evaluated by $V^{\pi}(s)$ merged with the result of a fast rollout policy $\pi_r(s|a)$, which was also trained using supervised learning. The cost of this approach is a higher computation load compared to default search heuristics. Therefore *AlphaGo* was optimized to execute the tree simulations on a CPU and calculate policy and value networks in parallel on GPUs.

The huge advantage of this approach is that the general algorithm can easily be adapted to similar problems because the neural networks depend barely on domain knowledge. However, for domains like hearthstone, a direct input like the *GO* board positions to a network would not be possible and therefore would require a comparable feature selection to our approach in Section 4.2. Nevertheless, the elegant pruning and fast rollouts of the search tree using a policy and good value function makes the general approach very adaptable to other domains with similar conditions. *AlphaGo* is also a good example how initial *supervised learning* from expert knowledge can speed up a learning progress. It is likely, that our work would benefit from a good expert dataset for *Hearthstone* but as stated in Section 4.7, we were not able to obtain a comparable dataset. In any case, it would be an interesting task for future work to implement a tree search comparable to the *AlphaGo* approach for *Hearthstone*.

7 Conclusion

We pick up the initial goal again for our conclusion, to develop an approximating policy which is capable to play the game *Hearthstone* in a reasonable way.
In order to do so, we investigated the general problem domain of *reinforcement learning* using formal models like MDPs. We found the *least squares policy iteration* algorithm as fitting solution to learn an approximating value function. As we were unable to develop a state-action value function for *Hearthstone*, we adapted the algorithm for state value functions.
We examined the rules and game mechanics of *Hearthstone* to gain a basic understanding of important properties of the game. By using these features, we modeled an approximating architecture for *Hearthstone* with around 120 distinct features. To gain a better understanding of the impact of different feature types we splitted the features in 7 sets which map to different aspects of the game. We further defined an *ϵ-greedy* agent which makes use of this value function to optimize his expected outcome when selection actions in *Hearthstone*. The agent and the value function features were implemented in an existing simulation framework, which also included two *Monte Carlo tree search* implementations for the game.
We extended the framework further with an implementation of the *least squares policy iteration* algorithm and implemented a training schedule which used two agents in *self-play* or one agent which played against a fixed opponent. To handle the large state space of *Hearthstone* and use our samples more efficiently, we implemented approaches to re-use training data, like *importance sampling* and *gradient base policy updates*.

As we evaluated the different feature sets for our value function, we found that even a very simple set of features which generalizes the most important game features like health of a player reaches a noticeable win rate against a random opponent and was also capable to compete with a *Monte Carlo* agent on a low degree. The feature sets with more advanced features reached win rates ≈ 0.6 against the selected *Monte Carlo* baseline while using only a fraction of the same computation time.

However, we found that against stronger opponents, like a *Monte Carlo tree search* with more evaluations, our agent was outperformed. We assume the leak of an accurate lookahead as main reason for this observation, which prevents the agent to select the optimal action order for it's own ply and also hinders us to evaluate an accurate value of the state transition which is induced by the *EndPly* action, when the agent offers the game to it's opponent.
Another limitation was the decision to learn one global strategy for the entire game. As we observed in our experiments, the policy was biased very strong by the observed samples distribution. We were not able to train synergies between cards or strategies for unique decks, as this would have increased the feature space of the value function by a lot. *Monte Carlo tree search* does not suffer from this problem as it could theoretically play well with every deck, as long as the amount of evaluations is high enough to exploit the deck's mechanics. Overall, we were able to learn a well strategy which was capable to play *Hearthstone* on a reasonable level with random decks.
We assume that this global policy is the main reason why sample re-use via *importance sampling* and the gradient based updates hat little to none effect on our results and the policy oscillated between different good policies on each iteration. The increased update time for *importance sampling* did also not match the improvement in the results. In terms of computation time, it was cheaper to simply generate more samples per iteration than to store and update old samples. This could change if the sampling of the environment was more expensive, e.g. when training with other value functions like a linear approximating architecture or different opponents.

8 Future Work

Finally we want to sum up ideas for further research, which did not make it into this thesis for time or scope reasons.

8.1 Human Play

So far, the agents were only trained and evaluated against other agents. However, humans may select entirely different actions than a *Monte Carlo tree search* or an ϵ*-greedy* agent. As *Hearthstone* disallows automated play in it's game, a direct comparison is not possible. However, the simulation framework could be adapted as kind of *assistant* which gives the player hints for valuable action sequences without interacting with the game itself. This would require the simulation to be as close as possible to the actual game and must include the time limitations for a ply.
In addition, the observed games can be used as additional training data, where techniques like *importance sampling* would probably reach a higher potential as the amount of available samples is limited and also lower than in large *self-play* simulations.

8.2 Opponent Modeling

This involves the problem of the *EndPly*-value which we described in Section 4.5. If we can evaluate the opponent's move, we get a better estimation if it is worth to end a ply early.
To model the opponent's action, we would need a prediction model for his deck. In fact, such prediction systems exist and can reach a high accuracy on real *Hearthstone* matches [18], therefore such an approach is likely to be feasible.
The open question would be how the opponent's actions are modeled. One possible way would be a general policy, as we developed in this thesis which performs likely actions of the opponent. With a better prediction system for decks and strategies, we could also try to model the opponent's deck strategy in a separate policy.

8.3 Deck Optimized Policies

As mentioned, we did evaluate deck building or deck specific strategies. However, *Hearthstone* is a game where special strategies with decks are possible. One example are *combo* decks, which try to draw a

deck-specific card combination which allows them to win the game instant if played in the correct order. Other decks may try to set up a unique minion combination on the board to gain the maximum value from buff spell cards. There are even more specialized decks, which abuse the card-draw mechanic and try to overflow the opponent with card draw, in order to waste his cards as he can't hold more than 10 cards at once.
It is hard to impossible to model all these strategies in a simple linear architecture like Equation (4) without increasing the feature space by a huge amount. However, if we restrict our value function for one deck, made out of exactly 30 cards, we can generate more adapted features like *n-grams* for card or minion combinations from this deck.
As *Hearthstone* is also a game with a strong *meta*, therefore the strongest decks are aggregated from a small amount of cards, it would be possible to build an prediction system for the opponent's deck configuration. We could drive this even further and train one policy for one matchup of two decks. By aggregating the results, weighted with the confidence of each deck prediction, this would allow us to maximize our deck policy against the most common opponents.

8.4 Increased Lookahead / Tree Search

We have shown, that a policy with only 1-step lookahead can compete with a *Monte Carlo tree search* agent. However, with an increasing number of evaluations, *MCTS* starts to outperform the policy. It would be possible to implement an approach comparable to the *AlphaGo* algorithm, where the search tree is pruned and shortened by the application of a trained policy and value function.
The challenge for such an approach is the *rollout* of hidden information, the action order and random effects. Random effects can be handled to a certain degree, for example a card draw can be modeled as a node with an outdegree of 30 (the amount of cards in the own deck). Hidden information could be modeled with a prediction system and a confidence ranking for these kind of nodes in the game tree.
A more straightforward approach would be to include the trained policy directly in the *rollout* step of the *Monte Carlo tree search*, therefore making the prediction of a game result more accurate.

List of Figures

List of Tables

References

[1] M. Zopf, "A comparison between the usage of flat and structured game trees for move evaluation in hearthstone," Master's thesis, TU Darmstadt, Knowledge Engineering Group, Apr. 2015. [Online]. Available: http://www.ke.tu-darmstadt.de/lehre/arbeiten/master/2015/Zopf_Markus.pdf

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. [Online]. Available: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html

[3] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[4] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[5] R. Munos, "Error bounds for approximate policy iteration," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ser. ICML'03. AAAI Press, 2003, pp. 560–567.

[6] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Machine Learning*, 1992, pp. 279–292.

[7] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[9] D. A. Berry and B. Fristedt, *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability)*. Springer, 1985.

[10] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988. [Online]. Available: http://dx.doi.org/10.1007/BF00115009

[11] S. J. Bradtke and A. G. Barto, "Linear least-squares algorithms for temporal difference learning," *Machine Learning*, vol. 22, no. 1, pp. 33–57, 1996.

[12] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *J. Mach. Learn. Res.*, vol. 4, pp. 1107–1149, Dec. 2003.

[13] H. Kahn and A. W. Marshall, "Methods of reducing sample size in monte carlo computations," *Journal of the Operations Research Society of America*, vol. 1, no. 5, pp. 263–278, 1953.

[14] "Hearthstone: Heroes of warcraft," http://battle.net/hearthstone, accessed: 2017-01-31.

[15] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, Mar. 1995.

[16] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[17] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[18] E. Bursztein, "I am a legend: hacking hearthstone using statistical learning methods," in *IEEE Computational Inteligence and Games 2016*, 2016.

[19] W. Glenn, "A comparison of the effectiveness of tournaments," *Biometrika*, vol. 47, no. 3/4, pp. 253–262, 1960.

[20] J. Baxter, A. Tridgell, and L. Weaver, “Knightcap: A chess program that learns by combining td(λ) with game-tree search,” in *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998, pp. 28–36.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016.