Leduc Holdem – Final Project

Doriah Adjedj
dept. of computer science
Hebrew University of Jerusalem
Jerusalem, Israel
doriah.adjedj@mail.huji.ac.il

Eli Chouatt
dept. of computer science
Hebrew University of Jerusalem
Jerusalem, Israel
eli.chouatt@mail.huji.ac.il

Samuel Tora
dept. of computer science
Hebrew University of Jerusalem
Jerusalem, Israel
samuel.tora@mail.huji.ac.il

"The analysis of a poker game should be quite an interisting affair."
John Nash, 1951

Contents

Abstract

If you have ever played poker, you certainly know the feeling of regret after making a decision that led to your loss. We then wonder if we made the right decision.

Many parameters can influence our decision: the choices that have been made by the players so far, the body language of our opponents, the community cards that are revealed as the game progresses and which strengthens or weakens our hand …

Professional players use all this information to make their decisions as profitable as possible.
But is there only one winning strategy in poker?

Keywords—Poker, Leduc Hold'em, reinforcement learning, searching algorithm, Q-Learning, CFR, Minimax.

I. INTRODUCTION

A. How did we get to this project ?

Our first desire was to find a winning strategy in the most popular variant of the card game of Poker: Texas Hold'em No Limit.

After some research we realized that it was impossible for us to do so with the time available and our computing power. Indeed, in the case of a face to face in poker, there are 10^{161} possible game situations, which is more than estimated atoms in the universe.

We were therefore looking for a simplified poker model that would allow us to verify our results.

We first opted for Kuhn Poker but this one no longer justified the use of AI because of its low complexity.

So, we finally decided to use the Leduc Poker model which is a simplified version of the Texas Hold'em.

The Leduc Hold'em version seeks to retain the strategic elements of the large game while keeping the size of the game trackable.

B. Complexity of the model

The version we will be working on has about 48 game possibility trees. We get that Leduc Hold'em is a lot more complex than Kuhn poker (288 vs 12 information sets), for most information sets we can no longer look at the action probabilities and deduce whether they are correct.

As we can see on Figure 1, there are 6 information sets on one graph which represents a single possibility tree for a specified distribution (Player 1 got a Jack, Player 2 got a Queen, the Community card is a King, and the small blind is Player 1). Knowing that there are 48 game possibility trees (depending on the card distributions and who is the small blind and the big blind), we then obtain 288 information sets for the Leduc Hold'em game.

Using Artificial Intelligence our programs find the optimal strategy empirically for every information sets much faster than a classic programming technique which will have to take into consideration all the possibilities for every information set.

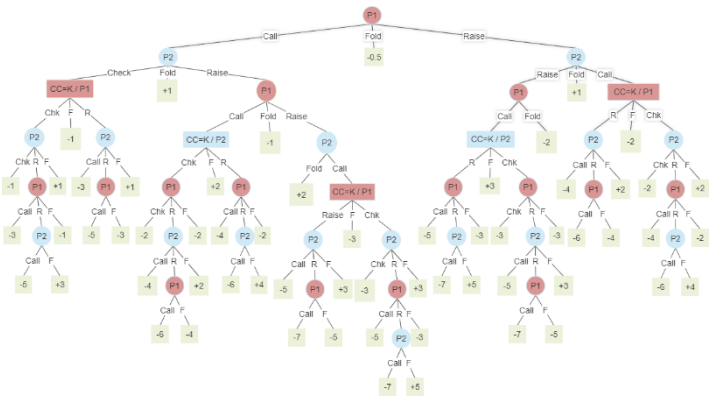

Fig. 1. Possibility Tree for a single distribution

C. Rules of the game

We focus on a two-player version of the game. It is played with 6 cards (Jack, Queen, and King of Spades ♠, and Jack, Queen, and King of Hearts ♥). At the beginning of the game, each player is dealt one card. There are two rounds and at the second-round a card is placed in plain sight – the Community card.

Playing the game :

- First Round

One player is randomly chosen to put one chip as small blind , the other player put two chips as big blind.
Start the betting part, it is a two-bet maximum (it means that each player can raise once) and a raise amount of 2.
The small blind acts first, he can call (= bet as much as the opponent), raise (= bet 2 more than the opponent), or fold (= give up, the other player wins).

- Second Round

A public card is revealed, the community card.
It is a two-bet maximum and a raise amount of 4.
The player in front of (or next to) the last player to have spoken acts first, he can check (= pass but is still in the game), raise (= bet 4 more than the opponent), or fold (= give up, the other player wins).

The player whose hand has the same rank as the community card (means he has a pair) is the winner. If neither, then the player with the higher rank wins. Like Texas Hold'em, high-rank cards trump low-rank cards (e.g., Queen is larger than Jack). A pair trumps a single card (e.g., a pair of Jack is larger than a Queen and a King).

The goal of the game is to win as many chips as you can from the other player.

D. Challenges of the project

Leduc Hold'em is a sequential decision problem which means that the agent's utility depends on a sequence of decisions. Furthermore, it is an imperfect information game due to the opponent's private cards, which make it a partially observable environment.

We will therefore have to face the following challenges :

- Imperfect Information Game: We cannot use game state because the information in the game model is imperfect. We will therefore have to work with sets of information in which all the information will be grouped except the card of the opponent which is unknown.
- High Variance of the payoffs (we can count on luck): An optimal strategy can still lose in a sufficiently small number of games due to the variance. It makes it difficult for a program to evaluate its performance over short periods of time, therefore, to verify the viability of a strategy we must test it in a sufficient number of games to apply the law of large numbers.
- Sequential Game: Our actions may not immediately lead to a payoff. So how can we assess whether an action is good or bad? In our algorithms, we will adapt to this problem.
- Use of the CFR algorithm: Algorithm we did not see during the course, we learned to understand it and use it for this project.

II. APPROCHES

In this project, we chose to use RLcards - a toolkit for Reinforcement Learning in cards game, that supports multiple card environments with easy-to-way interfaces for implementing various reinforcement learning and searching algorithms[1].

A. Reinforcement Learning (RL)

The purpose of RL is for the agent to learn an optimal, or nearly optimal, policy that maximizes the "reward function". A basic RL agent interacts with its environment in discrete time steps. At each time t, the agent receives the current state s_t, and a reward r_t. It then chooses the action a_t from the set of legal actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward associated with the transition (s_t, a_t, s_{t+1}) is determined.

When the agent's performance is compared to that of an agent that acts optimally, the difference in performance gives rise of the notion of regret. In order to act near optimally, the agent, must reason about the long-term consequences of its actions, although the immediate reward associated with might be negative.

In the case of Leduc Hold'em, the agent only has access to a subset of states (partial observability). Then, we represent it as a POMDP (Partially Observable Markov Deterministic Process) where the states are the information sets: they consist of the actions taken by the two players so far, the raveled cards (if there is), and the player's own private cards. We defined the reward of the agent to be the number of chips that the loser bet divided by 2 (negative if the loser our agent).

We will focus on two different reinforcement learning agents: Q-Learning agent and CFR agent.

i. Q-Learning Agent.

Q-Learning is a temporal learning agent: an unsupervised technique in which the learning agent learns to predict the expected value of a variable occurring at the end of a sequence of states[2]. RL extends this technique by allowing the learned state-values to guide actions which subsequently change the environment state.

To do this task, the agent has a function that calculates the quality of a state-action combination:

$$Q : S \times A \to \mathbb{R}$$

Q is initialized with 0 as default, and at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1}, and Q is updated by the value update iteration:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{(1)} + \underbrace{\alpha}_{(2)} \cdot \overbrace{\left(\underbrace{\underbrace{r_t}_{(3)} + \underbrace{\gamma}_{(4)} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{(5)}}_{(6)} - \underbrace{Q(s_t, a_t)}_{(1)} \right)}^{(7)}$$

(1) current value
(2) learning rate
(3) reward
(4) discount factor
(5) estimate of optimal future value
(6) new value (temporal difference target)
(7) temporal difference

The function approximates the value of selecting a certain action at a certain state. An episode of the algorithm ends when s_{t+1} is a terminal state[3].
The QLA takes three hyperparameters:

1. $\alpha -$ Learning rate: determines to what extent newly acquired information overrides old information.

2. $\gamma -$ Discount: determines the importance of future rewards.

3. $\epsilon -$ Exploration: Q-Learning is an ϵ-greedy algorithm, which means that with a probability of ϵ, we select an action

randomly during the train step (select the optimal action otherwise).

We thought about receiving good results using this method because Leduc Hold'em reduce the information state space of the Poker to an order realizable to learn from.

ii. Conterfactuel Regret Minimization (CFR)

CFR is a self-play algorithm: it learns to play a game by repeatedly playing against itself. The program starts off with a strategy that is uniformly random, where it will play every action at every decision point with an equal probability.

It then simulates playing games against itself. After every game, it revisits its decisions, and finds ways to improve its strategy. It repeats this process for billions of games, improving its strategy each time.

Let us explain some concepts that CFR use to compute his strategy.

- *Regret Matching:*

Regret is the loss in utility an algorithm suffers for not having selected the single best deterministic strategy.

Mathematically speaking, the regret is expressed as the difference between the payoff of a possible action and the payoff of the action that has been taken. If we denote the payoff function as *u* the formula becomes:

$$Regret = u_{possible\ action} - u_{action\ taken}$$

Clearly, we are interested in cases where the payoff of '*possible action*' outperforms the payoff of the '*action taken*', so we consider positive regrets and ignore zero and negative regrets.

The idea of regret matching is to choose actions with larger regrets more often in future iterations because they seem to be more profitable. Because we need a probability distribution over the actions, the algorithm will choose actions according to their cumulative regrets, normalized so they sum to one.

- *Counterfactual:*

Counterfactual means that we are going to look at the scenarios we would have arrived at with a different action from the one we have chosen.

- *Reach probability of a game node:*

In Poker like most of the sequential games, we cannot be sure to reach a certain game node because we can influence our play but not our opponent's play.

This introduces the concept of reach probability. The reach probability of a game node is defined by the probability of getting to this node using our strategy and our opponent strategy.

- *Counterfactual utility of a game state:*

The counterfactual utility of a game state h using a strategy σ is defined by the sum of: all the utilities of the leaf nodes that can be reached from this game state h, multiplied by the strategy of reaching them using our strategy from the game state h, multiplied by the probability of reaching them using a strategy that will always take actions that lead to them when it's possible.

More formally when z are the leaf nodes [7]:

$$v_i(\sigma, h) = \sum_z \pi_{-i}(\sigma, h)\pi(\sigma, h, z)u_i(z)$$

- *Counterfactual regret:*

The counterfactual regret of not having chosen a specific action in a game state h will be defined as the counterfactual utility of this game node if we choose this specific action minus the counterfactual utility of this game node using our strategy.

$$r(h, a) = v_i(\sigma_{I\to a}, h) - v_i(\sigma, h)$$

- *Cumulative regret:*

As we iterate over the graph, we add new regrets to the previous one, that is called the cumulative regrets. Like in regret matching we are only interested in the positive cumulative regrets to compute a probability. If they are negative, we set them to 0.

The positive cumulative regret of an action a of an information set I in iteration T is defined as follows:

$$R^{T,+}(I, a) = \max\left(0, \sum_{t=1}^{T} r^t(I, a)\right)$$

- *Update the strategy:*

The way we adapt our strategy over iterations is very similar to the way we do in regret matching. If the sum of all the positive cumulative regret of an information set I is 0 we take an action randomly. Else, we define the probability of taking an action a proportionally to their positive cumulative regrets.

This conducts us to the following definition of the strategy , in iteration T+1, of the player i, in the information set I if we take the action a: [7]

$$\sigma_i^{T+1}(I, a) =$$

$$\begin{cases} \dfrac{R^{T,+}(I, a)}{\sum_{a\in A(I)} R^{T,+}(I, a)} & if \sum_{a\in A(I)} R^{T,+}(I, a) > 0 \\ \dfrac{1}{|A(I)|} & otherwise \end{cases}$$

where |A(I)| is the number of possible actions for an information set I.

- *How CFR works*

The idea is the following:

When we need to decide the action to take, we use regret matching to compute the strategy.

We compute the payoff under this strategy as well as the expected payoff when taking a different action.

Theses payoffs are passed back up to the parent node.

We weight the payoff by how likely reaching each of the nodes.

- *Why we choose to present CFR*

In a two-player zero-sum game, if you compute the average strategy over those billions of strategies that CFR compute in his run, then that average strategy will converge towards a Nash equilibrium for the game.

B. *Expectimax Algorithm*

Leduc Hold'em is a zero-sum game, then we can determine the utility of any given state recursively, using search algorithms like Minimax. Expectiminimax is an extension of the earlier algorithm for incomplete information games: in addition of the min and max layers, we add intermediate 'chance' nodes, where we consider the different possibilities to substitute the hidden information, and we keep the average value of the children (assumption that at every step in the game, the probability of each card is uniform).

Because of the depth of the possibilities tree, we cannot travers all the tree at each choice. We need then to use a heuristic function to approximate the reward of a non-final node, because we will not necessarily reach a leaf.

The heuristic function we decide to use is a famous technic used by professional poker players, that consist of calculating the difference between the number of hands better and worse than our, and act according to the expected reward according to the earlier ratio.

III. Analysis

We now want to verify the effectiveness of the strategies returned by these different algorithms. To do so, we will confront them to three types of usual profiles in poker.

A. *Method*

i. Testing Agents

First, the random player. As its name suggests it, the player chooses a random action at each stage.

Second, the aggressive player. He will always choose the most aggressive way to play. So, he will choose his move according to this order: 1. raise, 2. bet, 3. check, 4. fold.

Third, the cautious player or tight player. He only plays his best cards (King or Queen in Leduc Hold'em for example).

ii. Choice of the Hyperparameters

We have already seen that QL has hypermeters that need to be given to the agent before starting training. To decides which values we will give, we decided to train the agents with some possibilities for each parameter:

α (*learning rate*)	0.001	0.01	0.1	0.2
γ (*discount*)	0.1	0.3	0.5	0.8
ε (*exploration*)	0.5	0.8	0.9	0.99

and, retain the best permutation for each test agent:
- Random: $\alpha = 0.001, \gamma = 0.3, \epsilon = 0.5$
- Aggressive: $\alpha = 0.001, \gamma = 0.5, \epsilon = 0.8$
- Cautious: $\alpha = 0.01, \gamma = 0.5, \epsilon = 0.8$

Each time we get a low learning rate, which can be explained because of the luck parameter, even for an optimal strategy, there are game situations where we cannot win. A low learning rate will make the agent learn over the long term.

About the discount factor, a higher learning rate against the aggressive and cautious agents can be interpreted by the fact that these strategies tend to make the game last until the final round.

Also, these two algorithms tend to play the same situations. That can explain a higher epsilon, to make our agent explore different situations (that may be better for him that one already discovered).

iii. Testing Methods

For the RL agents, to test the performances we decided to train the agents for a large number of episodes (sufficient to reach convergence), and we evaluated it every n episodes for 5000 more games (where n is the total number of episodes divided by 20) and plotted the progression curve. To evaluate the agents, we calculated the average reward (number of big blind won) on all the evaluation games.

Unlike RL agents, Expectiminimax does not need train period. Then, we decided to test it comparing the results for different depths and against the different testing agents on 10000 games.

B. *Results*

i. Presentation of the results

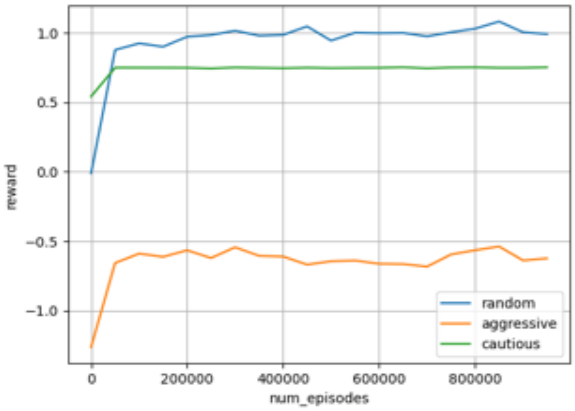

Fig. 2. The Q-learning Agent

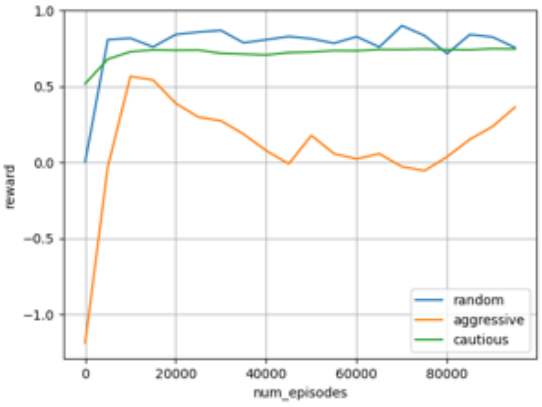

Fig. 3. The CFR Agent

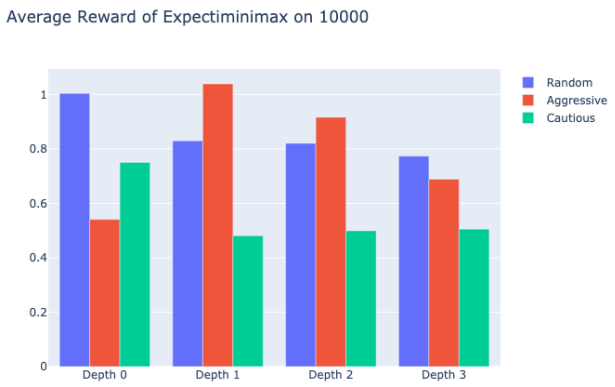

Fig. 4. The Expectiminimax Agent

C. Analysis of the results

i. Common Deductions on RL algorithms

First, the fact that the payoff against the random strategy is always positive suggests that the strategy learned by the RL algorithms follows a logic that corresponds to poker game theory.

The positive stability against the cautious player is explained by the fact that he only plays his best cards. Indeed, he loses his blind every time he receives a bad hand and win with high frequency when he plays his cards because these are the best. He therefore approaches a negative gain, because his strategy is clearly not optimal, with little variance because he does not take much risk and therefore plays very little with his luck.

By observing the curve of the aggressive player, we see that he wins at the beginning against our agents. This is explained by the fact that he will always try his luck even with low cards and therefore win whenever our agents fold at one step of the game or have worse cards at showdown.

ii. Differences between Q-learning and CFR

First notice that all the curves in Q-learning results are more stable than those in CFR. This can be explained by the fact that in Q-learning, the process that updates the Q table estimates does not include a random generator, so it is deterministic. Conversely, the strategy of CFR is based on regret matching, so it is probabilistic. It therefore seems logical that the variance is less felt in the results of Q-learning.

For these same reasons, Q-learning win more than CFR against the random player. When CFR will diversify its game and therefore sometimes choose a less good decision, Q-learning will always take the decision it considers the best. As a result, his gain will be better than the one of CFR against a player who does not follow any logic.

Regarding the aggressive agent, CFR responds much better over time than Q-learning.

iii. Deductions for Expectiminimax

Like for the RL agents, we get a positive average reward against the random agent which prove a coherent approach of the poker.

Furthermore, we can observe a convergence in the results after depth 1 against the random and the cautious agents. This phenomenon can be explained by the fact that, by the average done in the 'chance' nodes, deeper we traverse the tree, more the results are approximated. Plus, the heuristic used add to the approximation: this is no data remaining to exploitation.

Also, the results against aggressive have a remarkable improvement from depth 0 to 1. Indeed, at depth 0, we directly evaluate our state using the heuristic, without traversing the three. Then, Expectiminimax will only consider playing the good cards, folding the rest of the time (remember the strategy of cautious). And this strategy is exploitable by aggressive, which will try to raise no matter what. But, traversing the tree make our agent consider the situation where it is possible for him to win, even with lower cards.

The last point we wanted to analyze was the fact that we get lowest results against cautious agent. Like we said in the explanation of the algorithm, we assumed during the implementation, that the cards non revealed are distributed uniformly and are independent to the actions taken by the opponent, which is false against cautious agent that play only good cards.

IV. Conclusion and Discussion

According to the results obtained, CFR won against the three profiles. It would therefore seem that CFR is the agent who has come closest to a winning strategy in Poker. This is not surprising because as mentioned earlier, with CFR, the average strategy that it is computing converges towards a Nash equilibrium and a Nash equilibrium has an exploitability of zero, since it cannot be beaten by anyone on expectation.

It is possible to use these algorithms on the initial objective: the Texas Hold'em no limit but we would be confronted with a problem of complexity. A way around this problem would be abstraction. This is what was used in the case of the Libratus and Pluribus bots.

In 2015 researchers have announced that they have produced a strategy using CFR with such a low exploitability (0.000986 big blinds per game) that it would take more than a human lifetime of play, using the perfect counterstrategy, for anyone to have 95% statistical confidence to win against it. [5]

We can expect getting better results at Leduc Hold'em for Expectiminimax if we manage to approximate the real distribution of the unrevealed cards (until now assumed to be uniform), learning the strategy of an opponent by adding some reinforcement learning for example. However, during the analysis we saw that the pure strategy does not work well in incomplete information game. Furthermore, the approximations done at the 'chance' nodes and when using the heuristic function are amplified for Texas Hold'em version (more possible branches), harming the performance of the agent.

The Q-Learning agent used in this project has a Neural Network version, Deep Q-Learning (DQN). The earlier use of Deep learning makes the agent capable to manage complexity as Texas Holdem's[4]. But, as said before, a deterministic approach is not optimal for incomplete information game.

V. Code Usage

To run our project, you must have the RLcard package, that can be installed from PyCharm.

Then, you need to replace the content of **rlcard.games.leducholdem.game.**py by game_exptiminimax.py and the content of **rlcard.models.leducholdem_rule_models.py** by our leducholdem_rule_models.py.

To get the results used in this article, you need to run main with the different command lines:

1. Obtain parameters for QLA:

--algorithm **test_agent** --num_episode **num_episode** --num_games **num_eval_games**

2. Train and evaluate a RL agent:

--algorithm **algo** --num_episode **num_episode** --evaluate_every **evaluate_every** --num_games **num_eval_games** --log_dir **dir_name**

3. Evaluate expectimax:

--algorithm expectiminimax --num_games **num_eval_games**

Where the variables represent:

test_agent: agent used as opponent to test the performance of our agent {'random', 'V1', 'V2'}, where V1 is the aggressive agent, and V2 the cautious.

num_episodes: number of episodes used for the training.

num_eval_games: number of games during an evaluation step.

algo: which RL agent we want to train {'cfr', 'qla'}.

evaluate_every: number of episodes between two evaluation steps.

dir_name: directory name to stock the result plot.

VI. Code credit

1. The library RLcards: https://github.com/datamllab/rlcard
2. Project 4 from AI course (67842): http://www.cs.huji.ac.il/~ai/reinforcement/reinforcement.html

References

[1] https://rlcard.org

[2] https://web.stanford.edu/group/pdplab/pdphandbook/handbookch10.html

[3] https://fr.wikipedia.org/wiki/Q-learning

[4] https://www.deepmind.com/blog/deep-reinforcement-learning

[5] https://www.quora.com/What-is-an-intuitive-explanation-of-counterfactual-regret-minimization

[6] http://proceedings.mlr.press/v97/brown19b/brown19b.pdf

[7] http://cs.gettysburg.edu/~tneller/modelai/2013/cfr/cfr.pdf

[8] https://proceedings.neurips.cc/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf

[9] https://towardsdatascience.com/counterfactual-regret-minimization-ff4204bf4205

[10] https://www.aaai.org/AAAI22Papers/AAAI-1200.XuH.pdf