COMP423 - Reinforcement Learning and Dynamic Optimization
Poker Project Part 1 Report

Pantourakis Michail AM 2015030185
School of Electrical and Computer Engineering
Technical University of Crete

Date submitted: 30 June 2023

1 Introduction

A significant step towards understanding today's advancements in Reinforcement Learning (RL) is by first understanding the fundamental theory of Markov Decision Processes (MDP) for modeling decision making problems and the "exact" model-based algorithms that optimally solve these problems. However, these approaches are limited by their requirement for complete knowledge of the environment (model). A major breakthrough in RL was thus the development of the Q-learning algorithm, the first model-free learning algorithm with guaranteed convergence to the optimal policy.

In this report I present my course project implementing these algorithms and analyzing their ability to maximize earnings on a simplified, "toy" version of poker. I will start by first establishing the rules of this simplified game, the opponents designed as part of the environment, my model representation and how I implemented all of them in Python. With the problem formulation being established, I will then continue with how I applied Policy Iteration (model-based) and Q-learning (model-free) algorithms to this problem, reporting key observations, and most importantly, analyzing their behavior due to the nature of this game and what theory predicts.

2 Environment

2.1 Simplified game rules

Both "exact" algorithms are limited in solving problems with a relatively small number of states and actions. As poker is characterized by numerous parameters (number of players, position/order of players, legal actions, bidding round number, betting chip/token allowance, hidden and public cards, winning card combinations), the state-space quickly has a large number of states. To alleviate this issue for the purposes of this project, a simplified version of the most popular poker variant, Texas hold'em was used. Since Texas hold'em is a well-known game and its rules are also summarized in the project description, I will only provide a quick summary of its rules here, with emphasis on the unique deviations assumed here.

First of all, a dumped-down version of heads-up limit Texas hold'em is used as a basis, thus only 2 players participate. Both players start with a card in hand (not the normal of 2 cards) and with betting the same mandatory amount of 0.5 tokens (namely no small/big "blind" difference).

The game continues with a bidding round. Since this is a limit variant, either 1 (actions "bet" or "raise") or 0 (actions "check" or "fold") tokens can be placed by a player per turn of action, with a maximum of 2 tokens placed in total by each player per round. If a player folds, the other player is an instant winner and is rewarded with the full amount of tokens bet so far. Otherwise, 2 cards are further drawn and are publicly revealed, and a second bidding round, similar to the first one, begins. Unlike the real game, the order (position) of players in which they are required to take action remains the same for both rounds. For simplicity, no further rounds and card draws take place. Table **??** summarizes the legal order of actions in all game states.

Assuming no player has folded until the end of round two, the winner is then decided by comparing their "hand strength". To reduce the number of possible combinations, only cards with rank 'T', 'J', 'Q', 'K', 'A' (in ascending order) are available in deck (20 cards in total, 4 suits per rank). This simplification leaves only three winning combinations of hand and public cards: 1) 3 cards of the same kind (rank), 2) a pair of same kind, 3) no pairs. The rank of the pair or hand is used as a tie-breaker in rules (2) and (3). If all tie-breakers fail, the result is a tie and the tokens placed by both players are returned to them.

+++ Add table

2.2 Opponents as part of the environment

Two different types of "static" (non-adversarial) agents were created in this project. These opponents serve as both necessary components of the full state-space formulation required by Policy Iteration, and benchmarks for the performance of Q-learning.

The first opponent, hereafter called *Random agent*, is a completely randomized agent. Each time an action is required by it, it randomly picks -with equal probability- one of the actions allowed according to the game's current phase. Since the Random agent disregards any other card and opponent information, its opponent cannot infer any meaningful information from its actions.

The second opponent, named *Threshold agent*, is a completely deterministic agent, using only the strengths of its cards to determine its next action. Table **??** summarizes the action this agent will perform at any possible step. In contrast to the Random agent, each and every action of the Threshold agent provides information on the range of possible ranks held in its hand. Therefore, its predictable nature can be used against it by any agent trained on it.

+++ Add table

2.3 MDP and state/action space representation

Before proceeding with the implementation and results of this work, careful formulation of the state-space representation is crucial. This representation should balance between two opposing ends: 1) be as complete as possible to capture all information required by the "exact" algorithms used here, 2) have the smallest size possible to reduce computational demands without sacrificing completeness. Table 1 summarizes all features, also discussing both their necessity and how their value range was optimized.

Each state of the MDP is coupled with its legal actions only. Each state-action pair is then linked to one or more possible transitions. Each transition is in turn characterized by: 1) its conditional probability (given the state-action pair), 2) the next state, 3) the associated reward (which for a loss is negative), and 4) terminal status (as boolean). Note that in Texas Hold'em games, all intermediate states have 0 reward, and only terminal states may have non-zero values (and 0 only in case of ties).

2.4 Implementation

The entirety of this project was implemented in Python. All simulations and results reported here can be reproduced by running the Jupyter Notebook file `notebook.ipynb`.

In regards to the environment implementation of this simple poker variant, I adapted the corresponding object-oriented game structure of the rlcard Python library. Although this library may have already contained a functional environment for a number of poker games, this project's unique game rules demanded novel implementation of almost all classes.

In summary, the `Card` class models each possible card of the game, while the `Dealer` is responsible for maintaining the deck and drawing cards. The `Judger` class contains the aforementioned rules of winning and is responsible for deciding the payoffs of each player (implemented as `Player` objects containing the state features known to each one). A `Round` object encapsulates all rules associated with player position, legal actions of currently active player, effects of selected actions, and round completion. The `Game` class is responsible for synchronizing all the aforementioned objects so that poker games can be executed as described.

Feature	Value Range	Necessity	Minimization
position	'first', 'second'	Playing first or second has a direct consequence on the list of legal actions, and in case of Threshold agent, on the inferred hand range.	Unlike real poker, the simplification of retaining the same position for round 2 (flop) reduces number of possible transitions.
chips placed so far by currently acting player	'0.5', '1.5', '2.5', '3.5'	Knowing how many chips are already committed by player directly contributes to eventual rewards/losses.	'4.5' is omitted since it is only encountered in game terminal states
difference in chips of opposing player	'-1', '0', '1'	Knowing how many chips are already committed by opposing player efficiently merges information about eventual rewards/losses and current player's legal actions.	Since this is a limit hold'em game, this enumeration is smaller than using '0.5', '1.5', '2.5', '3.5', '4.5' directly.
rank of card in hand of currently acting player	'T', 'J', 'Q', 'K', 'A'	Strength of the card directly affects the player's chances of winning if no player folds until the end of round 2	Suit does not matter at all in the simplified winning conditions of this game version, therefore it can be completely omitted from state-space representation.
rank of remaining possible opposing player cards in hand in alphabetical order	'none' if not revealed yet, else all 10 combinations (e.g. 'AK', 'AJ', 'JK' etc.)	Strength of the card directly affects the player's chances of winning if no player folds until the end of round 2	Suit does not matter at all in the simplified winning conditions of this game version, therefore it can be completely omitted from state-space representation. Furthermore, the order of public cards does not matter in this variant, hence the alphabetical order offers a consistent way of reducing the number of states.
rank of remaining possible opposing player cards in hand in alphabetical order	Based on Table **??**: 'AJKQT' (no information), 'JQT', all 10 combinations of public hands, and all ranks (when the opposing hand's rank is surely known)	Needed only for Threshold agent, the strength of the opposing hand directly affects the player's chances of winning if no player folds until the end of round 2, and it also determines the probability of the Threshold agent's next action	When the opposing agent is not known to be the Threshold agent, only 'AJKQT' is used. See above too.

Table 1: Simplified poker state-space representation utilized by Policy Iteration and Q-learning algorithms.

As far as player types are concerned, they are modeled as agent classes. The two static agent models described in Section 2.2 are implemented by the `RandomAgent` and `ThresholdAgent` classes. Moreover, the agent classes trained using the Policy Iteration and Q-learning algorithms include the implementation of these algorithms and are named `PolicyIterationAgent` and `QLearningAgent` respectively. As a way for a human player to test the environment and play against a computer-controlled agent, the `HumanAgent` class is also included (see file `play.py` for an example script to try it out).

To run games, the `Env` class brings together the `Game` and opposing agent objects, is responsible for invoking the `Game` functions and extracting the complete current state representation. Finally, file `seeding.py` contains random seeding utility functions offered by rlcard, while a few generic utility functions are implemented in file `utils.py`.

3 MDP-based solution: Policy Iteration algorithm

Prior to experimenting with a true learning algorithm in this game, we first want to acquire a clear understanding of the environment. As already mentioned, modeling the environment (the combination of game rules and opponent behavior) as an MDP enables us to solve the problem optimally with dynamic programming methods such as Policy Iteration. Therefore, I start this section by highlighting my Python implementations of the MDP described in Section 2.3, for both Random and Threshold opponents. Next, I rationalize my choice of hyperparameter configuration for Policy Iteration, I continue with reporting the results of this algorithm, and finally conclude with an analysis of the yielded optimal policies.

3.1 MDP implementation

As shown in Table 1, a state is fully characterized by six features. Since my primary goal was to enhance readability of my implementation and results, I sacrificed memory usage efficiency by selecting a descriptive state representation using Python's dictionaries over lists. In particular, a two-level nested dictionary structure was formed, with state being the first level key, action being the second level key, and each state-action pair containing a list of all possible transitions (which are as described in Section 2.3). Following Table 1, the syntax of state keys is `{{position}}_{{my_chips}}_{{opponent_chips}}_{{hand}}_{{public_cards}}_{{opponent_range}}`, while the action values of Table **??** were directly used as action keys.

Since transition probabilities are entirely determined by 1) opponent action probabilities (opponent model), and 2) poker card probabilities (game rules), I needed to implement a way of calculating these odds for the unique poker variant of this project. Since the former is part of the game and irrespective of opponent models, I developed the function `get_transition_probabilities_for_cards()` in the Game class. This function returned a dictionary containing the odds of each card pairs (as ranks) being drawn in flop knowing our hand and our opponent's range (resulting file `win_probabilities.json`). Now given the public cards as well, this method also provides the odds of winning (file `win_probabilities.json`) and losing (file `loss_probabilities.json`), with tie odds also being trivially calculated. Finally, resulting file `range_probabilities.json` includes the odds of the opponent having a certain card range given their current range and all other revealed cards.

In regards to opponent action probabilities, these of course differ between Random and Threshold agents based on their definition in Section 2.2. To this end, their calculation in both corresponding classes is part of function `calculate_state_space()`. With this function I programmatically calculate all possible state-action-transition combinations. In the case of Random agent this function yielded 1000 non-terminal states and 2230 non-terminal state-action pairs (see file `random_agent_state_space.json`), whereas for Threshold Agent the state space was larger with 2075 non-terminal states and 4500 non-terminal state-action pairs (see file `threshold_agent_state_space.json`). The reason behind this difference is due to inference of the Threshold opponent's possible range of hands based on its actions, whereas we cannot infer any information based on a Random opponent's actions (see last feature of Table 1).

3.2 Algorithm implementation and hyperparameter configuration

For the development of Policy Iteration agent, I adapted the example for the Frozen Lake environment provided in class. Most of the provided implementation remained unchanged, however since I decided to use dictionaries for the representation of state-action-transitions, I had to replace the functionality meant for tabular representations with an equivalent for dictionaries.

Policy Iteration offers a pair of hyperparameters: 1) the convergence error, and 2) the discount factor γ. My chosen values for both hyperparameters was supported by the episodic nature of limit poker, since all games terminate after a very limited number of state transitions. Therefore, for convergence error I chose the default low value of 10^{-10} (since fast convergence is expected). Furthermore, I also defaulted $\gamma = 1$. Choosing a lower value for γ in poker is expected to motivate a more aggressive gameplay, trying to winning games as fast as possible by forcing opponents to fold. However, I did not go that route since 1) rewards are equally important regardless of how many steps a game lasted, and 2) longer lasting games may offer more opportunities for increased rewards, especially against simple opponents that have a high chance of choosing subpar actions.

3.3 Results

Having set the MDP and hyperparameters, we can now evaluate the performance of Policy Iteration both in terms of convergence to the optimal policy and of average payoffs per game.

First, as expected, convergence is rapidly achieved. Although speed of convergence depends on policy initialization which is random, most of the times only 3 iterations of policy evaluation and improvement are enough to reach the desired level of convergence (2-4 has been observed in numerous tries). Note that lowering γ a bit had only a effect on the resulting optimal policies, nevertheless this small effect was towards the expected direction of more aggressive playstyle (more bets/raises as optimal policies). These results were consistent against both Random and Threshold opponents.

Concerning the average payoffs achieved by the generated optimal policy per opponent, I tested them against their intended opponent in 5000000 simulated games. Expectedly, the Random agent was proven to be the worst player, providing a much larger margin for earnings (mean: 0.8761 tokens, standard deviation: 2.127 tokens) when compared to the more 'rational' Threshold agent (mean: 0.2251, std: 1.467).

3.4 Optimal policy analysis

Interpreting the optimal policy generated by Policy iteration requires a firm understanding of the environment. Furthermore, proving its optimality is challenging due to the large state-space of this game. Nevertheless, in this section I present a brief interpretation of the policies generated for both opponents, and a few sanity checks to support that they are indeed optimal.

3.4.1 Random opponent

Inspecting the optimal policy for Random opponents (file `random_agent_optimal_policy.json`), most profits are made by abusing the odds of it randomly folding at any step of the game. This is especially obvious when being in the advantageous second position, since the policy of always raising has at least a 50% chance of winning the game due to a random fold, and thus raising is always the optimal action ('fold' and 'bet' are the only legal options of the Random agent, let alone any chances of winning due to superior hand in case of 'bet').

Similarly, betting at the beginning of a round when playing first always grants you at least 1/3 chances of winning. Therefore, in 215 out of 230 such states the hand strength is enough to make betting is the optimal action, while the remaining 15 states are post-flop cases with very weak hands, where 'check' is indeed optimal. The optimal policy is slightly more conservative (75 out of 300 such states where fold is better than bet) is in the last possible action of a game after a post-flop raise, since then the chances of winning are entirely based on hand strength. These are the only states where fold may be optimal.

Finally, as an extra sanity check, I pitted the Random and Threshold agents against each other in

the same experimental setup (5000000 games). Even though the Threshold agent is a more rational player and still beats the Random agent, the results (mean payoffs: 0.3093, std: 1.643) clearly shows that it is far from being the optimal policy.

3.4.2 Threshold opponent

Analyzing the optimal policy for Threshold opponents (file `threshold_agent_optimal_policy.json`) is not so straightforward, since we can now "read" these opponents based on their actions, resulting in a larger state-space. Nevertheless, a few trends are also apparent here and are ultimately entirely based on card odds (even opponent actions), which in general results in a tighter playstyle when compared to the optimal policy for Random opponents.

Seeing a few examples: 1) Despite the weak hand, raise is optimal for state `second_0.5_0_T_AK_JQT`, since the Threshold opponent also has a weak hand and will fold more times than call (will call only if it has a Q). 2) For state `second_0.5_0_T_AJ_Q`, check is optimal because a loss is guaranteed; the opponent will never fold with a Q, and T loses against Q anyway. 3) For state `first_0.5_0_A_KQ_AK`, check is the best action because there is no chance for a Threshold agent to fold here, and the best we can do is a tie (if it has A). 4) For state `first_0.5_0_A_KK_JQT`, our win is guaranteed and betting may also increase our reward (if the Threshold agent has a Q will call).

In conclusion, as we will see in the following sections (and Tables **??**-**??**), Policy Iteration indeed yielded the best policy per opponent out of all policies shown in this project.

4 Model-free solution: Q-learning algorithm

4.1 Implementation

4.2 Hyperparameter configuration

4.3 Results

4.4 Convergence analysis

5 Conclusion

- Main point for game size & performance of algorithms
- Suggested next steps/improvements