Reinforcement Learning and Dynamic Optimization

Poker-Playing Agents using Policy Iteration and Q-learning

Final Course Project

Petros Bimpiris Konstantinos Nikolos

Spring Semester 2023

Introduction

In this project we implement the Policy Iteration and Q-learning algorithms for exact, value-based reinforcement learning. The environment is a simplified version of Texas Hold'em Poker[1]. We observe their performances versus a random and a *threshold*[2] agent and comment on the results.

1 Policy Iteration

1.1 Environment

There are three possible states: Check or Bet (Bet is the first raise done at each round), Call Raise or Fold and Call, Fold. The state of the player is decided by the last action the opponent has done. In the beginning of a game, player 1 always starts with check or bet available.

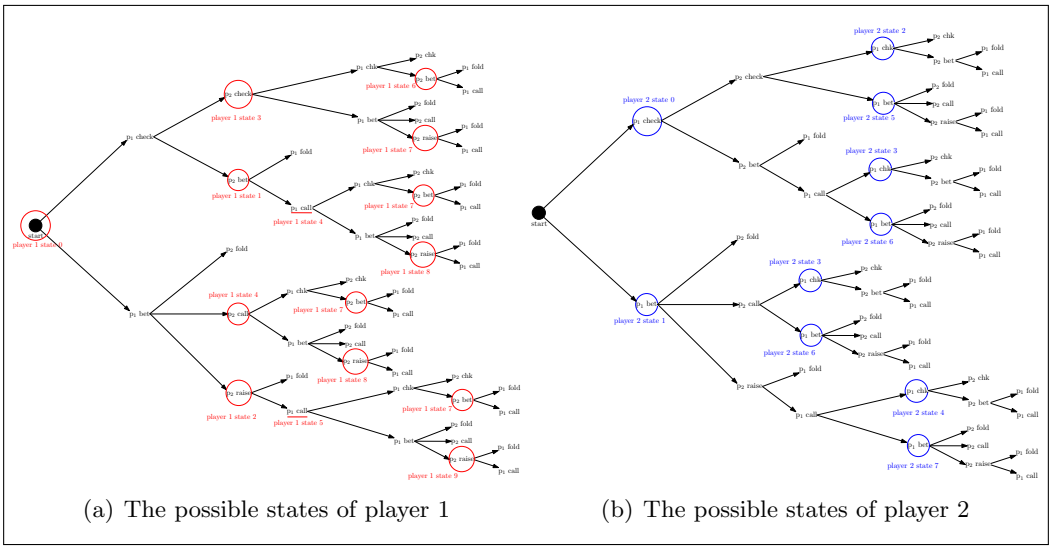
The main part of this project is the implementation of the agents in Python, found in the notebook files to which this report is complementary.

The environment keeps track of the actions of both players, the bets that have been done, the available cards and updates the available state for each player.

- Playing: Round 1 starts with only players cards revealed to them. Player 1 takes an action Check or Bet which will decide if player 2 will be in state Check or Bet or in state Call Raise Fold. If a bet is done by player 1 player 2 can raise also leading player 1 in state Call or Fold. If player 2 calls or checks back we move on to the next round where the public cards are revealed, and players can bet again. A game ends when a player folds or a call is done in round 2 by any player.

[1] there are only 2 players, each holding only 1 card, only 2 public cards and bets/raises are limited to 1 chip. There is also a 0.5 chip ante

[2] a *threshold* agent is one that only bets/raises if the state of the game (agent's hand, public cards) satisfies a certain (threshold) condition



(a) The possible states of player 1 (b) The possible states of player 2

Figure 1: State spaces for Policy Iteration

- Judger: at the end of a game a judger is called which checks both players' hands and public hands, in order to decide the winner. An amount of chips is returned to its player as a reward, positive if he won, negative if he lost, or 0 if it was a draw.

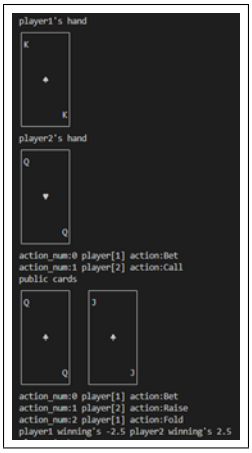


Figure 2: Example of the environment

1.2 Algorithm

The dictionary implemented for the policy iteration follows 11 states for player 1 and 9 states for player 2. Each state has the 5 available actions Check, Bet, Call, Raise, Fold with the unavailable actions at each state marked with a huge negative reward. In detail for player 1 there are 2 possible states Check or Bet and Call or Fold. But to keep track of the rewards we have multiple Check or Bet states with same transition probs but different rewards and states to go.

Transition probabilities are given by a function which calculates the probability of the opponent taking a certain action in case of Threshold opponent or there are uniformly

distributed in case of a Random opponent. The rewards are given when there is an action leading to a terminating state win or lose.

The reward is equal to the amount of chips each player has bet in a certain game. In fig. 2 player 1 must decide to fold leading to -2.5 reward which is the number of chips he has contributed so far to the game.

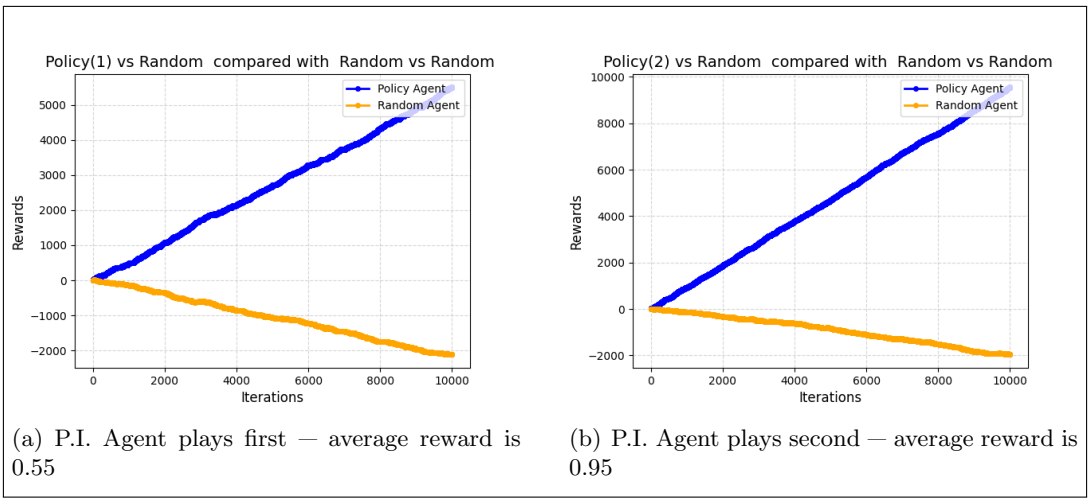
Transition probabilities leading to terminating states are given by a function which calculates the possibility of winning taking into consideration the agent's hand and public cards too. In order to encourage the agent to take action at the first round, which does not lead to a terminating state a small reward is given too by checking the agent's hand, if it is an A for example his reward for calling is 0.3.

There are 2 opponent types, one Threshold which bets/calls with K or A in the first round and a pair on the second round or just an A and one Random who randomly chooses an action. The environment is different for each opponent leading to separate training.

1.3 Performance

1.3.1 Running 10.000 episodes vs a Random Agent

In each plot we can observe the policy agent's cumulative reward playing against a certain opponent compared to a Random Agent playing with the opponent. Testing is done for 10000 games.



(a) P.I. Agent plays first — average reward is 0.55

(b) P.I. Agent plays second — average reward is 0.95

Figure 3: P.I. Agent performance against a random opponent, compared to the performance of a random agent vs a random opponent

Execution time was 31 secs for player 1 (11 states) and 22 seconds for player 2 (9 states)[3].

[3]the numbers may vary between executions, and executions in Google Colab will probably be slower than the ones mentioned here

In both cases we prefer a gamma close to 0.5 which strikes a balance between present reward and later stages, and an epsilon (converge) close to 0.01 which converges faster giving a decent average reward.

In player's 2 case agent plays optimally, winning almost 1 chip per game. He basically chooses to bet or raise most of the time having a good hand like A or K and sometimes checks with Q J T. In most cases agent bets or raises which is expected because of the high probability of player 1 folding with 50% chance or losing as he calls randomly with occasionally having a good hand. On hands like T J Q, he chooses to check instead of betting with lower chances to win on showdown. Average reward is also expected to be between 0.5 and 1.5 as player 1 will check or bet randomly and player 2 will bet or raise leading to player 1 folding with reward 0.5 or 1.5. In 10000 episodes the agent won 8091 times and 5431 of them with a reward of 0.5 or 1.5 with a percentage of 0.67 Policy of player 2

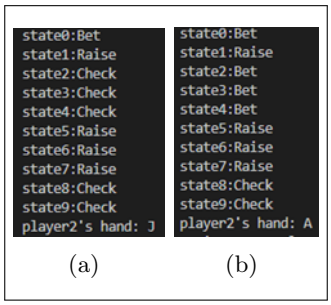


Figure 4: Policy of P.I. Agent as player 2 vs. a random opponent — states 0-1 are round 1, states 2-9 are round 2 and states 10-11 are terminating

In player's 1 case agent chooses most of the time to bet or call again with higher card like K or A but chooses to fold when raised with lower value cards like T or J and no big bets have been placed yet (state 6,7 - see fig. 1(a)) meaning he prefers not to call with a great probability of losing risking for a low amount of chips which makes sense. Average reward is also expected as in most cases agent will bets and opponent will fold winning 0.5 chips. In 10000 episodes the agent won 7259 times and 3259 of them were a 0.5 reward with a percentage of 0.449 which means player 1 bet and player 2 folded.

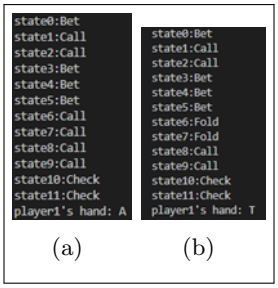
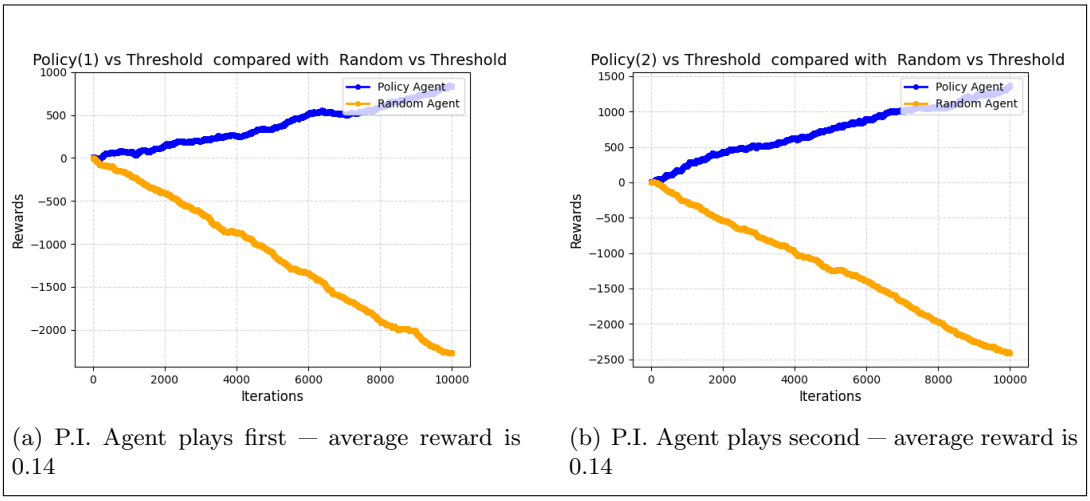


Figure 5: Policy of P.I. Agent as player 1 vs. a random opponent — states 0-1 are round 1, states 2-9 are round 2 and states 10-11 are terminating

1.3.2 Running 10.000 episodes vs a Threshold Agent

In each plot we can observe the policy agent's cumulative reward playing against a certain opponent compared to a Random Agent playing with the opponent. Testing is done for 10000 games.



(a) P.I. Agent plays first — average reward is 0.14

(b) P.I. Agent plays second — average reward is 0.14

Figure 6: P.I. Agent performance vs. a threshold opponent, compared to the performance of a random agent vs. a threshold opponent

Execution time was 36 sec for player 1 (11 states) and 27 sec for player 2 (9 states)[3], with ε (convergence criterion) set to 0.0001.

In both cases we prefer a gamma=0.3 which doesn't take into consideration so much the later states, but still not "bad" as there are max 4 states to go for the agent.

In player's 1 case agent chooses to check or fold with low value hands like J ,T or Q in round 1, and chooses to bet or call a raise in case of a strong hand like A or K. At round 2 agent bets or calls when having a pair or chooses to check or fold without one. This strategy works good playing against a tight player like the Threshold but gives small rewards because the agent chooses to play tight too to not lose. This policy is expected as a player 1 because the agent does not know anything about the hands of player 2 without him acting first. In most cases, the game ends at round 1, giving a reward of 0.5 or a draw at round 2 giving 0 reward to both players. In 10000 games agent won 4364 times, 3777 of them with 0.5 reward and 1324 times there was a draw between the 2 players.

In player's 2 case at round 1 agent chooses to bet when player 1 checks with any card leading to player 1 folding. If player 1 bets agent will raise with a good hand like A or K or fold with a bad one. At round 2 agent will bet when opponent checks which leads to him folding. If player 1 bets he chooses to call with A and no pair or raise when he has a pair or folds when not having any pair neither an A. One weakness of player 2 is that sometimes he chooses to raise at round 1 with low value hands like Q instead of folding and sometimes also chooses to raise at round 2 without having a pair. Again, most of the time the game ends at preflop because player 1 folds with T, J, Q giving a reward of 0.5 to the agent. In 10000 games agent wins 7915 times, 6069 of them with a reward of 0.5 (0.76%).

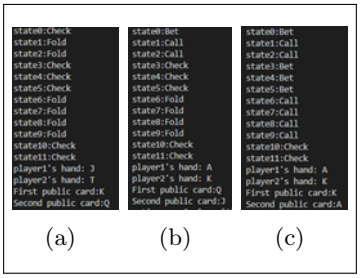


Figure 7: Policy of P.I. Agent as player 1 vs. a threshold agent — states 0-1 are round 1, states 2-9 are round 2 and states 10-11 are terminating

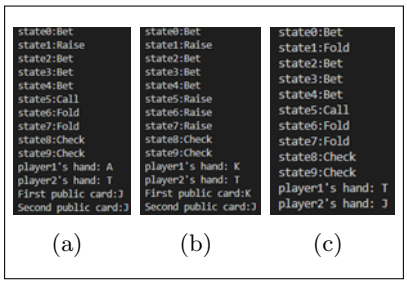


Figure 8: Policy of P.I. Agent as player 2 vs. a threshold agent — states 0-1 are round 1, states 2-9 are round 2 and states 10-11 are terminating

2 Q-Learning

2.1 Environment

The action space for the Q-Learning implementation is an augmented version of the one used in the Policy Iteration implementation (see fig. 1). Every state consists of:

- The player's hand
- The public cards
- The player's turn (player 1 or player 2)
- The current round
- The number of bets the player has made
- The number of bets the opponent has made

As a result the state space was quite big (taking into consideration all the possible permutations of the above, there are about 100.000 states), however anything less resulted in sub-optimal results[4].

The rest of the environment (the action space, the dealer, the judger and the opponents) is identical to the P.I. one.

[4]initially the cards were not included but the agent failed to learn efficiently, resulting in policies that barely incurred any reward

2.2 Algorithm

The Q-Learning agent's main component is the Q table (implemented as a python dictionary, since the keys needed to be states). One can visualize its rows as the states and its columns as the actions, each cell (s, a) containing the Q value of taking action a at state s. Initially all Q values are set to 0, and as the agent is trained they converge to their final values. Once every value has converged (after enough iterations), an optimal policy is derived from the Q table.

During training, the agent's policy is an ε-greedy one: when in state s, we perform a random action with probability ε and the current best action ($\text{argmax}(Q(s, a))$) with probability 1-ε. This aims to allow for some exploration of unknown states, as one of the conditions for Q-learning's convergence is that every state is visited, which would not be possible without exploration. The value of ε (and thus the probability of taking a random action) decreases with the passage of time to allow for more exploration in the first stages of training while ensuring that the actions that seem better are also explored more. The formula for the value of ε at episode t is $\varepsilon = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min}) * e^{-0.0005 \cdot t}$, where ε_{min} and (ε_{max} are customizable parameters. Their values were found by experimentation (just like the one of the exponential decay rate, 0.0005).

Once an action is chosen, the agent interacts with the environment which returns a potential reward, the next state and whether the game is over or continues. Once the reward is returned the agent updates its Q table using the standard temporal difference (TD) Q-Learning formula:

$$Q(s, a) = Q(s, a) + \alpha[r(s, a) + \gamma \max_{a'}\{Q(s', a')\} - Q(s, a)]$$

where s' is the next state (returned by the environment),
a' is any legal action in s',
$r(s, a)$ is the reward that performing action a at state s incurred,
α is the learning rate,
and γ is the discount factor.

The learning rate α and the discount factor γ are parameters of the algorithms that we can use to tune its performance. Their values can differ based on the environment configurations, and the values we chose are discussed in more detail below (section 2.3).

A part of the environment is also an opponent whose actions the agent trains against and is thus called the *trainer*. Experimentation showed that the agent learned quicker and better when trained by a threshold agent rather than a random one, which is also visible in the performance plots below section 2.3. This can be explained by the fact that the threshold trainer is a more formidable opponent that takes into consideration the state of the game and will thus be harder to beat. Therefore decisions that lead to victory versus the threshold opponent will usually work against a random one as well, while actions that beat a random opponent will most probably not work against a threshold one.

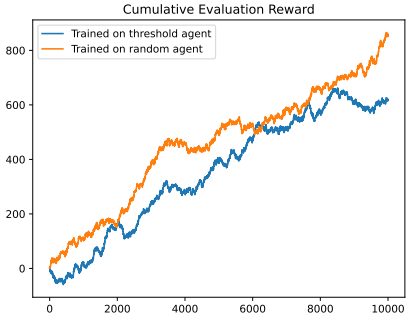
2.3 Performance

To test the performance of the Q-Learning agent we run 10.000 evaluation episodes (i.e. games) where it according to the policy it learned during training. The following experiments were conducted after 10.000 episodes of training, which were found to be enough for the policy to converge to the best possible value. Experiments were made with up to 10.000.000 training episodes, with no noticeable difference in convergence speed, resulting policy or performance against an opponent, so the 10.000 training episodes were deemed enough.

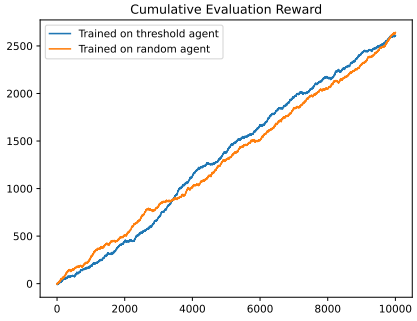
2.3.1 Against a Random Agent

When playing against a random agent the best results were observed with the following parameter configuration:

- the learning rate α has a constant value of around 0.1
- the discount factor γ has a constant value of around 0.5-0.75



(a) Q-Learning Agent plays first — average reward is 0.08



(b) Q-Learning Agent plays second — average reward is 0.25

Figure 9: Q-Learning Agent performance vs. a random opponent

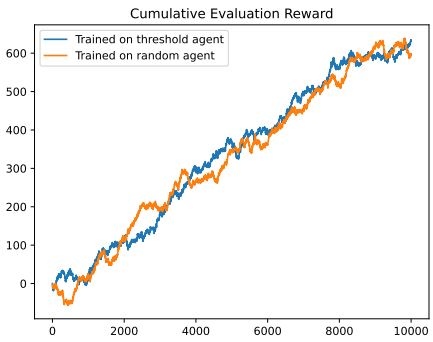
The results of the experiment can be seen in fig. 9. We observe that the performance is significantly better when the agent plays second, which is to be expected since it has more information about its opponent, leading it to more "educated" decisions.

We also observe that the rewards that the agent manages to collect over the course of the 10.000 games are less than those that its P.I. counterpart collected, deeming the Q-Learning agent decent yet sub-optimal. This could perhaps be explained by the fact that the learning rate is constant, making the algorithm vulnerable to under-exploration of potentially profitable actions because of a bad start, or by a premature convergence to a sub-optimal policy. However, as mentioned above, that effect did not seem to go away even with up to 10.000.000 training episodes, so further optimizing the algorithm would be very costly.

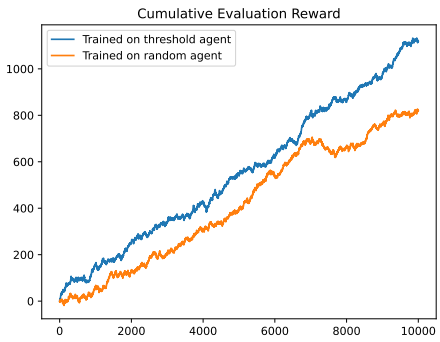
2.3.2 Against a Threshold Agent

When playing against a random agent the best results were observed with the following parameter configuration:

- the learning rate α has a constant value of around 0.1
- the discount factor γ has a constant value of around 0.5-0.75



(a) Q-Learning Agent plays first — average reward is 0.06



(b) Q-Learning Agent plays second — average reward is 0.12

Figure 10: Q-Learning Agent performance vs. a threshold opponent

We observe fig. 10 that the rewards accumulated by the Q-Learning agent against the threshold opponent match the ones that the P.I. one achieved (see fig. 6), deeming them both **optimal** in that particular setting.

The fact that the Q-Learning agent performs almost as well against both random and threshold agents can be explained by the fact that it was trained against the latter and is thus more "tight" in his choices, because it has *learned* that when an opponent e.g. raises, it most probably means negative reward. As a result when the random opponent randomly raises, the agent, having no knowledge of the opponents hand, assumes that it would lose and thus folds. This handicap does not come from overfitting to the threshold opponent that the agent trained with, but rather results from the very nature of the game — it is (unbeknownst to the random agent performing it) a *bluff*.