Reinforcement Learning in Heads-Up No-Limit Texas Hold'em

David Choi and Jack Kelley

December 19, 2018

1 Introduction

The goal of this project was to develop a learning agent that plays Heads-Up No-Limit Texas Hold'em poker, a game of imperfect information and uncertainty. We wanted to examine how well reinforcement learning, which is often used in perfect information games, would fare in an imperfect information game. We adapted the general concept of an algorithm from Jose Pedro Marques' "Reinforcement Learning applied to the game of Poker" from the University of Porto. We compared and analyzed the agent's performance against MonteCarlo Bot, a simple, simulation-based bot that acts on a preset plan, and against two basic bots, AlwaysCall and AlwaysAllIn.

2 Background and Related Work

Artificial Intelligence in Texas Hold'em poker has been widely explored, with most of the focus being concentrated on neural networks and deep learning, including bots such as Liberatus, which claims to currently outperform professional poker players in No-Limit Heads-Up, and Cepheus, which has "essentially" solved the game of Limit Heads-Up Texas Hold'em. Reinforcement learning specifically, however, has been less explored in the field of poker agents, for one simple reason: folding is a very large part of poker, but does not reveal useful information about the expected reward of the action, as the opponent's cards remain unknown. As such, it becomes difficult to measure the Q-value of folding, a problem we encountered in our own project.

3 Problem Specification

Texas Hold'em poker is a betting game in which players take turns acting with 3 choices: fold, or give up the current hand, call, or match the opponent's previous bet, or raise, making one's own bet. Each player is granted 2 hole cards at the beginning of each round, which is hidden from the other player, and a round consists of 4 rounds of betting, called 'streets'. At each street, some amount of community cards are revealed to all players, and the goal of the game is to create the best 5-card hand out of the total 7 available to each player (5 community and 2 hole cards). As such, there are 2 ways to win a round: reach the end of the final street, called 'the river', and have the better hand, or bet in a way as to cause the opponent to fold on any street before the showdown.

Hand Ranking (h)	$[0 \ldots 3]$
Street (s)	$[0 \ldots 3]$
Opponent Action (a)	$[0 \ldots 2]$
Position (p)	[0, 1]
State	(h, s, a, p)

Table 1: State Space

In order to create a state space for a learning agent, we decided on 4 criteria: hand ranking, street, opponent's action, and position. Hand ranking, or the quality of the current hole cards with any revealed community cards, was evaluated through MonteCarlo sampling of win-rate: at each instance, we simulated 100 times how our current hole and community cards would fare against 2 random cards the opponent could have, plus random cards for the remaining community cards that have not yet been revealed. We measured how many times out of the 100 we won, estimating the strength of the current hand as well as the potential for the hand to develop into a better hand in later street. We divided the win-rates into 4 divisions, with win-rates 0-45, 45-60, 60-75, 75-100 as the cutoffs. We assigned a value for each street a state could be in, for 4 total streets: preflop, flop, turn, river. We took into account the last action the opponent took which could result in a state for the agent, for 3 values: no action (as in the agent is first to act), call, and raise. Finally, we recorded the position of the agent in each state, which could be Dealer (who acts last), and Big Blind. Thus, we had a total of 96 possible states.

4 Approach

The agent's learning goes as such: at any given decision point, the agent is in a state which it looks up in the Q-Table. The value for each state in the Q-Table is a tuple, $(h, s, a, p) \rightarrow (f, c, r)$. It then normalizes the values of the tuple as such:

$$f \rightarrow \frac{f}{f+c+r} \quad c \rightarrow \frac{c}{f+c+r} \quad r \rightarrow \frac{r}{f+c+r}$$

Then, the agent takes an action with the following probabilities:

$$call \rightarrow [0 \ldots c] \quad raise \rightarrow [c \ldots c+r] \quad fold \rightarrow \text{if else}$$

The agent stores all states explored per round, and at the end of the round, updates all explored states according to the table below where X represents the amount won or lost in that round. Essentially, the assumption is that if the agent called and won, perhaps it should have raised earlier. If the agent called and lost, perhaps it should have folded earlier, and so on. The algorithm attempts to encourage behavior that resulted in winning and discourage behavior that resulted in losing. Note that due to the fact that when one folds the opponent's cards remain hidden, folding tends to be a self-promoting action, as it makes it more likely the agent will continue to fold in the future. Also note that the structure of the algorithm inherently promotes some randomness in the agent's action, as to make it less predictable and thus less exploitable.

The Q-Values for each state in the Q-Table were initialized to random integers except for the fold value, which started as 0. The idea was that folding is the least revealing of actions, as the round ends without further information about the opponent's hand, so it is better that the agent calls or raises in order to observe the opponent's reactions.

Action	Outcome	Update	Reward
Fold	Loss	Fold	+2X
		Call	-X
		Raise	-X
Call	Win	Fold	-X
		Call	+X
		Raise	+2X
	Loss	Fold	+2X
		Call	-X
		Raise	-2X
Raise	Win	Fold	-X
		Call	-X
		Raise	+2X
	Loss	Fold	+2X
		Call	+X
		Raise	-X

5 Experiments

We tested our agent against 3 opponents: AlwaysCall, which always calls, AlwaysAllIn, which always raises all-in when given the chance, and MonteCarlo, which uses a similar simulation algorithm as the previous hand-ranking algorithm in order to raise with hands with high win-rates, call with hands with moderate win-rates, and fold with hands with low win-rates. As a baseline, below are the results of MonteCarlo playing against the other 2 test bots over approximately 20,000 hands.

AlwaysCall	AlwaysAllIn
20 bb/100 hands	16 bb/100 hands

5.1 Results

As the learning agent relies on its present data-set to make decisions, we found that the quality and source of the data had a large impact on its performance. After learning 10,000 hands versus each agent consecutively into the same Q-Table, the learning agent performed as such over 20,000 hands:

MonteCarlo	AlwaysCall	AlwaysAllIn
-12 bb/100 hands	8 bb/100 hands	4 bb/100 hands

As shown, when the data in the Q-Table was sourced from 3 different opponents, the learning agent struggled against each of them, underperforming compared to the MonteCarlo bot. When the learning agent was given 3 different Q-Table datasets to match each of its opponents, learning for 30,000 hands against each bot, it performed as below over 20,000 hands:

MonteCarlo	AlwaysCall	AlwaysAllIn
11 bb/100 hands	27 bb/100 hands	21 bb/100 hands

6 Discussion

Our results show that the current form of the learning agent is very opponent-specific. Given that the opponents it played against were polar in playstyle, it struggled to reconcile the different strategies using one dataset for all opponents. When given new datasets to train against each specific opponent, the learning agent performed much better, beating MonteCarlo by some margin and beating AlwaysCall and AlwaysAllIn by larger margins. Nonetheless, our agent was not able to outperform MonteCarlo by a significant margin, indicating issues with the algorithm. Examining the Q-Tables manually, it appeared that folding often became a self-reinforcing action, causing the agent to fold too frequently, detracting from its profits when it does bet out. Additionally, given the reward system currently being used, the learning agent developed a general strategy within about 5,000 hands, and struggled to change or adapt its strategy in subsequent games. Given the non-deterministic, highly variant nature of poker, it seemed that some statistical anomalies could potentially cause problems in the agent's policy.

In the future, we would like to explore variations of the algorithm, including changes to features such as the number of simulations it runs when it evaluates the strength of a hand. As it stands the agent simulates 100 hands per state, which was chosen for performance concerns, although it seems the number may be too low to consistently rank hands accurately. In addition, we could try to add more 'buckets' to the hand ranking criteria, which may make the agent more robust, although a few additions could increase the state space by a significant amount. Another major concern was the agent's inability to play against multiple styles of opponents using the same dataset. In the future, this issue could be solved by adding additional features to the state space that account for opponent playstyle, namely the opponent's level of aggression and range of played hands. Finally, the issue of folding being a self-reinforcing, negative reward action should be addressed. An algorithmic solution could exist, although from our tests and from literature it seems that it may be and issue with the game of poker itself that makes reinforcement learning less efficient as a learning strategy for a poker playing agent. As it is difficult, for computer agents as well as humans, to tell whether a fold was the 'correct' move at any given decision point, it is difficult to formulate a reward system for reinforcement learning that effectively captures the reward of folding.

7 System Description

Our experiments were conducted in the PyPokerEngine public engine. The code for each agent is in a file named after its agent. The main simulating of games is done in the simulate.py file. It currently contains 6 'flags' for running games, set to a boolean value, with each representing a different match-up of bots to be run. Each match-up has 2 values that need to set: the value of the loop, representing the number of games to simulate, and the max_round value, which represents the max number of hands to be played per game if neither player loses all its chips before then. To simulate a match-up, flag the corresponding boolean to True, set the desired number of games/rounds, and run python simulate.py. The console will print the average stack size over all games for player 1, as well as the number of games completed. It is also possible to see the individual actions that a player makes by setting the verbose flag in start_poker to 1. The Q-Table for each match-up is saved as a .pkl file using the python Pickle library, and loaded into the learning agent on intialization.

8 Group Makeup

This project was a collaboration between David Choi and Jack Kelley. Jack did a bulk of the initial research into possible algorithms and literature on reinforcement learning, and David took the lead on implementation, although the code was worked on together through Github. David took the lead on writing up the report.

9 Bibliography

PyPokerEngine - https://github.com/ishikota/PyPokerEngine

Ball, Philip. Game Theorists Crack Poker. Scientific American, Nature, 8 Jan. 2015

Marques, Jose Pedro. Reinforcement Learning applied to the game of Poker. University of Porto, 2013.

Rupeneite, Annija. Building Poker Agent Using Reinforcement Learning with Neural Networks. University of Latvia. (2014)