Will our new robot overlords play Hearthstone with us?

CS 229 Final Report

Hubert Teo hteo@stanford.edu Yuetong Wang yuetong@stanford.edu
Jiren Zhu jirenz@stanford.edu

June 6, 2016

1 Hearthstone

Hearthstone is a online Trading Card Game released in 2014 that is gaining in popularity. The basic driving principles of Hearthstone are relatively easy to describe and understand. It is a turn-based one versus one game. Both players start by choosing a hero and a deck to draw from. In the beginning of each turn, a player gains certain amounts of resources including mana and cards. In his turn, a player can make as many moves as desired (as long as there are enough resources or the rules otherwise allow it). Valid moves include playing cards and using the player's hero's special abilities. The ultimate goal is to win the game by removing all hit points of an opponent.

It is natural to model this game as a Markov Decision Process, with rewards only at terminal states. Every game situation is a state and the move that a player takes is an action. Like any other turn-based trading game, it has a sprawling collection of mechanics that apply only in certain circumstances. Thus writing a good AI for Hearthstone remains a difficult and interesting problem because of two main challenges.

1. With an MDP, the biggest impediment to a learning is the size of the state space. For Hearthstone, because there are various special effects and a lot of game mechanics, the number of possible game states is huge. A naive solution based on traversing the MDP would fail miserably because the chance of seeing the same state twice is essentially zero. The AI has to be able to evaluate previously-unseen states in a reasonable fashion.

2. The other limitation is that actions in Hearthstone are highly coupled with states. The rules only allow certain actions to be applied, depending on the game state. This means that a naive approach that enumerates the fixed set of all possible unique actions in the MDP is intractable: the set of all possible actions is far greater than the number of actions applicable to a particular state. Even so, there are approximately 20 unique actions available at each game state. The combinations of them result in a big action space that is hard to enumerate.

These challenges mean that a good AI has to be tailored strategically to the specific restrictions and characteristics of Hearthstone – a custom-built AI is needed to play this game with any measure of success. We aim to use learning techniques to match or outperform current AIs for Hearthstone, most of which are manually-coded heuristic agents.

2 Methodology

2.1 Platform

We based our project off the hearthstone simulator Hearthbreaker, an open source platform developed by danielyule and other contributors [3]. The well-functioning Python 3 game engine is feature-complete up to last year's Hearthstone rule-set. It also comes with an Agent framework that is easily extensible to implement our AI. Hearthbreaker also comes with several baseline AIs of its own. There is a "Random Agent" which performs a random valid action at every step and a "Trade Agent" which looks for the move that trades best with a comprehensive set of hand-coded heuristics. We use the "Trade Agent" as the heuristic agent against which we evaluate our AI.

2.2 Feature Extraction

Finding an effective representation of the huge game state space is critical to formulating a tractable learning problem. We make several observations specific to Hearthstone based on prior knowledge:

Observation 1. *The current game state provides enough information to determine how well both players are doing.*

This means that the game is well-parametrized by the current game state: the game history yields little benefit save for allowing card-counting techniques that keep track of the set of possible remaining cards in the deck.

Observation 2. *The intrinsic "value" of many game components can be roughly understood in terms of independent contributions to the "goodness" of a game state.*

While there can be many combinations of minions, minion specific effects, hero specific mechanics, cards and decks, we argue that we can treat each of these components as having independent contributions to a universal "value" of a game state, because the specific mechanics have less importance in relation to the overarching strategy. This ignores special interactions between game components, but is still in line with our goal to learn a good general strategy for Hearthstone.

Guided by these two observations, we engineered a feature extractor that captures the critical resources of both players. The second observation means that it should be possible to find the game value as a linear combination of the feature vector components, but does not preclude using more sophisticated non-linear function approximators.

Specifically, from each minion, we extract health, attack damage, and an indicator variable denoting if it can attack, sorting the triples to canonicalize our representation. We also include the hero's health and armor value and some other miscellaneous features. Considering that there are at most 7 minions in the playing field for one player, this gives 30 features for one player. Repeating this process on the opponent, we obtain a total of 60 features.

2.3 Monte Carlo Methods for Action Evaluation

Armed with a game state representation, we are now faced with the problem of evaluating the quality of an action. Unlike in deterministic games, we cannot simply enumerate the states that result from taking an action, because actions can have non-deterministic effects. To further complicate matters, each player is allowed to perform any number of legal actions in any order within a turn.

Observation 3. *The set of actions that have non-deterministic effects is small. The majority of the non-determinism results from drawing cards and from the lack of knowledge of the opponent's deck and hand.*

Observation 4. *The number of actions within a turn is always finite; there are no cycles in the action graph.*

(4) implies that the action graph is a directed acyclic graph (DAG), with complete turns mapping to paths in the action DAG. (3) suggests that Monte-Carlo techniques will be effective. Hence, we arrive at an algorithm for evaluating the quality of an action. First, simulate taking the action on a copy of the current game state. Next, perform a Monte-Carlo traversal of the action DAG, simulating each action from each game state, and return the maximum value of the visited states. This yields a Monte-Carlo estimate of the maximum value at the end of a complete turn after taking the action, which we interpret as the value of the action.

2.4 Action DAG Pruning

In practice, the action DAG can be too large to be efficiently traversed completely because in the late game, each player has more resources and thus more actions to take. With the large branching factor, this results in an exponentially-growing number of states to visit. We took several measures to prune our traversal and optimize performance.

1. Some actions are order-independent, so there might be multiple paths to the same state. Since we evaluate the quality of an action by the value of the final state, it is not necessary to visit each state more than once. Hence, we hash visited game states, and avoid them in our traversal, produces a spanning tree of the visited states.

2. We perform a depth-limited search of the DAG. At depths exceeding a certain maximum depth, we cease to explore every action available at the node, and instead switch to a greedy traversal strategy, continuing only into best possible action.

3. We use an A* search strategy, traversing active game states in an order that prioritizes shallower nodes and nodes with better value. This ordered traversal enables us to set a hard budget on the maximum number of states to be traversed.

This gives us a performance-tunable search strategy, not unlike general MCTS [2]. For both training and evaluation we set the maximum depth to 2 and the traversal budget to 25.

2.5 AI

Assuming that we have a model that evaluates the value of a state given the feature vector representation, combining these components produces a complete AI agent. At each state, we enumerate all legal actions and select the one with the best value according to the evaluation method above. Note that since actions have non-deterministic results, we perform a fresh Monte-Carlo traversal at each step until the best action is to end the turn.

3 Models

We now proceed to the most important part of a good AI: the way it evaluates the current game state. We developed two approaches to this task: supervised learning and reinforcement learning. Both models share the same 60-dimensional feature extractor.

3.1 Supervised Learning

3.1.1 Dataset Preparation

We pit the existing heuristic AI "Trade Agent" against itself, recording the game state at the end of each turn. Each game history is then processed, and a score assigned to each game state. The base score f is $f_{\text{win}} = 10$ if the player won, $f_{\text{lose}} = -8$ if the lost and $f_{\text{tie}} = 3$ for a draw (when both players perish simultaneously). The score for a game state d turns away from the end of the game and whose current player's base score is f is assigned as

$$y = 0.8^d f$$

This mimics the Q-learning discount scheme with a discount of $\gamma = 0.8$. Feature vectors are then extracted from each game states and the resulting x, y pairs recorded into the dataset.

3.1.2 Regression Model

We used two models for supervised regression: a linear regression model and a deep neural network model implemented within the `scikit-learn` [5] and `scikit-neuralnetwork` [1] frameworks. The deep neural network contains a ReLU-activation layer with 100 neurons, a sigmoid-activation layer with 200 neurons, a tanh-activation layer with 100 neurons, and a final linear combination layer for output.

3.2 Reinforcement Learning

We used the Q-learning algorithm to learn the best linear approximator to the Q function, making several non-standard modelling choices along the way.

3.2.1 Action-path MDP

It is difficult to run Q-learning directly on the Hearthstone MDP as is, since each player takes a variable number of actions in each turn. Hence, the game progression alternates between players irregularly, slowing convergence because the discounted future reward would vary depending on not just the number of turns but the number of actions taken by each player.

Hence, we reformulated the MDP to interpret an entire path in the action DAG (corresponding to a sequence of legal actions) as an action, with the restriction that the last action must be to end the current turn. Note that unlike with the action evaluation scheme, there is no issue with non-determinism here: we can simply continue the game from the last game state on the action-path once we have selected an action-path.

3.2.2 Function Approximation

We explored two approaches to approximating the Q-function with a linear function. Given the current state s and the action sequence a (which we also treat as a state, the resulting game state according to our simulation), we can choose to either predict the Q-value from only a, or from both s and a. More specifically, the two function approximators tested were:

$$Q_1(s,a) = \theta_1^T \phi_1(s,a) = \theta_1^T \phi(a), \qquad Q_2(s,a) = \theta_2^T \phi_2(s,a) = \theta_2^T \begin{bmatrix} \phi(s) \\ \phi(a) \end{bmatrix}$$

The first model (final state only) corresponds to the assertion of Observation 1. The second model (state pair) is a generalization of the first, but might be harder to train due to the increased number of weights.

3.2.3 Q-learning

With these two function approximators, we can simulate games using the game engine by running Q-learning with an ϵ-greedy ($\epsilon = 2$) strategy for both players using the same model. To efficiently select random action sequences, we used random walks on the action DAG. The best action sequence and its value were found using the same search routines used

for action evaluation. The Q-learning update is as follows: having observed that taking action a from state s leads to state s' with reward $R(s')$, set

$$w^{(t+1)} := w^{(t)} + \eta \left[R(s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \phi(s, a)$$

where the discount factor $\gamma = 0.8$ and the rewards $R(s')$ are the same as in the supervised learning model: $f_{\text{win}}, f_{\text{lose}}, f_{\text{tie}}$ for winning, losing, and tied states respectively.

3.2.4 Experience Replay

Observation 5. *The only non-zero rewards occur at the last turn of the MDP.*

Unfortunately, when using plain Q-learning, this means that no information is learned from all the non-terminal moves in the first game, since the rewards are all zero. This causes convergence to be haphazard as earlier games have an outsize influence on the ϵ-greedy exploration strategy.

To speed convergence and reduce variance with respect to the observed state transitions, we implemented an experience replay training scheme, inspired by Mnih et. al.'s paper [4] Each training epoch consists of a simulation phase, an instant replay phase, and an experience replay phase. First, an entire game is simulated to termination without performing any Q-learning updates. The state transitions in the game history are then replayed in reverse and used to update the model. Next, the new game is added to the experience bank of state transitions, which is truncated by removing random samples if it exceeds its fixed maximum capacity (100). 50 experience replay samples are then drawn from the bank and used to update the model. This accelerates convergence and helps to smooth the exploration strategy, since the model remains constant in the simulation.

4 Model Training and Evaluation

For the supervised models, we simulated 2000 games, generating a dataset with about 40000 data points, each corresponding to an intermediate game state (each game takes about 10 turns per player, and thus gives around 20 game states). We then randomly sampled 10, 20, 50, 200, 500, 1000, 2000, 4000, 12000, 20000 and 40000 data points independently from the dataset into sub-datasets and trained both supervised models independently on each sub-dataset.

The reinforcement learning models were both trained with experience-replay Q-learning for 1, 5, 10, 20, and 50 epochs, with each training run performed independently.

All four models, each with various training set sizes are tested against the existing heuristic AI using the Monte Carlo action evaluation strategy, with a maximum depth of 2 and a traversal budget of 25. We record our AI's winning rate after simulating 100 games against the benchmark heuristic AI.

5 Results

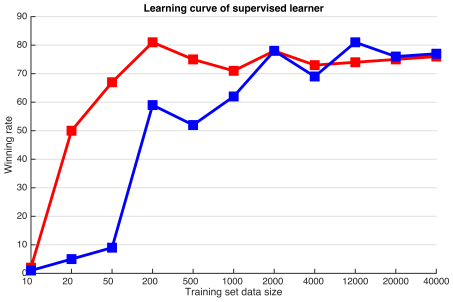

Figure 1: Learning curve for supervised linear (blue) and deep neural net (red) models.

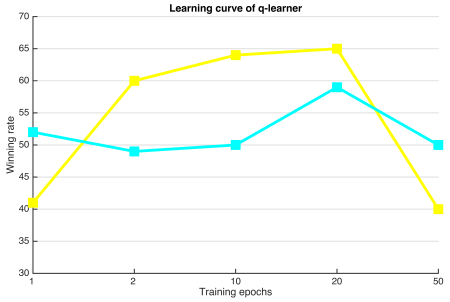

Figure 2: Learning curve for final state (yellow) and state pair (cyan) reinforcement learning models.

Currently, our AI agents readily beat the existing baseline heuristic agent provided in the engine. We report the learning curve for all four models with increasing training set size for supervised learners (figure 1) and training epochs for reinforcement learners (figure 2). Note that the training times are measured in different units for our supervised and reinforcement models.

The supervised learning models perform well against the heuristic agent, both achieving a maximum win rate of 81%. The deep neural net appeared to converge significantly faster, with both models tapering out at approximately 75%.

The reinforcement learning models still matched the heuristic agent, but with a less impressive margin. The state pair model hovered at around a 50% win rate, and the final state model initially outperformed it to achieve a 65% win rate. However, both models deteriorated after too many training epochs.

Training set size	10	20	50	200	500	1000	2000	4000	12000	20000	40000
Supervised (linear) (%)	1	5	9	59	52	62	78	69	**81**	76	77
Supervised (deep neural net) (%)	2	50	67	**81**	75	71	78	73	74	75	76

Training epochs	1	2	10	20	50
Reinforcement (final state) (%)	41	60	64	**65**	40
Reinforcement (state pair) (%)	52	49	50	**59**	50

Figure 3: Winning rates against heuristic AI

5.1 Supervised Model

It is somewhat surprising that our supervised models perform so well against the heuristic agent, especially given that they are trained with data extracted from its behavior. Examining the learned weights for the best linear model reveals reasonable values: intuitively bad things such as the opponent's hero health and total attack damage of opponent minions have negative weights, and good things such as our player's hero health and healing multiplier have positive weights. Thus, our AI agent appears to have learned a general strategy for playing Hearthstone.

This hints at a possible explanation for why it fails to exceed a 90% winning rate: apart from the inherent randomness in the Hearthstone game, it is also not taking advantage of compounding effects from multiple spells, minions and special hero powers. Since the deep neural model appearing to be very adept at learning to make use of the information we provide, this suggests that it might be possible to extend our game state representation with more specific identifying information about the individual minion and hero types.

5.2 Reinforcement learning

Though the reinforcement learning models do not perform as well as the supervised models, we must note that they received less training: 50 training epochs is approximately equivalent to a training set size of 1000. In addition, the lack of supervision in Q-learning means that the models were responsible for their own exploration of the strategy space and were not able to leverage the knowledge of the heuristic agent.

When training, there was significant instability in the model weights learned especially during the initial few games. This is problematic because the Q-learning strategy is dependent on these initial weights, so any initial biases might result in a feedback effect. Bad initial experiences lead to worse exploration strategies and poor learning. The experience replay mechanism also has the unintended effect of giving outlier games more influence on the model. Further analysis is required to determine if this was the cause of the fall in the winning rate at 50 training epochs. Nonetheless, a possible remedy is to implement bootstrapping, training the model on a small number of games between the heuristic agent and itself before switching back to the Q-learning training procedure.

6 Conclusion

The ability of our machine learning AI agent to outperform the hand-coded heuristic agent in Hearthbreaker is certainly encouraging. Its success proves that it was able to learn the essence of the game and excel at it, even without leveraging specific interactions between game components. The challenging next step will be to improve it to the extent that it is able strategize with respect to all the special interactions between game components and match or surpass expert human opponents.

References

[1] aigamedev. *scikit-neuralnetwork*. `https://github.com/aigamedev/scikit-neuralnetwork`. 2015.

[2] C. B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (Mar. 2012), pp. 1–43. ISSN: 1943-068X. DOI: `10.1109/TCIAIG.2012.2186810`.

[3] danielyule. *hearthbreaker*. `https://github.com/danielyule/hearthbreaker`. 2015.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). URL: `http://arxiv.org/abs/1312.5602`.

[5] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.