Do Androids Dream Of Explosive Sheep?

Aleksander Dash, Nolan Handali, Franklin Jia
{adash, nolanh, frankyj3}@stanford.edu

Abstract

Hearthstone is an online trading card game that has rapidly increased in popularity since 2014 to become the most popular card game in the world, and one of the most popular mobile games as well. It is a two-player turn-based game that can be modeled as an adversarial game, and we implemented a minimax agent with a linear evaluation function, the weights of which were learned by running Monte Carlo simulations and using TD-learning. The player achieved a peak win rate within 10% of top human play against an aggressive opponent, and made action choices very similar to what a human would pick.

1 Introduction

1.1 Background and Motivation

Hearthstone is the most popular Digital Collectible Card Game (DCCG) in the world. We all enjoy playing it, and wanted to take on the challenge of developing an agent with human-level play for an arbitrary card deck. Currently, there are illicit bots in the ranked mode of the game, and they are all hard-coded to play aggressive strategies and attack the opponent directly. These bots are fairly weak, so our motivation was to create an AI that would greatly outperform these hard-coded bots. Furthermore, the most common type of deck on the competitive environment is aggressive decks, so if our AI could win against an aggressive agent, it would likely do fairly well in ranked mode. Additionally, a large part of playing well was determining what cards the opponent has, so we also aimed to create a predictor for the cards our opponent has in their hand/deck.

Hearthstone is a relatively simple game at its core. Both players have a certain "hero", and start the game with a deck of 30 cards that they select. At the start of a turn, players draw 1 card from their deck, and gain "mana" up until they have 10 mana, which is a resource used to play cards. In any given turn, players can make as many valid moves as they can. Valid moves consist of playing cards, given that you have enough mana, attacking with each minion on the board, and using your unique "hero power". Each player starts with 30 health, and the game ends when a player hits 0 health.

As a result, the game can be modeled as a Markov Decision Process, with nonzero rewards solely at the termination of the game. A game state consists of minions on the board, players' health, cards in your hand, and amount of mana you have. Any move the player takes is an action. We used a minimax model, and the inputs to this model are the various game states. Then, the model uses depth limited search to find what the best action the player can take is, and outputs that move. For deck prediction, the input is the set of cards the opponent has played so far in the game, and our k-nearest-neighbors algorithm outputs the set of 30 cards it predicts the opponent has in its deck, which we can then take a random sample of to guess the cards in their hand.

1.2 Division between classes

For CS 229, we focused more on feature extraction and backward search, as well as implementing a deck predictor. Then, we also utilized the deck predictor in our minimax agent, as we needed to have a good idea of what the opponent's moves were in order to find the optimal value and actions. For CS 221, we implemented the minimax agent and TD-learning, as well as a tool to determine which cards to "mulligan" at the start of the game, which is where the player chooses to replace some of their starting cards with random ones from the deck. However, due to the nature of the project (a minimax agent using mulligan data from batch gradient descent and playing the game with a value function that uses function approximation with weights learned from TD-learning all the while predicting the opponent's hand using K nearest neighbors so the minimax agent can be more accurate), all the different components fit tightly together.

2 Related Work

There are only a few existing projects relating to this problem. Among them is a project written by GitHub user peter1591 in which they used Monte Carlo Tree search to play hearthstone as well as using a neural net to try and predict who would win the game based on a game state[1]. The strength of their method is the fact that their simulation engine is more bare bones, as they focused less on the details of specific cards, and is also written in C++, so their simulations will take less time. Additionally, by using MCTS, they do not need an explicit evaluation function, while we learned one for the purposes of minimax. However, a weakness is that they might not capture all the features necessary for optimal play due to the simplistic model.

3 Dataset Preparation

3.1 Simulation engine

We based our framework off of the open source simulation project Fireplace by jleclance and others[2]. It implements a game engine written in Python3 that has all the rules in place for the game, and is simply missing implementations of the cards from the most recent expansion. We needed to adapt the framework in order to run multiple games in succession and keep track of results, as well as do learning. Furthermore, although a simple agent that does random actions exists, we needed to write our own agents to better mimic real play. In addition to our learning player, we also implemented an aggressive player for our learner to play against, which attacks the opponent whenever it gets a chance.

3.2 Data Generation

Ideally, we would have gotten replay data from online to form a data set to train our model on, but Blizzard Entertainment, the company that created and manages Hearthstone does not store replay data, and the third party software that does is not allowed to give us the replay data. To get around this, we decided to generate our own data by running simulations of the various agents versus each other. In general, we trained by running our agent versus the aggressive agent we wrote. This allowed us to quickly gather data by running hundreds of games in only a couple hours. For context, human play takes around 5-10 minutes per game, and some last even longer.

Additionally, for our deck predictor, we acquired the data from hsreplays.com, which is a third party site for Hearthstone that collects data, and stores the most popular and most played decks[3]. Then, for testing data, we randomly selected a deck, and 1 by 1, fed the model cards from that deck and calculated the accuracy with which the model predicted the remaining cards.

4 Methods

4.1 Feature Extraction

4.1.1 Prior Knowledge

In order to find a foundation for feature extraction, we used prior knowledge to realize some facts about Hearthstone that would prove useful for feature extraction. These were as follows:

1. The current state of the game captures all that we need in order to do well. There are very few cards in the game, most of which are never seen, that depend on what has happened in the past or what is not currently on the board. As a result, the game is captured well by the current board state.
2. The value of any given state is fairly well captured by high level components, and not individual interactions. In other words, the individual cards that we have in our hand or the specific positioning of the minions on the board is largely irrelevant, and all that we truly care about is how well we are doing in relation to our opponent, which we can capture with broad, vague features. Specifically, we extract our current health/armor, our mana we have remaining, the power we have on the board, the number of minions we have, and the same for the opponent. Furthermore, we wanted a way to capture our current advantage on the board, so we also extracted how much more of our mana we used compared to our opponent, the difference in minion attack power and number of minions, difference in the number of cards in our hand, and difference in health. Finally, we had a bias term in our feature vector. Thus, we had a total of 16 high level features that parameterized our game state.

Once we had these features, we wanted to find the optimal set of features.

4.1.2 Backward Search - finding the best features

For our minimax player to perform well, we need to provide the minimax player with a set of features that would best represent the relevant information at a given state to evaluate how good being in that state would be. To derive this set of features, we implemented backward search as follows.

We began with the feature set composed of the 16 features discussed in the previous section. While our feature set was non zero, we deleted features one at a time from our current feature set and train our player on that subset of features. Then, we determine which of the single deletion subsets results in the highest win rate over 200 games after training with the current features. After we have tried every subset created by removing 1 feature from our current set, we replace our feature set with the subset of features that resulted in the highest win rate. Then, we continue the algorithm.

Once backward search finishes, we return the highest performing set of features to use in our evaluation function during minimax exploration.

4.2 Evaluation function - Temporal Difference Learning

Now that we have the features to capture the game state, we need to figure out some way of evaluating how good this game state is. We decided to use a linear evaluation function.

In order to do this, we used Temporal Difference learning, or TD-learning. The state space is too large to use value-iteration because of the sheer number of cards, so we wanted a way to update the value estimate of a state as we simulate games, and TD-learning serves that purpose. TD-learning is an RL algorithm that uses Monte Carlo Simulations to estimate the value at a certain state. It takes the experience, which is a tuple of (state, action, reward, new state), and evaluates the prediction, as well as the target. The prediction is the value at that state(V(s;w)), while the prediction is the value at the new state multiplied by a discount factor and added to the reward ($r + \gamma$V(s'; w)). We wish these to be close, so use a simple squared loss function :

$$\frac{1}{2}(\text{prediction(w)} - \text{target})^2$$

Taking the gradient, and substituting, we have that the update rule for each (s,a,r,s') tuple is

$$w \leftarrow w - \eta\Big(V(s;w) - (r + \gamma V(s';w))\Big)\phi(s)$$

To train, we simulated games by playing against the aggressive agent, and using an ϵ-greedy strategy with $\epsilon = .75$. The reward was 0, except at the end, which was +100 for a win, and -100 for a loss. γ was set to .9 because our deck tends to fare better the earlier it can win, which is usually by turn 10 or earlier. The learning rate η was determined by trial and error, as will be explained in the Results section.

4.3 Playing the game- Minimax

To actually figure out the best action, we implemented a minimax player with depth 2. Minimax assumes your opponent is trying to minimize the value while you are trying to maximize, both of which are true for Hearthstone. Minimax works by following the recurrence:

$$V_{\max,\min}(s, \text{depth}) = \begin{cases} \text{Reward}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & \text{depth} = 0 \\ \max_{a\in\text{Actions}(s)} V_{\max,\min}(s', a, \text{depth}) & \text{currPlayer} = \text{agent} \\ \min_{a\in\text{Actions}(s)} V_{\max,\min}(s', a, \text{depth} - 1) & \text{currPlayer} = \text{opponent} \end{cases}$$

Each agent plays optimally against each other, and we select the maximum value. Rather than a single move being an action, we made it so that an "action" is a sequence of valid moves until the player ends their turn. Because these sequences can be so long, we decided to prune and only explore the 3 best/worst action chains respectively. Our eval function is what was learned from TD-learning, and the reward is as stated previously.

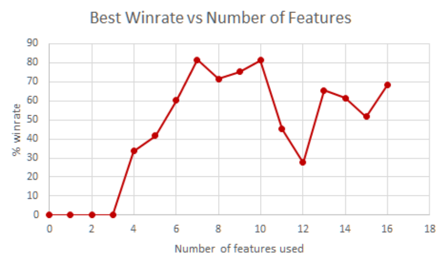
4.4 K-Nearest Neighbors Deck Prediction

In order to better evaluate which actions to take in the current turn, we need to consider the moves that the opponent could take during minimax exploration. In Hearthstone, you can't see the opposing player's hand or what cards reside his/her hand, which proves to be a problem for minimax exploration. However, we do get to see what class the opposing player chose to play, narrowing down the possible cards that opposition could play. In addition, Hearthstone decks can often be predictable since players often take decks constructed by professional players from online, also known as net decking. We used this knowledge to our advantage.

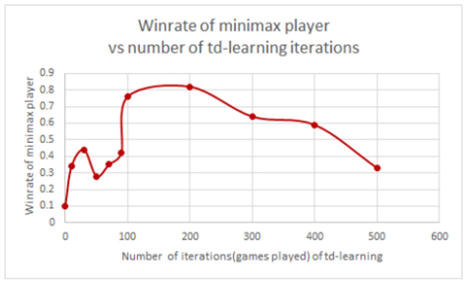
Once we've seen the opposing class, we can ignore all the decks that don't belong to that particular Hero. As the opponent plays cards, we keep track of the known cards and input that into our K-Nearest Neighbors algorithm, which outputs the decks that are most likely to be played based on the "distance" of those given cards to deck configurations. The distance of a card is inversely proportional to its frequency among all the games played in our database, which results in the algorithm being more likely to predict decks that are relatively popular - however, since the distance also decreases by a larger factor if the opponent plays rare cards, our algorithm is able to pinpoint decks that are not the most common by the unique cards they run that differentiate them from the majority of other decks. As the game progresses and more cards become known, deck prediction becomes better, as shown in the Results section.

5 Results

We tested by simulating 200 games of our minimax agent versus the aggressive agent and recording the win rates. We tested various portions of the project individually.



(a) Win rate with number of features



(b) Win rate curve vs number of TD-learning iterations

5.1 Feature Extraction

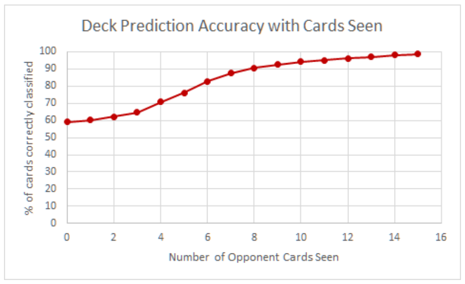
In our testing, we found that the best number of features was 7. During backwards search (figure a), we played our minimax player against the aggressive agent for 200 games using those features in our evaluation function, and found the best set of features. The weights for the evaluation function were found using 200 iterations of TD-learning. The optimal features it found were features that didn't capture things about the individual state of the board, but rather compared how well we were doing to our opponent. These features are the main things competitive players consider when playing, so our model managed to find features that are applicable in real life. Furthermore, we believe that higher dimensional feature vectors overfit during training, as it focused too much on the specific details of the board. After the peak, the win rate rapidly decreased as we tried to prune more features because the model did not capture enough of the board to make accurate decisions.

5.2 TD-Learning

[4] In figure b, we see that with the number of iterations TD-learning trained for, the win rate gradually increased, until rapidly plummeting. We trained with an ϵ-greedy policy, and we choose ϵ to be .75 because we found that when we did a purely random policy, we lost too fast and never explored enough of the game state. We also set the learning rate to .001, because when we tried with larger learning rates, the model overshot changes to its weights in the beginning of training, and the weights approached infinity. We could have fixed this with regularization, but decided adjusting the learning rate was simpler. The model peaked at a 81.5% win rate, which is close to optimal. A top human player playing the deck that our model used could feasibly achieve over a 90% win rate against the aggressive agent. We found that the weights the model learned match human thought and strategy accurately. It learned that having more minions and more minion power than the opponent was good, and although your own health is important, it was more important to kill the opponent. Also, the deck that we gave the model to play performs better by focusing on controlling the board and ignoring your own health total, and the model gave higher weights to having more minions, which is how the deck is supposed to play.

When training, the model lost a vast majority of the time due to its random policy. We believe that as the model kept losing, all the weights turned negative, as everything it did led to a losing scenario, so it failed to learn anything about what led to a winning scenario. To fix this, we could try and train it against a different agent that wouldn't be as aggressive or win as often, or we could play the model against itself during training. Alternatively, we could try and use a smaller ϵ so that we would choose the current estimate for the optimal option more often.

5.3 Deck Prediction



(a) Percent of cards predicted accurately vs number of cards seen

The deck predictor, as shown, achieved over a 90% accuracy rate after only seeing a third of the deck. Due to the fact that there are people who don't "net-deck", and add rarely seen cards to their decks means that the predictor cannot achieve 100% accuracy, until seeing all the cards. In comparison to a human oracle, the predictor does worse than a human when little cards are seen, as the human will make more generalizing assumptions about what the opponent is playing based on what is popular in the current meta-game at the moment, but after about a third of the deck has been seen, the predictor does about as well as humans would do, and when about half the cards in the deck have been seen, the predictor accurately predicts 29 out of the 30 cards in the opponent's deck.

6 Conclusion

Our deck predictor worked quite well at almost human level performance. Additionally, our minimax agent achieved high win rates against the aggressive agent using the TD-learned weights for our evaluation function, and the best set of features we found using backward search. The actions found are very close to what a human player would make in the same situation, and the win-rate it gets is very close to top human play.

For future work, we would like to implement a neural net for TD-learning, rather than using a linear approximation function. Furthermore, we could implement MCTS, to try and explore more branches, as currently, due to the limitations of the simulation engine, we can only explore depth 2, and have to prune all but the most promising 3 branches, as otherwise, the computation time takes too long. Additionally, given more time, we could start with more features, such as having each card be its own feature, so that we do consider the individual interactions between cards and synergy that makes the cards stronger than the sum of its parts. We ignored them for the purposes of simplification of this project and the fact that they are not of the utmost importance, but they may be able to improve our win rate slightly, and they would allow us to learn how to play different decks that require cards to be played in a specific, exact order in order to win better.

For deck prediction, we would like to implement a neural net with softmax output in order to determine which deck the opponent is using. Furthermore, the next step would be to try and predict not just the cards in the deck, but the cards in the opponent's hand. Professional players will make educated guesses as to what is in the opponent's hand, and play accordingly, but our model simply assumes a random sample of their deck, which is not true. Thus, if we could predict what was in their hand, we could use that in our minimax agent to achieve an even higher win rate.

Finally, if it were not against the terms of service, we would like to pit our bot against real players in ranked mode, and see how well it would perform.

References

[1] peter1591. *hearthstone-ai*. https://github.com/peter1591/hearthstone-ai. 2017

[2] jleclanche. *fireplace*. https://github.com/jleclanche/fireplace 2017

[3] hsreplays.com. https://hsreplay.net/analytics/query/list_decks_by_win_rate/?GameType=RANKEDSTANDARD &RankRange=LEGEND_THROUGH_TWENTY

[4] Percy Liang, Stefano Ermon "Games 2" (class lecture, CS 221, Stanford, CA, Oct 25, 2017).

Contributions

Aleksander implemented the majority of the logic for TD-learning as well as the minimax agent implementation and K-nearest neighbors deck prediction, handling most of the coding responsibility for those parts. He also cleaned up the code-base at the end of the project to get it ready for submission and review.

Nolan wrote code to adjust the simulation engine to work for our purposes. He also worked on implementing the cards that were missing and he worked on feature extraction and trying to extract the best set of features.

Franklin also worked with the framework, and trying to get the engine to be able to work for our purposes. He did most of the testing and wrote code to assist with that task, such as implementing the aggressive agent.

The final report and project were done collaboratively. Nolan and Franklin did most of the writing, and Aleksander did proofreading and final edits.

Thanks for a great quarter! We learned a lot and had a lot of fun!