Heart of AI

Sam Keyser, Zach Wentz, Charles Nortrup, Franklin Cole, Mitchell Johnstone*

Department of Electrical Engineering and Computer Science

Milwaukee School of Engineering

1025 N Broadway St, Milwaukee, WI 53202

Abstract

This research project delves into the implementation of the card game Hearts using reinforcement learning techniques, facilitated by the Python library RLCard. Extending RLCard's capabilities, Hearts was integrated into its suite of card games to investigate the effectiveness of various reinforcement learning algorithms in understanding and strategically playing the game. Using algorithms such as Deep Q-Networks (DQN) and Neural Fictitious Self Play (NFSP), diverse state representations are analyzed to capture crucial game dynamics, including hand composition, trick history, and opponents' actions. This study aims to discern the strengths and weaknesses of these approaches in tackling the complexities of Hearts, thereby providing valuable insights into the advancement of AI systems for game playing.

* Team Lead

Introduction

This paper explores the utilization of the Python library RLCard, an agent for reinforcement learning research and applications in card games, to develop an AI agent that proficiently plays the card game Hearts. [4] Cards are fundamental elements of countless games that embody a vast range of strategies, probabilities, and psychological nuances. From the strategic maneuvers in Poker to the tactical plays in Bridge, card games are ever-growing grounds for exploring computational intelligence. Among these, the game of Hearts, which is a game with roots in the 19th century, requires players to balance risk and reward through each hand.

Hearts is a four-player, trick-taking game. In each round, each player is dealt thirteen cards. The player with the two of clubs plays that card to kick off the initial round. Then, each remaining player plays a card of the same suit (clubs). If a player does not have a card of the same suit, they may place down any other card in their hand. The player who played the highest card of the original suit (clubs) takes every card played in that initial trick (stack of four cards) and sets the trick cards aside. The player who takes the trick is responsible for starting the next trick. After the first round, the starting player may choose to play any starting card, with the exception that if a penalty card (including the queen of spades and all heart-suited cards) has not been played in the round yet, then a penalty card may not lead the trick. Players will again try to match the initial suit played in the new trick, and the player that played the highest card that matches the suit of the initial card in the trick takes the cards. This is repeated until thirteen tricks have been played and collected (i.e., all 52 cards are played).

At the end of a round, player penalty points are calculated based on the tricks collected by each player. Normally, each heart-suited card collected is worth one penalty point, and the queen of spades is worth thirteen penalty points. In the case where one player collects all fourteen penalty cards (known as “shooting the moon”), that player gets zero penalty points, and the three other players each get twenty-six penalty points. After calculating player penalty points, a new round begins. Traditionally, rounds are played and restarted until a player has one-hundred or more total penalty points accrued over many rounds. At this point, the winner is the player with the lowest number of penalty points overall.

To create an AI that plays hearts, we used reinforcement learning. Reinforcement learning is a branch of machine learning which focuses on training AI agents to make sequential decisions in dynamic environments to maximize their rewards earned. [3] These algorithms learn through interaction with an environment by taking actions, observing the state, and adjusting their next moves based on the reward given. Reinforcement learning relies on a feedback loop where the agent navigates an environment, taking actions to achieve desired goals while receiving rewards or penalties based on its decisions. Through this trial-and-error process, the agent learns optimal moves and strategies that dictate actions for different states, aiming to maximize long-term rewards. Key components of reinforcement learning

include the agent, which is a decision maker, the environment in which the agent plays in, the actions of the agent, the state of the environment, and the rewards.

Related Works

Card games are particularly amenable to representation in computer programs due to their well-defined rules and explicit action spaces. Most difficult card games have players act on “imperfect information,” as players do not have complete knowledge of all the cards in play, including those held by other players. This adds a strategic element as players must make decisions based on incomplete information, often involving bluffing or risk assessment. This characteristic in card games makes them ideal for exploring artificial intelligence (AI) applications in strategic decision-making. [5]

Previous efforts have been made to create AI players for a variety of imperfect information card games, including Texas Hold'em [1] and Blackjack [2] showcasing the potential for AI to excel in complex, strategic environments. Python packages like RLCard offer a platform for implementing and experimenting with AI agents in popular card games, providing researchers and developers with standardized environments and interfaces for reinforcement learning (RL) research.

Methods

RLCard is a Python toolkit developed by the Tsinghua University Data Sciences and Machine Learning Lab to facilitate research and experimentation in reinforcement learning (RL) using card games. It addresses the need for standardized environments and interfaces for RL agents to learn and play card games effectively. By providing a variety of card games and customizable environments, RLCard enables researchers to focus on developing and evaluating RL algorithms rather than implementing game mechanics from scratch.

The Game of Hearts

The design for the game of Hearts in this research project was compressed from the traditional version of the card game. As touched on earlier, in a traditional game of Hearts, a round consists of 13 tricks (all 52 cards being played), and a game consists of multiple rounds (as many rounds that need to be played until at least one player crosses the threshold of 100 penalty points obtained). In the implementation for this research project, a round is only a single trick (4 cards played, with one card being from each player), and a game is 13 rounds. The game simply ends once all 52 cards have been played one time, which makes the agent learn to optimize a single round in traditional hearts. Other card games that were already implemented in RLCard, such as Bridge, were compressed in a similar fashion, so it was decided that the same should be done with the Hearts implementation.

State Engineering

As for the information provided to an agent during training, the state consists of the following five components:

- The cards in an agent's hand.
- The cards in the current trick.
- All the cards played so far in a game by all players.
- The number of points an agent currently has.
- The total number of points seen in the game overall.

It was reasoned that these components are the main pieces of information that could reasonably help a human determine which card would be best to place in the trick to reduce points acquired, so the same components were provided to the reinforcement learning agents with the hope that they would help the agents learn how to effectively play and win a game of Hearts. Also, it was figured that any other crucial pieces of game information could be derived from the five components provided. For example, for an agent to determine whether it should try to "shoot the moon," it would be important for the agent to know if another player has already collected a penalty card. There is no direct state component that provides this information, but this information can be derived from the number of points an agent currently has and the total number of points seen in the game overall.

Reward Engineering

To train a reinforcement learning agent, reward points were given to an agent after the completion of a game of Hearts. An agent's reward points were derived from the penalty points obtained by the agent in one Hearts game. There are two different methods of giving points to agents in the game of Hearts. The first (and more likely) method of awarding points occurs when no player has "shot the moon." In this case, an agent's points are the negative of the card points accrued. For example, if an agent accrues five heart-suited cards during a game (five penalty points), then the agent will be assigned -5 points. If an agent picks up no penalty points during a game, then it will receive positive one points, to encourage the agent to pick up fewer penalty points in a game.

The exception to this method of awarding reward points occurs when one of the four players in the Hearts game "shoots the moon," meaning that said player has collected all the penalty cards. As mentioned earlier, zero penalty points are given to the player that "shot the moon," and twenty-six penalty points are awarded to each of the other players, punishing these other players for allowing the one player to pull off such a feat. Therefore, for our implementation of assigning reward points, the player that has "shot the moon" is awarded 1000 points as a highly positive reward. The three other players are each assigned -50 points as a severe punishment.

It was reasoned that by giving an agent a negative reward for taking on penalty cards, an agent would learn to minimize the number of penalty cards acquired. Additionally, by adding the different reward point awarding strategy that occurs when someone has “shot the moon,” it was figured that an agent would also be able to learn to “shoot the moon” themselves and learn to not allow for another player to “shoot the moon.”

Algorithms

Q-learning is a computational method of determining the expected reward for an action in each state. However, with large state spaces, it’s exponentially more difficult to learn all potential responses. To approximate responses without mapping the entire space, Q-learning and deep neural networks were combined to create DQN, or Deep Q-Network, which completely changed reinforcement learning. It is appropriate for a range of tasks and performs best in settings with big state spaces and distinct action spaces. Its capacity to manage intricate decision-making scenarios, which enables it to pick up efficient policies in demanding settings, is one of its key advantages. On the other hand, DQN may have trouble with continuous action spaces, and precise hyperparameter tuning may be necessary to get the best results. Furthermore, especially in training environments with non-stationary dynamics, it might experience instability.

Neural Fictitious Self-Play, or NFSP, combines game theory and deep learning methods to teach players how to play games with imperfect information, like poker. It uses fake self-play to create training data and a deep neural network for strategy learning. The main advantage of NFSP is its ability to converge to a Nash equilibrium even in intricate multi-agent settings. Although NFSP works well, it requires a lot of processing power and can have stability problems when it's being trained. It is still, however, a potent instrument for dealing with strategic decision-making in competitive environments.

Results

The reinforcement agents were trained under a variety of conditions, but most of the research focus was put towards changing the duration of the training and the state representation. Using the RLCard training scheme, reinforcement agents were trained on two NVIDIA® T4 GPUs in the Milwaukee School of Engineering’s supercomputer, ROSIE. Each episode involved playing and learning from a single game of Hearts. Both the DQN and NFSP algorithms were used, and each agent was trained in 4 player games against random agents (agents which play a random move among the set available to them). The trained agent was evaluated every 100 episodes to find the average reward across 100 simulated games.

For the time experiment, after training both agents for about 1 million episodes, the resulting rewards were smoothed by taking a rolling window average over 100 of the evaluation episodes. This provides a better view of the general trend over time without high

frequency noise. The plot of the smoothed rewards is provided in figure 1. From the figure, it seems that the DQN agent outperforms the NFSP agent by an increasingly larger margin over time. We speculate as to why in the following Discussion Section.

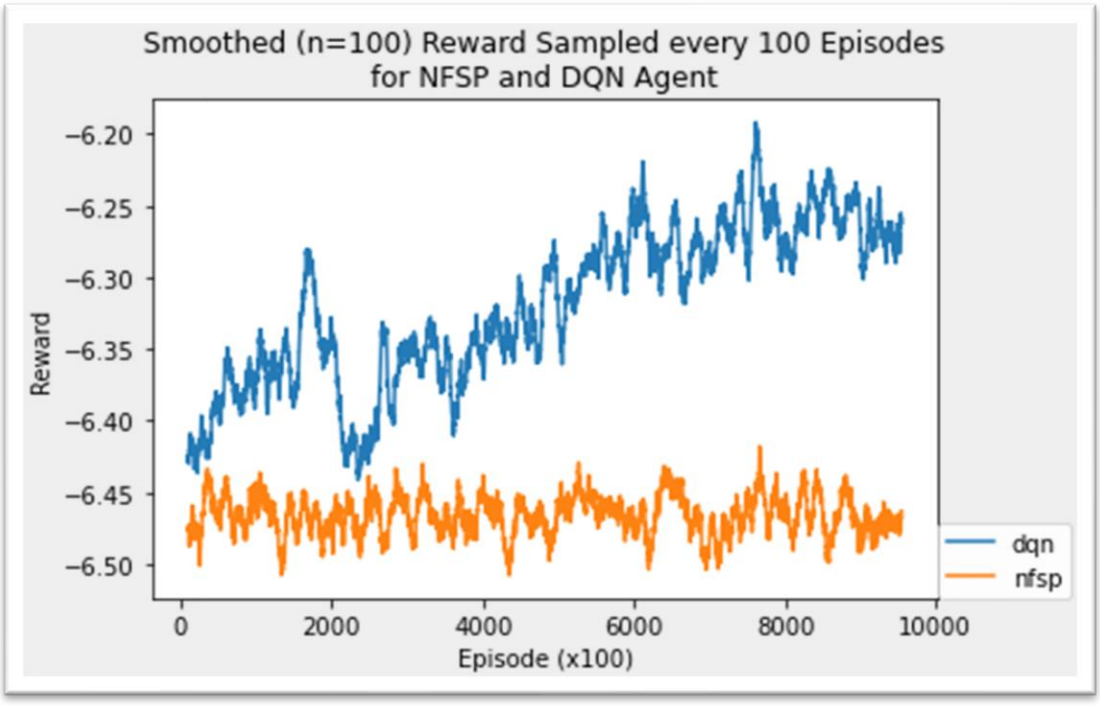

Figure 1: NFSP and DQN agent training results

The other experiment performed was the behavior of the agent when the state space was modified. Because the DQN agent showed a more drastic increase in learning from the first experiment, a DQN agent was used in the state experiments.

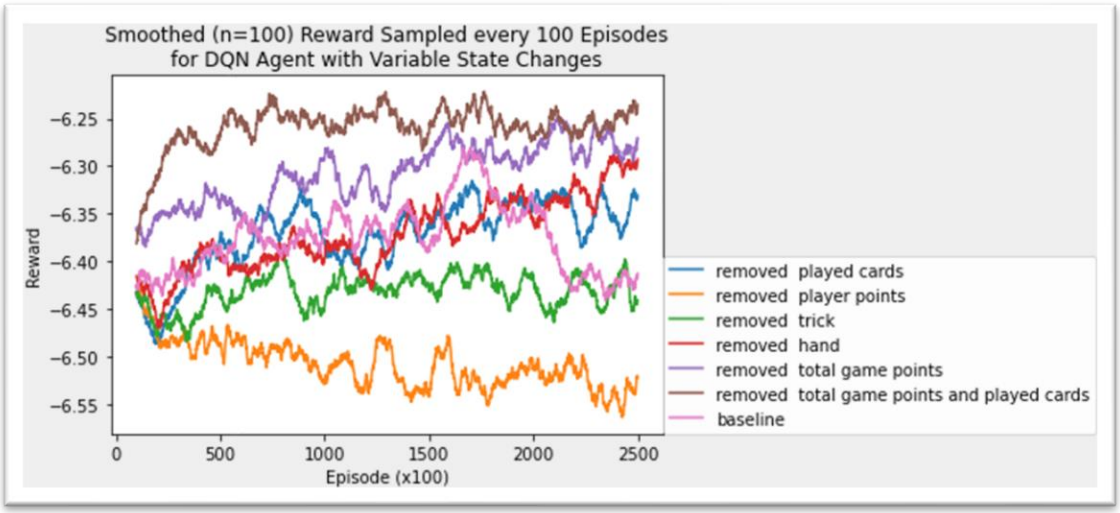

Figure 2: State Change Experiment results

The state modification involved individually removing each major component discussed in the State Engineering section, and then training a separate DQN agent for 250 thousand episodes with the modified state. As well, an additional experiment was performed that removed both the total game points and the played cards, as their individual results were promising. The smoothed results of the experiment are shown in figure 2.

Discussion

Within a game of Hearts, 26 points are distributed between 4 players (13 heart cards worth 1 point and 1 queen of spades worth 13 points). From this, it can be estimated that the expected average reward per player should be around 26/4 = 6.5 points in a game of hearts. Given this, we can see that both our NFSP and DQN agents outperform the expected baseline, which is evidence of the agents are better than random behavior. However, only the DQN had an upwards trend in the training, indicating it likely was actively learning, while the NSFP was only slightly better than random.

While both agents do better than our estimated baseline the DQN agent seemed to outperform the NFSP agent by a large margin. We speculate that this may be due to the greedy nature of DQN. Because DQN prioritizes taking what it believes is the most valuable action it lends itself to more aggressive play which may perform better against random agents. NFSP is based around an assumption that there is a strategy to learn to play against. With random agents there is no such strategy for it to pick up, which may be confounding the agent.

In the state change experiments, the performance decreased when the player points were removed from the state. All the other experiments had upwards trends and stayed above the baseline, but removing the agent's points plummeted the reward, making it worse than random play. This trend is likely due to the agent not being able to correlate collecting points taken to having a negative reward. By maintaining the count of a player's points in the game, the player can make calculated risks each trick. Without knowing the points, the player has accrued so far in the game, the agent can't relate cards collected to specific points, so it can't learn trends in the game. From this, we can guess that by not knowing the points collected in the game, the agent has riskier play and gets worse performance.

Meanwhile, the exclusion of portions of our state resulted in better DQN agent performance. In particular, excluding game points or game points *and* played cards had a positive effect on the performance. We suspect that the exclusion of these elements from the state representation results in the agent playing more aggressively, e.g., playing high cards, when possible, which should be an effective strategy against random agents, as random agents cannot plan out how to win tricks. This may indicate that the agent was learning to collect all the tricks, which would result in "shooting the moon." Alternatively, by removing un-needed state space, it meant the agent had less information to focus on, making it understand the rest better. In this instance, by removing context from the history

of the game, the agent could focus on whether to take the current trick or not, using more aggressive tactics when it'd be advantageous to take the current trick.

Conclusion

This research project successfully applied reinforcement learning techniques, utilizing the RLCard Python library, to develop AI agents capable of playing the card game Hearts. Through the implementation of Deep Q-Networks (DQN) and Neural Fictitious Self-Play (NFSP) algorithms, the DQN agent demonstrated superior performance over the NFSP agent, likely due to its more aggressive playstyle, which proved advantageous against random agents.

Furthermore, the study highlighted the importance of state representation in reinforcement learning. Removing certain components from the state significantly changed the performance. Removing player points negatively impacted the agent's performance, making it worse than random agents since it couldn't relate points collected to the reward it obtained. Removing the game points and the played cards from the state resulted in the best agent performance, likely in more aggressive playing and a focus on capturing better tricks. These results highlight the significance of maintaining a comprehensive state for effective decision-making of reinforcement learning agents.

For future work, it is suggested to explore additional modifications to the state representation and using additional algorithms for training the models. Additionally, to better simulate competition, it would be beneficial to evaluate the agents against more sophisticated opponents to further evaluate their capabilities. Overall, this study contributes to the advancement of AI systems in game playing and underscores the potential of reinforcement learning in strategic decision-making contexts.

References

[1] E. Steinberger, "PokerRL," GitHub repository, 2019. [Online]. Available: https://github.com/TinkeringCode/PokerRL

[2] Geiser, Joshua, and Tristan Hasseler. Beating Blackjack - A Reinforcement Learning Approach. Stanford University, doi: https://web.stanford.edu/class/aa228/reports/2020/final117.pdf. Accessed 23 Mar. 2024.

[3] F. Al Mahamid and K. Grolinger, "Reinforcement learning algorithms: An overview and classification," in 2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Sep. 2021. DOI: 10.1109/CCECE53047.2021.9569056

[4] D. Zha et al., "RLCard: A Toolkit for Reinforcement Learning in Card Games," arXiv:1910.04376 [cs.AI], 2020.

[5] N. Brown et al., "Combining Deep Reinforcement Learning and Search for Imperfect-Information Games," arXiv:2007.13544 [cs.GT], 2020.