FAI FINAL

宸宇 林

June 2024

1 Introduction

Texas Hold'em poker is a complex and challenging game to solve due to several factors. The game involves hidden information, as players' hole cards are private, creating uncertainty and requiring inference about opponents' hands. Additionally, the large state and action space, with numerous possible game states and strategic decisions across multiple betting rounds, adds to the complexity. These elements make developing an effective poker bot a demanding task.

2 Methods tried

2.1 Win rate prediction

Texas Hold'em involves a combination of 52 cards, leading to a vast number of possible hand combinations and board states. Even though our competition is a simplified version with only two player, it's still difficult to calculate and store all possible combinations of win rates. Therefore, I've tried two different ways to predict the win rate in real time. They are card embedding and Monte Carlo simulation.

2.1.1 Card embedding

Card embedding reduces this high-dimensional data into lower-dimensional vectors, making it computationally feasible for AI models to process and learn from the data. There are several ways to encode the cards, such as CNN and transformers. For simplicity, I've tried the CNN approach. The implementation is based on this repository [1], which transforms cards into image like vectors, then the vectors are passed to a classifier to determine which hands are more powerful.

2.1.2 Monte Carlo simulation

The other way I've tried is to run the Monte Carlo simulation. Given my hole cards and the current community cards, it will first fill the community cards to 5, then randomly assign 2 cards to the opponent. Afterwards, I use the built in *HandEvaluator* to see who wins in this round. The simulation can be run for multiple times. I decide to run 10000 times to achieve a reliable result.

Even though most of the time both methods yield great result, card embedding might be unstable under certain circumstances. It's possible that the instability for card embedding can be mitigate after fine tuning the parameters. However, considering the performance of Monte Carlo simulation is good enough, I decide to adapt it in the following implementation.

2.2 Model design

2.2.1 Deep Q-Networks

Deep Q-Networks, a.k.a. DQN, is a type of reinforcement learning algorithm where an agent learns to make decisions by interacting with the environment. The agent receives rewards based on its actions and aims to maximize the cumulative reward over time. In DQN, a neural network approximates the Q-value function, which estimates the expected future rewards for taking a specific action in a given state. The idea is to encode the hole cards along with the round state information to a vector as input to train the model by reinforcement learning. I've implemented it with the assistance of emulator class in this repository [2] and train it by playing with different types of players.

2.2.2 Rule-based

Considering the restricted competition rules, implement a rule-based model to mimic human can have decent performance. After studying from world champions (by watching videos) and manually playing with baseline players, I set up complicate rules so that my player will take action base on current win rate, breakeven rate, seats (big blind or small blind), opponent's previous action, round count, current stack, money spent in this round, opponent's habit and so on. The detail of those rules will be described in the **Implementation details** section.

3 Comparison of my methods

For win rate prediction, considering the performance and stability, I choose Monte Carlo simulator as my predictor. The number of simulation for each given hands I set to 10000, since number lower than that might lower the reliability of predicted win rate, number greater than that might exceed the timing constraint (10000 simulations take around 2 seconds).

For model selection, rule-based model outperforms DQN model in my implementation. One possible cause might be the large variance of different players' strategies. DQN player might overfit on certain player and result in poor performance against another player. On the contrary, my rule-based model will take different actions when facing with different types of players (by observing their previous actions). On top of that, training a DQN model is time consuming, while design a rule-based model can be evaluated in just minutes using multiprocessing. Thus, I'll choose rule-based model to submit.

4 Implementation details

After playing against baseline players and learning from real world poker champions, I found out that the poker players can be categorized into 3 types: **honest player**, **bluff player** and **smart player**. Honest player takes actions based on the win rate of his hands, so raise a small amount of money frequently is a good strategy against him. Bluff player tends to raise despite having weak hands. It is better to be patient and tries to re-raise if having a strong hands. Smart player strikes a great balance between those two, and he'll dynamically adjust his strategy based on his current stack. If his net profit is negative, he is more likely to all-in, trying to bluff his opponent. Based on these domain knowledge, I design the following criteria.

4.1 Early stopping

Since our game only takes 20 rounds, and each round I'll lose at most 5 or 10 if I fold at the beginning. I can calculate the minimum net profit I need to win the game at each round, and once my net profit

exceed that amount I'll fold rest of the rounds yet still win the game.

4.2 Win rate and breakeven rate

At each round I'll calculate the win rate by Monte Carlo simulation and the breakeven rate, which is $risk/(risk + reward)$. If the win rate is lower than the breakeven rate, I'll fold. Otherwise, I'll take different actions considering win rate and round state. If I'm small blind in this round, I'll be more aggressive, which is calling or raising with lower win rate, to bluff my opponent. If I'm big blind, and my opponent's previous action is call, I'll tend to bluff because he might have weak hands. But if I'm big blind and my opponent raises, I'll be more cautious since he might have strong hands. I'm not going to write down the exact threshold to take each action (which is in the code), but the overall concept is don't be too conservative and tries to bluff even with weak hands. As illustrated in [3], raising is preferred under most cases to provoke an immediate fold by the opponent.

4.3 Dynamic adjust

Dynamically adjust the win rate threshold base on the net profit and the round count. If the net profit is positive, the model tends to be more cautious. On the contrary, if the net profit is negative, it's more likely to raise. The model will be more aggressive exponentially to the round count since with fewer round the chance to get a nice hand become lower.

4.4 All in

As described before, raising is more preferred under most cases. All in is the strongest way to provoke a fold by the opponent. For most player, they tend to fold facing an all in if their hands are not perfect. Therefore, if my hands are good enough, I'll all in to bluff. Just as I can calculate the money needed to win at each round, I can calculate the money needed to lose as well. If I found that I'm going to lose in the near future, I'll all in without hesitation.

4.5 Handle bluff player

The model will count the frequency the opponent decide to raise. If he raise too often, I'll define him as a bluff player, and I'll adjust my strategy by being more patient to catch his bluff.

5 Discussion and conclusion

Player	1	2	3	4	5	6	7
Win	236	300	300	189	298	224	153
Lose	64	0	0	111	2	76	147
Win rate	79%	100%	100%	63%	99%	75%	51%

Table 1: The results after playing 300 game against each player.

I've played 300 game each against baseline1 to baseline7, and the results are demonstrated in 1. As shown in the table, my model performs well against baseline1, 2, 3, 5, and 6, but the win rate against 4 and 7 is lower than 65%. After observing their strategies, baseline4 is a bluff player while baseline7 is a smart player. I found it is difficult to design a perfect strategy against both of them. A more aggressive model might beat baseline7, but might be defeated by baseline4 since it'll call or even raise my bluff. Perhaps a better solution is to design complete different strategy against those two types of players, however its result will largely depend on the way we distinguish them and might cause worse result.

6 Method I choose to submit

- Model selection: rule-based
- Win rate prediction: Monte Carlo simulation

References

[1] EvgenyKashin. Tensorpoker. https://github.com/EvgenyKashin/TensorPoker, 2017.

[2] Ishikota. Pypokerengine. https://github.com/ishikota/PyPokerEngine, 2016.

[3] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.