Solving Leduc Hold’em Counterfactual Regret Minimization

Christopher Wolff
Department of Computer Science
Stanford University
cw0@stanford.edu

Jerry Zhenbang Tan
Department of Computer Science
Stanford University
ztan035@stanford.edu

Abstract

Poker has been one of the benchmark challenges for artificial intelligence. Playing well requires handling uncertainty in chance outcomes, uncertainty in the opponents’ hands, and uncertainty in the opponents’ strategies. Furthermore, solution techniques for games with imperfect information have important real-world applications in areas such as negotiations, auctions, and cyber-security. In this paper, we aim to solve a simplified form of poker called Leduc Hold’em [1]. We show that we can use variants of counterfactual regret minimization (CFR) in order to find an approximate Nash equilibrium (NE) for the game. We evaluate our learned policies by computing their exploitability and measuring their performance against various baseline algorithms. In our experiments, we show that while our learned policies are not unexploitable, they demonstrate strong performance against the baselines in practice.

1 Introduction

Leduc Hold’em is a two-player, imperfect-information, zero-sum game. It is played with six cards: two Jacks, two Queens, and two Kings. Initially, players are randomly assigned to be the small blind and big blind, respectively. Each player is then dealt a card and posts a 1 chip and 2 chip ante, depending on the respective initial assignments. There will be two betting rounds, each with a maximum of two raise actions. So the possible actions are {raise, check, fold} if acting first in a round, {re-raise, call, fold} if faced with a raise, and {call, fold} if faced with a re-raise. The first betting round has an allowed raise size of two chips. Then, a community card is dealt, and the second betting round proceeds. In the second betting round, players are allowed to raise by four chips. Finally, the player who made their opponent fold or who has the best hand at the end of the second betting round wins. The strongest possible hand is a pair of Kings, followed by a pair of Queens, and a pair of Jacks. If neither player has a pair, the one with the higher-ranked card wins.

The game can be view as a tree, in which the nodes represent all information of a situation, including the action history, both players’ cards, and the public card. The branches of the tree correspond either to chance outcomes or actions of a player. An *information set* is a set of nodes that are indistinguishable for one player. That is, the nodes in an information set contain the same private information for one player, but possibly different hidden information, such as the card of the other player. We denote the set of all possible information sets $\mathcal{I}$ and set of all possible actions $\mathcal{A}$. A policy $\sigma_p(I)$ maps each information set I that player p could find themselves in to a probability distribution of actions that they plan to take when in I.

Our goal will be to find an NE of the game [2]. An NE is a set of strategies, in which neither player has an incentive to unilaterally deviate from their strategy. In other words, no player can improve their expected reward by changing to a different strategy. If we play an NE strategy, we are guaranteed a minimal expected reward. Due to the zero-sum nature of the game, this minimal expectation is exactly zero. However, if our opponent deviates from the equilibrium, our expectation could increase. In other words, if we play an NE strategy, we are guaranteed to win, and how much we win depends on the mistakes our opponent makes.

2 Related work

A well-known family of algorithms for finding approximate NEs in two-player zero-sum games is called CFR [3]. A variant of CFR has been used to defeat some of the top professional poker players in No-Limit Texas Hold'em (NLHE) [4]. Recently, it was shown that this approach can be combined with real-time search to beat top professionals at 6-player NLHE.

CFR is not the only family of iterative algorithms capable of solving large imperfect-information games. [5] showed that online convex optimization converges to a Nash equilibrium in $O(1/T)$ iterations, which is far better than CFR's theoretical bound. However, in practice, the fastest variants of CFR are substantially faster than the best first-order methods [6]. Moreover, CFR was shown to be more robust to the state- and action-space abstraction used in the game.

Vanilla CFR requires full traversals of the game tree, which is infeasible in large games. One method to circumvent this is Monte Carlo CFR (MCCFR), in which only a portion of the game tree is traversed on each iteration [7]. At each iteration, a subset of leaves is sampled, and the regret of each action node is calculated by the sampled regret divided by the probability of reaching that node to counter-act the bias introduced in the sampling process.

Finally, [6] shows that despite using universal function approximators, deep reinforcement learning algorithms' success in perfect information sequential decision making games does not readily extend to imperfect-information extensive-form games. They often do not converge to good policies [6], neither in theory nor in practice. In comparison, combining CFR with state- and action-space abstractions by deep neural networks yields state-of-the-art performance.

3 Methods

3.1 Counterfactual regret minimization

In order to find an approximate Nash equilibrium for Leduc Hold'em, we experiment with three variants of CFR: PureCFR, PureCFR+, and MCCFR. The basic idea behind all of these algorithms is that we repeatedly traverse the game tree, and keep track of our cumulative *counterfactual regret* for each action at every information set. That is, we learn a function $\mathcal{R} : \mathcal{I} \times \mathcal{A} \to \mathbb{R}$. Intuitively, the counterfactual regret of an action tells us how much we would have preferred to take that action over the one we actually took, in hindsight. More specifically, the counterfactual regret of an action is the difference between the expected value of that action and the expected value of the action we actually took.

Initially, we start with a uniformly random policy and $R(i, a) = 0 \; \forall \; i, a \in \mathcal{I} \times \mathcal{A}$. We then play against ourselves repeatedly. At every iteration, one of the two players is assigned to be the traverser. This is the player who's strategy is being updated. At all of the traverser's decision nodes, we sample an action according to the current policy, and then update R using the counterfactual regrets of the other actions. At the other player's decision nodes, we also sample an action, and then use it to update the *average policy*. The average policy is simply a counter that keeps track of how often we played each action at each information set.

Lastly, we update our policy $\sigma : \mathcal{S} \to \mathcal{A}$ after every iteration. This is done using *regret matching*. Each action is selected proportional to its positive regret across all previous self-play iterations. After training, the counts in the average policy are normalized and yield our final policy, which will hopefully be an approximate Nash equilibrium.

We also experiment with an extension to PureCFR, which we call PureCFR+. The main difference is that after every iteration, we reset all negative regrets to zero. Additionally, we compute the final policy as a weighted average of the policies at every time step, with a weighting factor of the respective time step t. Intuitively, resetting the regrets to zero helps to explore actions that have large negative regrets, and weighting the policies ensures that more recent policies have a greater contribution to the final policy. This extension is strongly inspired by CFR+ [8], which applies the same changes to regular CFR.

Finally, we also use a variant called MCCFR, which modifies PureCFR so that the traverser calculates an expected value of actions on the identified player's turn, rather than sampling just a single one.

3.2 Best response values

In order to evaluate our final policy analytically, we would like to compute its *exploitability*. But first, we need to define the notion of a *best response*. A best response to an opponent's policy always chooses the action with the highest expected value, given full clairvoyance of said policy.

To compute the best response, we pretend that our opponent's actions are part of the environment, and then use backwards induction to infer the value of each of our own actions. Implementing the traversal for the best response requires handling three scenarios. If it's our turn, we recursively compute the value of each action, and then choose the action with the highest one. If it's our opponent's turn, we again compute the value of each action, but compute the expected value under the policy that we want to evaluate.

Finally, we may encounter a leaf node. During traversal, we keep track of the *reach probabilities* for each hand that our opponent could have. The reach probability of a hand is the probability that our opponent reaches the current state with that hand. At the leaf nodes, the utility is the expected utility over all possible hands of our opponent, sampled from the reach probability distribution.

Once we have a best response, computing the exploitability is trivial. We simply look at the value of the best response at the root node. There is a minor implementation detail, however. The simulation environment we use does not allow us to have explicit control over chance nodes. This means that instead of computing the expected value at these nodes, we need to use an approximation. We use Monte Carlo sampling to do so. This means that we sample the chance node a fixed number of times, and take the average value as our approximation.

4 Experiments [1]

We use the RLCard framework [9] in order to simulate the Leduc Hold'em environment. We start by training three policies – one using PureCFR, one using PureCFR+, and one using MCCFR. For each, we use 35,000 games of self-play. Every 1,000 iterations, we evaluate the policy by computing the average reward against a baseline that chooses actions uniformly at random and two agents from the RLCard package – one trained using neural fictitious self-play (NFSP) [10] and one trained using vanilla CFR. Each evaluation runs 10,000 games in order to compute a low-variance estimate of the expected reward. Furthermore, we compute the exploitability of the policy, using Monte Carlo sampling with a branching factor of 50 at chance nodes instead of an explicit expectation. Figure 1 shows the results of this experiment.

In all three experiments, we can see that we quickly perform very well against a random policy. We also learn to beat the NFSP agent over time. While NFSP is guaranteed to converge to a Nash equilibrium over time, we find the NFSP policy in RLCard is sub-optimal, given that we clearly achieve a position expected reward against it. After exploring the source code, we found that the state representation it uses is lossy, as it does not include the complete action history and instead represents each information state using only the available card information and stack sizes. This means that there are distinct information sets that the NFSP agent cannot distinguish, which is likely a cause for its poor performance against us. In addition, our policy is on par with the CFR agent from RLCard, and achieves approximately zero expected reward per game. Finally, we observe that the exploitabilities of our policies decreases over time, but do not appear to approach zero. This is surprising, given that we expect to converge to a Nash equilibrium eventually. Despite this, we still see that we do well empirically against other agents.

We can see that the three variants of CFR exhibit similar performance characteristics, and their differences are likely attributable to the stochasticity in the training and evaluation procedures. This result is consistent with [7]; the update rules of the algorithms are extremely similar and only differ

[1] Our code is available at https://github.com/zhenbangt/aa228_final_project.

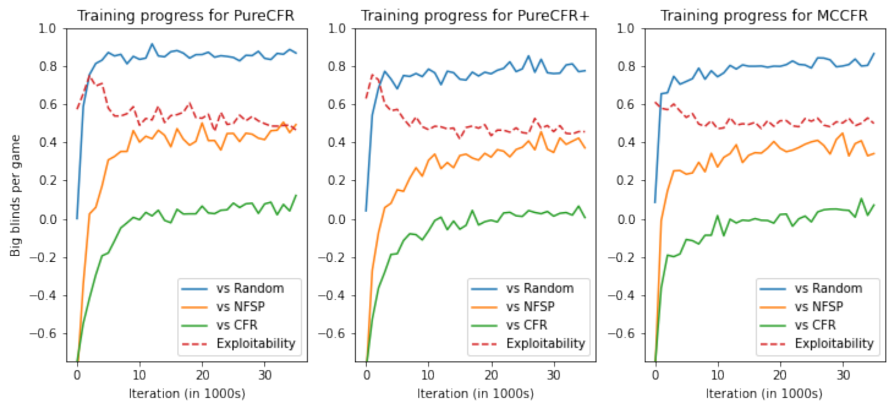

Figure 1: Training progress for CFR variants applied to Leduc Hold'em.

in how they sample actions and weight previous regrets. All of their regret estimates are provably unbiased, and should converges to the true regret values at approximately the same rate.

Next, we evaluate the performance of one of our final policies qualitatively. Namely, we evaluate the PureCFR+ policy. We do so by playing against the agent repeatedly using an interface that comes with RLCard. Despite having significant poker experience, we found it difficult to notice any imbalances. The agent appears to have learned many key concepts of poker, and found that it learned to bluff, value-bet, hero-call, and fold in various situations. The following are key behaviors that we noticed. The third behavior was surprising to us, given that we were only familiar with regular NLHE, where it's common for top players to fold frequently when facing raises before the public card appears.

1. Never call a bet on the river with a hand that's strictly dominated by all hands that the opponent could have. Similarly, when closing the action for a round and having the best possible hand, always raise if possible.
2. Balance your play so that you can show up with various types of hands in every situation. Balance your bets to include bluffs whenever you're value betting. This has the effect that the opponent can never put you on a narrow set of hands, and is often indifferent between two or more actions.
3. Call a lot before the public card appears. Hand strengths can change drastically depending on what the public card is, so you almost always have the correct odds to call when your opponent raises in the first round.

We illustrate this behavior partly by analyzing a specific line. Suppose we are in the small blind and both players are dealt a card. In the first round, we call, the opponent raises, and we call again. The public card is dealt and it's the K♠. We check, and the opponent raises. What should we do with each of our possible holdings? Figure 2 shows the learned action probabilities for each situation. We can see that when we hold the worst possible holding – a Jack of any suit – we usually fold. When we have the best possible hand – the remaining King – we almost always raise. Additionally, we bluff-raise whenever we have the Q♠, and sometimes when having the J♡. Interestingly, we rarely bluff-catch in this situation, and only do so occasionally with the J♠, which is unexpected given that to us, it makes more sense to call with a Queen than a Jack because it is a stronger hand. Perhaps the algorithm hasn't fully converged at this node, or maybe a pattern emerged that we cannot explain ourselves.

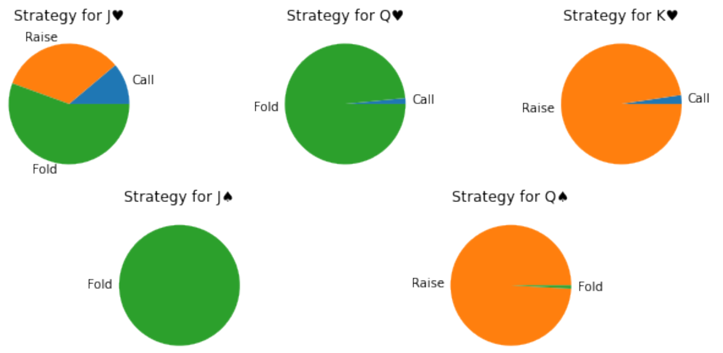

Figure 2: Our learned strategy for each possible holding in the line call:raise:call:K♠:check:raise.

5 Conclusions and future work

In summary, we show that we can learn a policy for Leduc Hold'em using three different algorithms: PureCFR, PureCFR+, and MCCFR. Each of the algorithms results in a strategy that performs well against various baselines, and reaches relatively low exploitability against a clairvoyant opponent. Across all metrics we measured, the three algorithms perform very similarly. Furthermore, we found that the empirical performance is strong even before the exploitability decreases significantly.

In the experiments, we found converging to zero exploitability is difficult. One hypothesis is the self-play procedure suffers from the chicken-and-egg problem: in early self-play iterations, both players are playing poorly and neither might make much progress. The agent may get stuck in a local optimum that is hard to escape from. This is reflected in both our results and [7] that CFR algorithms requires a significant number of iterations to converge. To speed up the training process, we might incorporate imperfection demonstrations [11] in the the self-play process. This might also open the possibility to transfer domain knowledge between similar poker games.

Finally, we are interested in exploring the development of exploitative strategies that can take advantage of an opponent's weaknesses. This could be done by first computing an NE strategy, and then applying a deviation that's based on the observed behavior of the opponent. Doing so requires a careful trade-off between exploitation and exploitability, as any deviation from an equilibrium strategy could leave us vulnerable to exploitation ourselves. To the best of our knowledge, a thorough analysis of this trade-off is still largely unexplored territory.

Contributions

Christopher worked on the PureCFR and PureCFR+ implementations and wrote code to compute the exploitability of a policy. He also created the two visualizations and wrote the Methods and Experiments sections of the report. Jerry implemented the MCCFR and wrote the abstraction, introduction, related works and future work sections.

References

[1] Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and D. Chris Rayner. Bayes' bluff: Opponent modelling in poker. *CoRR*, abs/1207.1411, 2012.

[2] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

[3] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems*, 2008.

[4] Noam Brown and Tuomas Sandholm. Libratus: The superhuman ai for no-limit poker. pages 5226–5228, 08 2017.

[5] Gabriele Farina, Christian Kroer, and Tuomas Sandholm. Online convex optimization for sequential decision processes and extensive-form games. *CoRR*, abs/1809.03075, 2018.

[6] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. *International Conference on Machine Learning*, 2019.

[7] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. *Advances in Neural Information Processing Systems*, 2009.

[8] Oskari Tammelin. Solving large imperfect information games using CFR+. *CoRR*, abs/1407.5042, 2014.

[9] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

[10] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016.

[11] Yang Gao, Huazhe Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell. Reinforcement learning from imperfect demonstrations. *CoRR*, abs/1802.05313, 2018.