university of groningen / faculty of science and engineering

Reinforcement Learning agents playing Hearthstone: The influence of state description when learning in a large, partially observable state space

Bachelor's Project Thesis

I.A. Tutea, i.a.tutea@student.rug.nl
Supervisor: Dr M.A. Wiering

Abstract: Many real world problems cannot be easily formalized into compact states that fully describe the real state of the world. It is thus important to look at how different ways of representing a state would influence the performance of AI agents. In this research we compare the performance of reinforcement learning agents which learn using different game state representations when playing "Hearthstone", an online collectible card game which has a state space size much larger than complex games such as Chess. Moreover, its states are only partially observable by the player and randomness governs over certain events. It is found that, when playing with very simple decks that do not require complex strategies, agents which have a simpler state description have a win rate of about 81% against a random player, whereas agents which have a larger state description only win 78% of the time regardless of the learning algorithm used. However, when playing with more complex decks that require synergistic actions in order to maximize their performance, agents with a simpler state description performed worse than before while the agents with larger state descriptions performed better with win rates of 88% against a random agent.

1 Introduction

Being able to make good decisions in complex, real-world situations is important for Advanced AI systems. Many of those situations are not easily formalized into compact states and many of these states cannot convey every single piece of relevant information for the decision making process. Thus it is important to look at and try to improve on the performance of AI agents when presented with large, partially observable state-space problems.

In this research we will look at how Reinforcement Learning (Sutton and Barto, 2018) agents perform when playing the online video game "Hearthstone", a discrete, imperfect-information card game with a very large state space. A conservative approximation of the state space size is 2.85×10^{51} (da Silva and Goes, 2017), which is approximately 10^{20} times more than common estimations of the number of reachable positions in chess.

In the recent years Hearthstone has been used more and more as a testbed for Artificial Intelligence research. Its great combinatorical complexity with partial randomness and the availability of online tools and resources such as the ones provided by the passionate developer community HearthSim (https://hearthsim.info) that help speed up the development make Hearthstone an attractive challenge for AI studies. Previous research looked into creating Hearthstone-playing agents using different kinds of approaches such as Monte Carlo Tree Search (MCTS) (Santos et al., 2017), combining MCTS with Supervised Learning algorithms (Swiechowski et al., 2018), attempting to improve MCTS policies with deep neural networks (Zhang and Buro, 2017) and Adaptive Neural Networks (da Silva and Goes, 2017). AI research that used Hearthstone as a testbed is not only related to the creation of game playing agents. Other studies looked also at deck creation using evolutionary

algorithms (García-Sánchez et al., 2016) and at predicting win chances in given game states (Grad, 2017).

Many of the previously developed AI agents for Hearthstone use tree search methods in order to look ahead at possible future states or outcomes. This approach was proven to work very well in deterministic games such as Chess (Silver et al., 2017; Baier and Winands, 2013) or Go (Silver et al., 2016), however in Hearthstone the particularly high branching factor caused by the partial randomness of the game makes constructing and evaluating trees inefficient in terms of time.

In this thesis two Reinforcement Learning algorithms will be explored in the context of agents playing Hearthstone: Q-learning (Watkins, 1989; Watkins and Dayan, 1992) and Monte Carlo learning (Sutton and Barto, 2018). These agents will only look at the current state of the game and, because the full state description size is very large, will only look at a subset of the information arranged and preprocessed in certain ways. This subset of information will be referred to further as the *state representation*.

We will then finally try to answer the following research questions:

- How would a trained agent that is only given a small subset of the information from a state perform compared to an agent that is given the full information?
- How would a Q-learning agent and a Monte Carlo learning agent perform against a random agent, heuristic-based agents and against game experts?

2 Methods

In this section a short introduction on Reinforcement Learning will be presented in subsection 2.1. Then, in subsection 2.2 there will be more details given about the game of Hearthstone. In subsection 2.3 we will look at the different state representations used and at the structure of the whole system that was used for training and testing.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of Machine Learning in which the learner is a decision-making agent that acts within an environment and receives feedback (rewards or penalties) for its actions when judged in terms of solving a certain problem or performing a certain task (van der Ree and Wiering, 2013; Sutton and Barto, 2018). By performing several training, trial-and-error runs in a given environment the agent will generate experiences from which it can learn to optimize its policy and thus maximize the total overall reward received per run. In the context of RL, a policy π is the pair (state, action-to-be-taken), or formally $\pi : S \rightarrow A$, $\pi(s) = a$.

2.1.1 Markov Decision Processes

In RL it is assumed that we have an underlying Markov decision process (MDP) at hand which does not necessarily have to be previously known by the agent (Wiering, 2010). A finite MDP consists of the following components (Wiering, 2010; Watkins, 1989; van der Ree and Wiering, 2013):

- The state space $S = \{s_0, s_1, s_2, ..., s_n\}$, which represents the finite set of possible states;
- A set of actions available that could be taken at each state $A(s_i)$;
- A transition function $P(s, a, s')$ that denotes the probability of an agent that is at state s and performs action a to end up in state s';
- A reward function $R(s, a, s')$ that gives the reward received by the agent when performing action a in state s and reaching state s' as a consequence;
- A discount factor $\gamma \in [0, 1]$ with the purpose of discounting the value of later rewards as opposed to earlier rewards.

2.1.2 Learning Algorithms

The main goal in RL is to have the agent learn an optimal policy that will allow it to choose the best action at any given state. One way to achieve that is to make the agent learn how good certain actions are at certain states. The value of any (state, action) pair is given by the $Q(s, a)$ function and it is

referred to as the Q-value. The $Q(s, a)$ function can be formally expressed as:

$$Q(s, a) = E[\sum_{i=0}^{\infty} \gamma^i r_i | s, a] \tag{2.1}$$

where r_i is the reward received at time i and $E[.]$ is the *expectancy operator* (Ross, 2007). The above formula can be read as "The Q-value of action a at state s is the expected discounted cumulative reward given that state and that action".

Some aspects change when discussing the case of playing a game against an opponent. The state resulted after making a move is not fully determined by the agent's action choice, but also by the opponent's action choice. In such a case it is possible that choosing the same action in the same state would yield a different reward and lead to different states based on the opponent's action. This is problematic because it means that the Q-value of a certain (state, action) pair will vary based on what the opponent plays. A solution to this is to keep a running average of all the previous Q-values of that pair. This is known as *the Q-learning algorithm* (Watkins and Dayan, 1992; Watkins, 1989; van der Ree and Wiering, 2013):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_a Q(s', a) - Q(s, a)) \tag{2.2}$$

where s' is the state the agent reaches after the opponent moves and $\alpha \in [0, 1]$ is the *learning rate* that regulates how much the old Q-value will be modified towards the new Q-value at each step. At each step some sort of lookup table should then be updated with the new Q-value of the given pair.

Another possible approach is to use the observed return G to compute an approximation for the expected reward of a (state, action) pair. Generally, G is given by:

$$G = \sum_{k=0}^{T} \gamma^k r_k \tag{2.3}$$

where T is the time of the final state. However, in our specific case, non-final actions yield a reward of 0 and we do not have discounting ($\gamma = 1$), which simplifies the above equation to $G = r_T$. Unlike in Q-learning, the updating of the Q-values is done not after every action, but after a complete episode (after a game, in our case). The estimation of the new $Q(s, a)$ after an episode is computed by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(G - Q(s, a)) \tag{2.4}$$

This is known as *Monte Carlo learning* (Sutton and Barto, 2018).

In both approaches, after a sufficient number of steps the lookup table will ideally hold (accurately estimated) Q-values for all the (state, action) pairs and thus the best policy that emerges will be given by:

$$\pi(s) = \arg\max_a Q(s, a) \tag{2.5}$$

2.1.3 Function Approximators

The solution above works well for problems with small, limited-dimensions state spaces. However, many problems, including the game of Hearthstone, have large, high-dimensional state spaces. For the latter situation the usage of lookup tables for storing Q-values becomes problematic because it would require a very large number of iterations before it can reach reliable estimations for every (state, action) pair. On top of that, if a trained agent will encounter the possibility of a (state, action) pair that it has never seen before it would be unable to generalize its previous information in order to estimate a realistic value.

One solution to this problem is to train a Neural Network to approximate the Q-value of (state, action) pairs instead of relying on a lookup table. Two types of Neural Network approximators will be used in this paper: a fully-connected Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN).

2.2 Hearthstone

"Hearthstone" is a popular online Collectible Card Game (CCG) published by Blizzard Entertainment. In this game players collect cards from card packs they buy, build decks and play against other people with the decks they have created.

2.2.1 Types of Cards

There are four main types of collectible cards in the game:

- **Minions**: Minions are cards that a player can play from their hand into the battle zone. Once in the battle zone, they can attack opponent minions in the battle zone or the opponent hero once per turn normally, but not on the turn they were played.

  In Figure B.1 (a) you can see a minion called Magma Rager. It costs 3 mana to play this minion, it has an attack value of 5 and its maximum health is 1 (or we can use the shorthand "3 mana 5/1" to refer to it). You can also see that this minion is part of the "Elemental" tribe, which is useful in case one has other cards that interact with Elementals. Some minions may have text on them that triggers certain events in certain conditions. One such example you can see in Figure B.1 (b). This is a "4 mana 2/4" minion that says "Battlecry: Draw an Elemental from your deck". Battlecry is a keyword that stands for "whenever this card is played from hand". There are other keywords and effect types in the game which are described more thoroughly in the appendix.

- **Spells**: Spells are cards that a player can play from their hand in order to trigger their effect. In Figure B.2 (a) you can see Arcane Explosion. A Mage class spell that costs 2 mana and deals 1 damage to all enemy minions when played. After a spell is played it is sent to the Graveyard zone.

- **Heroes**: At the beginning of a game, each player starts with a default hero, which has 30 health and 0 attack, based on their class that they chose in the deck creation phase. A hero always comes with a hero power attached to it. That is an effect that has a certain cost and that can usually be used once per turn. Unlike minions, there can only be a single hero card in the battle zone per player.

  Besides the default hero that a player starts with, a player can also have playable hero cards in their deck. In Figure B.3 (a) there is an example of a playable hero. When played, this card replaces the existing hero with this one and triggers its "Battlecry" effect. The old hero power is also replaced with the default hero power of the new hero (Figure B.3 (b)). The maximum health and current health remain the same as on the replaced hero, however, when played, the new hero gains armor (you can think about it as additional health points) equal to the value specified on its card (in the case of Figure B.3 (a) that is 5).

- **Weapons**: Heroes usually come with an attack value of 0, which means that they cannot attack. Weapons are cards that can be played into the battle zone and that give the hero an attack value. In Figure B.4 (a) we can see Fiery War Axe, a "3 mana 3/2" weapon. That is, it costs 3 mana to play this card and it gives the friendly hero 3 attack on your turn. The 2 represents the durability. Whenever the hero attacks while the weapon is in play, the durability of the weapon decreases by 1. When the durability reaches 0 the weapon is destroyed. There can only be one weapon in play per player. Playing a weapon while you have another one in play will result in destroying the old weapon regardless of its durability and replacing it with the new one.

2.2.2 Rules of the Game

A game of Hearthstone has a singular win condition: getting your opponent's hero to 0 health. A game can result in a draw if the same action will reduce the health of both heroes to 0. A draw is considered a loss for both players.

The game starts with the "mulligan" phase. Both players' decks, which have exactly 30 cards each, are shuffled. The player that goes first is decided by a coin flip. In this phase the player is presented with three cards if they are going first, or four, if they are going second. The players can now choose which cards to keep in their starting hand and which to reshuffle into their deck. For each card reshuffled a player gets to draw another one from the deck. Finally, the second player is given "The Coin", a 0-cost spell that grants 1 extra mana (more on "mana" in the following paragraph) for a turn aiming to compensate for the so-called "first player advantage". This concludes the "mulligan" phase.

The players now take turns. At the beginning of each turn a player is granted a "mana crystal" (up to a maximum of 10) and all their used mana crystals from the last turn are refreshed. Mana crystals

are used as a resource for playing cards. A player thus will have one available mana on their first turn, two on their second and so on. A player can decide to use all their mana on one card in a turn, play multiple smaller-cost cards or not play anything at all.

A player's turn-start then follows with drawing a card from their deck. After this, the player can choose to play cards and/or attack with some or all of their cards in the battle zone.

Playing a card means paying the cost of the card in order to place it into the battle zone if it is a minion, hero or weapon or in order to activate its effect if it is a spell (see subsection 2.2.1)

Attacking means choosing a friendly minion or the friendly hero and an enemy minion or the enemy hero to "fight". A "fight" between A and B means that we will subtract A's attack value from B's health and we will subtract B's attack value from A's health. If after performing both subtractions any of the two combatants' health is smaller or equal to zero, that card is destroyed (sent to the graveyard zone). Note that minions or heroes that have an attack value of 0 cannot attack. Additionally, minions cannot attack on the same turn that they were placed into the battle zone unless specified otherwise on the card.

Lastly, in order to prevent infinite games, whenever a player needs to draw a card and their deck is empty their hero will take X damage, with X starting at 1, after which X will increase by 1. This is referred to as "fatigue" damage.

2.3 Experimental Setup

In order to be able to answer the research questions a system was built to accommodate the training and testing environments and the different agents. The system contains the following components: 1) A simulation of the game playable by agents; 2) Agents that can accept a game-state and return a valid move; 3) A script that can feed the live state of the real game client into an agent. All the code used for this research can be found at https://github.com/alextutea/ReLAI-HS.

2.3.1 Playing the Game and the Simulation

Between graphics rendering and server communication time, training the agents on the real game client would have been too slow. Thus, a graphics-free local simulation had to be built. The Fireplace Hearthstone simulator (Leclanche, https://github.com/jleclanche/fireplace/) was used as a base. This is an old simulator built in python for a previous version of Hearthstone that was now updated in order to include all the cards and the latest mechanics of the game. The simulator uses the same event queuing system present in the real game in order to ensure fidelity to the source. Both in the real game and in the simulation a game-state is a list of entities where each of those entities has a dictionary of tags. The sum of all the tags fully describe an entity and the sum of all entities fully describe a game-state, no other information is needed.

For testing against real players online it was also necessary to design a way for the agents to interact with the real game. For that purpose the open source python-hearthstone and hslog python libraries were used to monitor and parse the log files of the game client into a list of entities that can be then interpreted by the agents.

2.3.2 Agents

An agent has to be able to accept a game-state object (i.e. a list of entities), calculate all the possible legal actions at the given state, and return one of said legal actions based on different criteria. The returned action is either fed into the simulation or prompted to the player of the real game. The only difference between the agents is the way they choose what action to return. In this paper we will compare agents based on Reinforcement Learning systems among themselves and against scripted agents.

There are four scripted agents:

- **Random**: Chooses a legal action at random;
- **No-Waste**: Chooses a legal action at random, but it never chooses the end-turn action if it has other actions available;
- **Facer**: First attacks the opponent's hero with all the available cards, then plays randomly until it ends its turn;
- **Trader**: Attempts to "trade" advantageously with the opponent's side of the board before attacking the opponent's hero. A "trade"

means attacking opponent minions instead of the hero. A good trade is one where a higher value opposing minion is destroyed by a lower value friendly minion. Consider the following example: My "1 mana 1/1" minion attacks an opposing "3 mana 5/1" minion; I subtract my minion's attack (1) from the opposing minion's health (1) and my opponent subtracts its minion's attack (5) from my minion's health (1). Both minions are destroyed as an outcome, but this is advantageous for me because my opponent spent more resources on its minion and the potential damage that minion could have dealt was higher than the potential damage of my minion; if on the friendly side of the board there would be a "1 mana 1/1" and a "5 mana 5/5" and on the opposing side only a "3 mana 5/1", even though I can attack the opposing minion with either of the friendly minions and the outcome would be destroying both combatants in both cases, it is always advantageous to "trade" the smaller minion into the opposing minion because then we are left with a bigger, more valuable, minion.

In the case of the Reinforcement Learning (RL) agents the choice is made by feeding the given state into an RL system and outputting the move returned by the system.

2.3.3 Representing Actions

In order to understand the structure of the system that outputs an action, we must first understand what an action is in the context of Hearthstone. Generally, an action can be either playing a card, attacking a target with a friendly minion or ending the turn. A player can make zero, one or more actions in one turn. A turn concludes if and only if the end-turn action was chosen by the current player. Every action in the game besides the end-turn action has a *source*, which is the entity that will initiate the action. That could be either a friendly minion that attacks a target or a card in hand that will be played. When playing a minion specifically, the player also needs to place it in an available spot in the battle zone. We refer to that spot as the *position* at which the minion will be played. Moreover, certain cards give the player the opportunity to choose one out of two possible effects when played, which we refer to as *suboptions*. Finally, when minions attack or when cards have certain on-play effects they require the player to choose a *target*.

In the original game code the source and target components of a move are card IDs, whereas the position and the suboption are represented by the index of the battle zone spot and the index of the suboption respectively. This is not ideal because there are a large number of combinations of source and target cards and having an output node in our RL system for each possible card combination would be problematic. What was done to simplify this was taking advantage of the fact that a player cannot have more than seven cards at one time in the battle zone and cannot have more than ten cards in hand. Knowing that we can fully describe any move with only four numbers: The index in zone of the *source*, the *position* in the battle zone where we want to place the source, the index of the *suboption* chosen and the index in zone of the *target* of our action. Any part which is not used (e.g. playing a minion that does not have any suboption effects or targeted on-play actions) will just be passed as a 0. For example, playing the leftmost minion in the current player's hand which has no effects attached to it onto the second slot in the battle zone will be encoded as [1, 2, 0, 0]. An end-turn action is encoded as an array of zeros.

There is a second type of action a player can make in Hearthstone and that is a "choice". Unlike a regular move, a choice is required only in certain situations. The most common such situation is at the beginning of the game in the "mulligan" phase (see subsection 2.2.2, second paragraph), but there are other cards that offer player cards to choose from. These are handled separately from regular moves both in the real game client and in this system and thus whenever the player is in a state in which they have to make a choice the RL system will recognize this and send a "choice" rather than a "move". A choice is treated as a single number that is the decimal conversion of a binary array of yes/no choices (e.g. the player chose the first and the third card in a four-card mulligan, thus its choice is binary represented as [1, 0, 1, 0] and 5 is our final number)

Thus, any action a can be formally represented as:

$$a = \{a_s, a_p, a_b, a_t, a_c\} \tag{2.6}$$

where a_s, a_p, a_b, a_t and a_c are integers corresponding to each part of action a.

2.3.4 RL System Architecture

Reiterating, the RL system needs to be able to accept a game state and output an action described as above. The way this system was designed was as a manager of five RL subsystems, one for each of the parts of an action (source, position, suboption, target and choice). Each part of an action has a finite amount of possible values (e.g. there are only 7 possible *position* values to choose from) and its corresponding subsystem will output an array of Q-values (one Q-value for each possible value). The manager system will then compute and iterate through all the possible valid actions at the current state and score each move based on the mean of the Q-values of the subsystems relevant for the move. The move with the highest score will be the chosen move of the whole system. Thus, $Q(s, a)$ is computed by the RL system as follows:

$$Q(s, a) = mean(S_{a_s}, P_{a_p}, B_{a_b}, T_{a_t}, C_{a_c}) \quad (2.7)$$

where S, P, B, T and C are the aforementioned arrays of Q-values yielded by the corresponding subsystems.

The difference between RL agents is given by three aspects: the learning algorithm used (Q-learning or Monte Carlo learning), the architecture of the neural network and the way each subsystem serializes the game state before feeding it to its neural network (i.e. the state representation).

There are two neural network architectures that were used in this paper which were arrived upon after several preliminary experiments:

- A fully connected Multi-Layer Perceptron with three hidden layers of 16, 32 and 16 units, the input layer is equal to the size of the serialized game state given and the output layer equals the size of the set of possible values of its corresponding part of action.

- A Convolutional Neural Network with two Convolution layers of sizes 64 and 32 with a kernel size of 2 and a stride of 1, followed by a fully connected layer of 64 units.

There are three state descriptions that are explored in this paper:

- A small-sized representation that serializes the full game-state into a 1-dimensional array that contains 5 numbers: the current health of the friendly hero, the current health of the opposing hero, the position in hand of the lowest cost minion, the position in hand of the highest cost minion and the position in the battle zone of the highest attack minion. This state description will be referred to as the **minimal** state description

- A medium-sized representation that serializes the full game state into a 1-dimensional array that contains 77 numbers: 3 numbers (attack, current health and mana cost) for each visible minion in the battle zone (both friendly and opposing) and for each card in hand, friendly hero current health, armor and attack and the opposing hero current health and armor. This state description will be referred to as the **stats-only** state description ("stats" is the term commonly used to refer to the cost, attack and health values of a card).

- A large-sized representation that serializes the full game state into a 2-dimensional array that contains the full description of every visible card arranged in such a way that it mimics the way the cards are actually arranged in the real game. The visible cards are the ones in the friendly hand zone, friendly battle zone and opposing battle zone plus the two heroes. As mentioned above, a card is fully described by the set of tags attached to it. In total there are 86 such tags. In this state representation we create a 11×8 sub-matrix (artificially adding two zeros) for each card filled with unique number correspondents of its tags. Each of these sub-matrices will be placed next to each other horizontally based on zones (i.e. all sub-matrices corresponding to cards in hand will be next to each other, etc.) and those zone-based arrays of sub-matrices will be placed on top of each other vertically in the final state representation. This process will yield a state representation topologically similar to the real game graphical representation where on the lower part of the screen one sees the cards in

their hand, then on top of that the friendly cards in play and then the opponent's cards in play, but instead of being comprised of pixels, the cards are represented by a matrix of their tags. Including the 0-filled lines that separate the different cards, this state representation is a 36×90 matrix (see Figure 2.1). This will be referred to as the **fully-visible** state representation. Because this is a large, spatial representation the CNN will be used as a function approximator for agents that use this state representation.

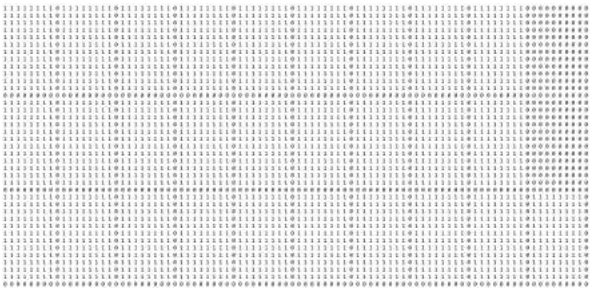

Figure 2.1: The Fully-visible state representation. Each light area is a 11×8 submatrix which contains the numerical representation of the tags which fully describe a card. Each dark area are pads of 0s. On the first row there are the cards in the opposing play zone (7) followed by the opposing hero and hero power, on the second row there are the friendly cards in play (7), hero and hero power, and on the third row there are the cards the friendly player has in hand. In total this is a 36×90 matrix.

2.3.5 Training Process

The training process is done by letting the agent play 10^4 games against itself. The Q-learning based agents learn after every action, whereas the Monte Carlo learning based agents learn after every game. Every parameter value below was chosen after conducting preliminary experiments. In the case of Q-learning the reward $R(s, a, s')$ is 1 if s' is a win state, -1 if s' is a loss state, and 0 otherwise and a discount factor $\gamma = 0.85$ was used. In the case of Monte Carlo learning all moves are trained on the final reward (1 for a win, -1 for a loss) and no discounting of the reward was used. The learning rate was set to 0.005 for all approaches and the Adam optimizer was used (Kingma and Ba, 2014). In order to achieve a more stable solution, experience replay (Lin, 1993), with $batch\_size = 32$, and target training (Mnih et al., 2015), with a soft Polyak averaging ($\tau = 0.125$) update, were incorporated.

In terms of the exploration strategy, ϵ-greedy was the method of choice for this research, with ϵ starting at 0.2 and slowly annealing towards $\epsilon_{min} = 0.01$.

Table 3.1: Win rates of trained agents vs. scripted agents using the simple deck

vs.	Random	No-Waste	Facer	Trader
Minimal - MC	81.2%	**81.0**%	49.8%	59.7%
Stats-only - MC	80.3%	79.0%	53.0%	62.0%
Fully-visible - MC	77.9%	77.4%	50.8%	57.1%
Minimal - Q	81.4%	76.1%	**53.7**%	59.3%
Stats-only - Q	**82.7**%	79.3%	50.2%	**62.1**%
Fully-visible - Q	78.0%	75.7%	52.0%	60.6%

Table 3.2: Win rates of trained agents vs. scripted agents using the complex deck

vs.	Random	No-Waste	Facer	Trader
Minimal - MC	76.8%	76.8%	59.5%	61.7%
Stats-only - MC	79.8%	77.0%	60.0%	67.1%
Fully-visible - MC	**88.3**%	**86.0**%	**67.5**%	**71.8**%
Minimal - Q	77.5%	76.1%	53.1%	65.2%
Stats-only - Q	79.9%	79.2%	59.0%	62.6%
Fully-visible - Q	86.8%	85.2%	62.7%	65.0%

3 Results

In order to better understand the performance of the agents two different decks were created to be used for the following tests. One deck is very basic, having only cards that have no special effects or synergies between cards, while the other has cards that have special effects and that requires a complex use of card synergies in order to maximize its performance. We will refer to these two decks as the *simple deck* and the *complex deck* respectively. Both deck lists can be found in Appendix C. In each test presented further the same deck will be used for both players.

In order to answer the research questions, agents that use each state representation with each of the

Table 3.3: Win rates of trained agents vs. real players using the complex deck

vs.	Rank 15	Legend
Fully-visible - MC	3W / 7L	1W / 9L

learning algorithms presented above and each of the two decks were created and trained. In total there were $3 \times 2 \times 2 = 12$ agents that were trained and tested.

During a training process each agent was trained for 500 games at a time against itself, followed by a test round of 100 games against random without exploration and without learning. There were 20 such rounds, adding up to a total of 10^4 training games. Each training process was repeated three times from scratch and the aggregated learning progress can be examined by looking at the plots in Appendix A. The line represents the mean win rate at each test round between the three training processes of an agent while the shade shows the range of the results at that specific test round.

After the training was completed, each RL agent was tested for 10^4 games against each of the four scripted agents presented in subsection 2.3.2. Each test was repeated three times and the average result was reported. The results of these tests are presented in tables 3.1 and 3.2 as win rates.

Lastly, we have tested the Fully-visible MC agent against two real players using the complex deck. One of them is rated "rank 15" (considered average within the set of players of the game) whereas the other one is rated "Legend" (considered within the top 1% of all players). 10 games were played against each opponent and the results can be seen in table 3.3.

4 Discussion

In the light of these results we can make the following observations:

- When using a simple deck, agents that use the Fully-visible state representation perform worse than the other agents. Conversely, when using a complex deck, agents that use the Fully-visible perform better than other agents.
- Generally, the performance of RL agents versus scripted agents seems to be better when using a complex deck than when using a simple deck. A possible explanation for this is given by the fact that the impact of a good move is lower with simple, no-effect cards than with synergistic cards.
- When playing with complex decks and using the Fully-visible state representation, Monte Carlo learning seems to perform better than Q learning.

Given the fact that competitive decks in Hearthstone are highly synergistic, we can conclude that a Monte Carlo learning agent using the Fully-visible state representation is the best choice for a general Hearthstone playing agent out of the options that were tested in this research.

Returning to the initially stated research questions we propose the following answers:

- **Q:** How would a trained agent that is only given a small subset of the information from a state perform compared to an agent that is given the full information? **A:** When playing with simple decks that do not require much synergistic play the former performs better, but as complexity increases the latter achieves better results.
- **Q:** How would a Q-learning agent and a Monte Carlo learning agent perform against a random agent, heuristic-based agents and against game experts? **A:** Most of the approaches presented in this paper can consistently do better than 50% against random and heuristic-based agents regardless of the deck used with the best approach peaking at 88.3% against random and more than 65% against all heuristics. Against real players, as seen in table 3.3, the agent does not perform better than 50%, however it is able to win, even against a "Legend" rank player.

It is important to mention that during the games played against real players the agent always performed sensible moves and most of the time was able to get an edge over the opponent at certain parts of the game. However, in most games the agent would make a bad move at some point in the game which a human player can exploit and easily dominate from there. A possible cause of this is

that the agent is not yet able to generalize equally well the Q values for all (state, action) pairs.

5 Conclusion

In this paper we have compared three ways of representing the state of a Hearthstone game and two Reinforcement Learning algorithms in trying to create a Hearthstone-playing agent. We found that when playing with simple decks, the smaller state representations perform better, whereas when playing with more complex decks a larger state representation will yield better results. The overall best agent was using the large state representation with Monte Carlo learning and scored a 88.3% win rate against a random agent.

Future work could consider letting the agent train for more than 10^4 games and see whether that would improve the generalization ability of the system. In previous research it was found that Q learning learns best when training against the opponent it will be tested against rather than self (van der Ree and Wiering, 2013). Thus, future work could also look at different ways of training: possibly either against the heuristic agents or against a combination of other agents and self.

References

Baier, H. and Winands, M. H. (2013). Monte-carlo tree search and minimax hybrids. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE.

da Silva, A. R. and Goes, L. W. (2017). Hearthbot: An autonomous agent based on fuzzy art adaptive neural networks for the digital collectible card game Hearthstone. *IEEE Transactions on Computational Intelligence and AI in Games*, (01):1–1.

García-Sánchez, P., Tonda, A., Squillero, G., Mora, A., and Merelo, J. J. (2016). Evolutionary deckbuilding in Hearthstone. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8.

Grad, L. (2017). Helping AI to play Hearthstone using neural networks. *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 131–134.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Lin, L.-J. (1993). Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Ross, S. (2007). In *Introduction to Probability Models (Ninth Edition)*. Academic Press, Boston.

Santos, A., Santos, P., and Melo, F. (2017). Monte carlo tree search experiments in Hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 272–279.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Swiechowski, M., Tajmajer, T., and Janusz, A. (2018). Improving Hearthstone AI by combining MCTS and supervised learning algorithms. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8.

van der Ree, M. and Wiering, M. (2013). Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. In *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford.

Wiering, M. (2010). Self-play and using an expert to learn to play backgammon with temporal difference learning. *JILSA*, 2:57–68.

Zhang, S. and Buro, M. (2017). Improving Hearthstone AI by learning high-level rollout policies and bucketing chance node events. *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 309–316.

A Appendix

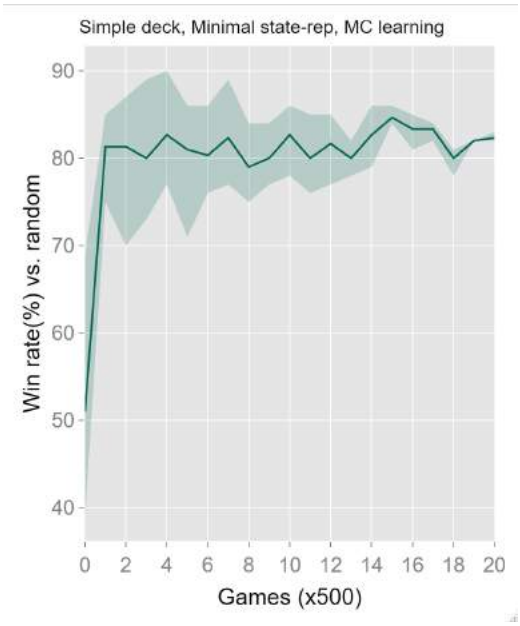

Figure A.1: Simple deck, Minimal representation, Monte Carlo learning

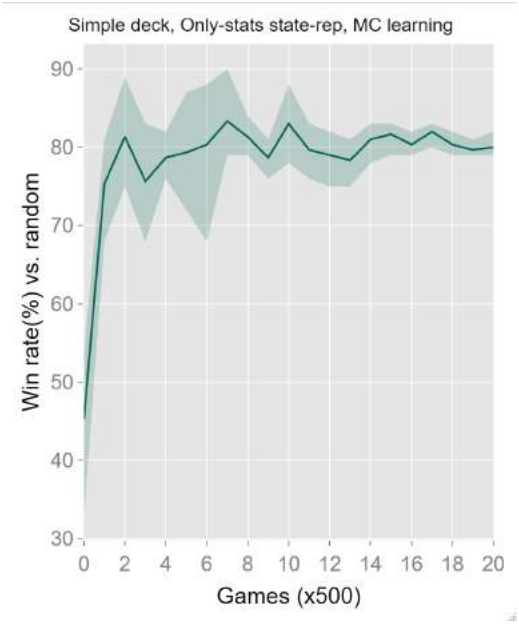

Figure A.3: Simple deck, Only-stats representation, Monte Carlo learning

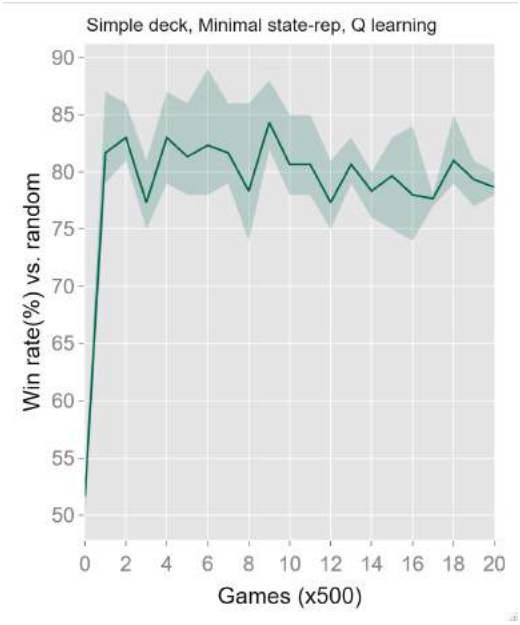

Figure A.2: Simple deck, Minimal representation, Q learning

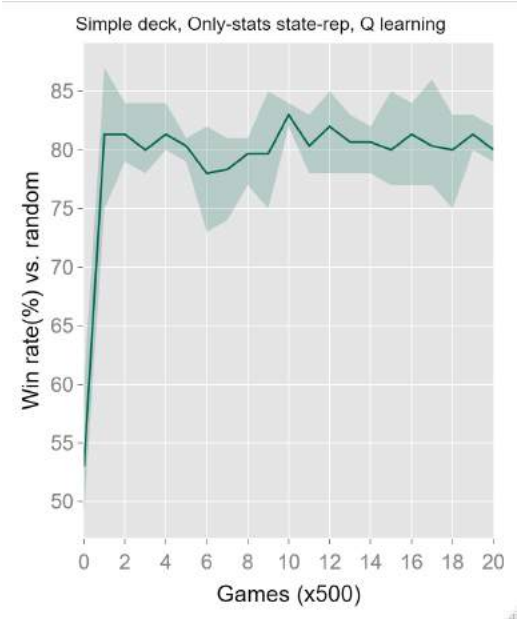

Figure A.4: Simple deck, Only-stats representation, Q learning

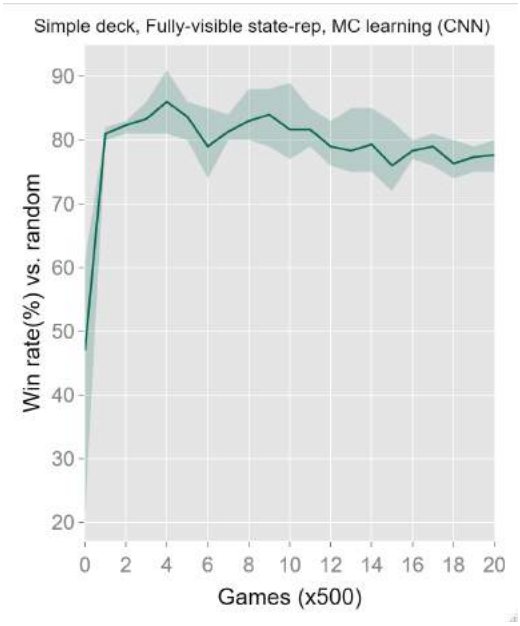

Figure A.5: Simple deck, Fully-visible representation, Monte Carlo learning

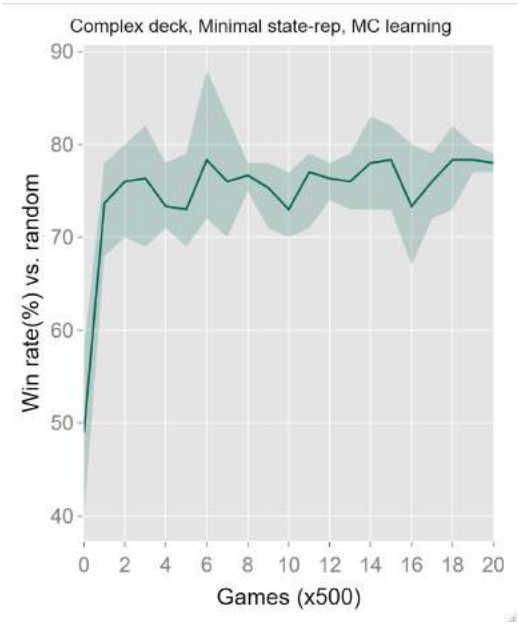

Figure A.7: Complex deck, Minimal representation, Monte Carlo learning

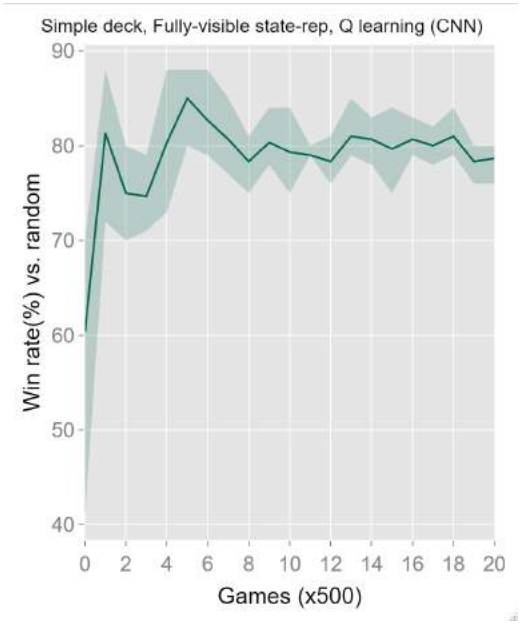

Figure A.6: Simple deck, Fully-visible representation, Q learning

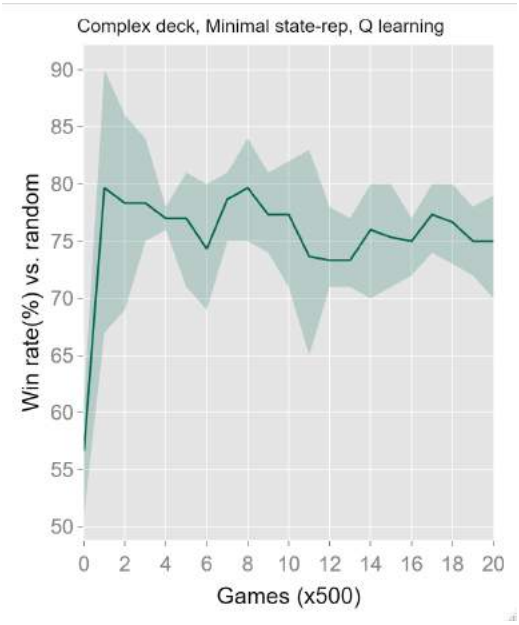

Figure A.8: Complex deck, Minimal representation, Q learning

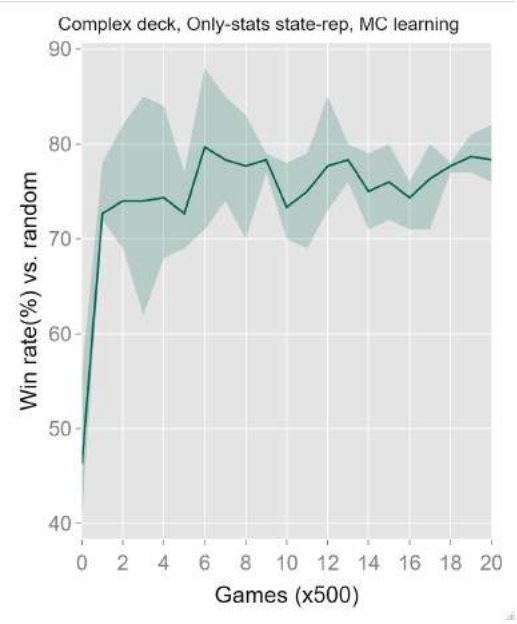

Figure A.9: Complex deck, Only-stats representation, Monte Carlo learning

Figure A.11: Complex deck, Fully-visible representation, Monte Carlo learning

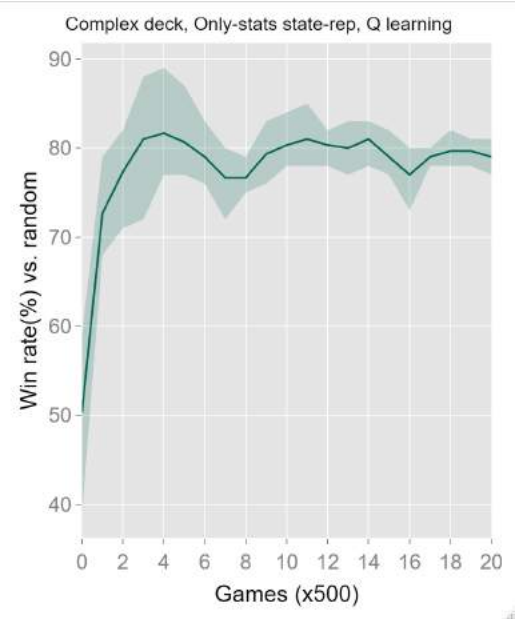

Figure A.10: Complex deck, Only-stats representation, Q learning

Figure A.12: Complex deck, Fully-visible representation, Q learning

B Appendix

(a) Magma Rager (b) Sandbinder

Figure B.1: Example minions

(a) Arcane Explosionr (b) Crystalpower

Figure B.2: Example spells

(a) Frost Lich Jaina (b) Icy Touch

Figure B.3: Example hero and hero power

(a) Fiery War Axe (b) Silver Sword

Figure B.4: Example weapons

C Appendix

Simple deck - card list:

- Hero: Rexxar (Hunter)
- 2x Acidic Swamp Ooze
- 2x Bloodfen Raptor
- 2x Boulderfist Ogre
- 2x Chillwind Yeti
- 2x Core Hound
- 2x Magma Rager
- 2x Murloc Raider
- 2x Oasis Snapjaw
- 2x River Crocolisk
- 2x War Golem
- 2x Wisp
- 2x Archmage
- 2x Ogre Magi
- 2x Dalaran Mage
- 2x Kobold Geomancer

Complex deck - card list:

- Hero: Rexxar (Hunter)
- 2x Bluegill Warrior
- 2x Coldlight Seer
- 2x Grimscale Oracle
- 2x Murloc Raider
- 2x Murloc Tidecaller
- 2x Murloc Tidehunter
- 2x Murloc Warleader
- 2x Toxfin
- 2x Ironbeak Owl
- 2x Mountain Giant
- 2x Acolyte of Pain
- 2x Defender of Argus
- 2x Abomination
- 2x Amani War Bear
- 2x Sen'jin Shieldmasta