PA026: Counterfactual Regret Minimization

Martin Horáček

May 28, 2021

1 Introduction

In games like chess and go, the player can observe the whole state of the game. These games are called perfect information games. The traditional technique of solving these games is minimax search that uses heuristic evaluation of nodes after some depth is reached. Stockfish, one of the best chess AI, uses this traditional approach. Another technique popularized in recent years by Deepmind and their AI AlphaZero is a Monte Carlo Tree Search (MCTS) combined with neural network for state value and policy approximation.

Games, in which a part of the state is hidden to the player, are called imperfect information games. They include games like poker, bridge, liar's dice. Many real world problems, such as negotiations and auctions, can be modeled using imperfect information games. The problem of finding optimal strategies to imperfect information games is NP-hard.

To see why games with hidden information are harder to solve than perfect information games, we will compare chess and poker. Consider two chess openings called Queen Gambit and Sicilian Defense. Optimal strategies in both openings are totally independent and do not influence each other. Compare that with having pocket aces preflop in poker. Since we have the best preflop hand, it would make sense to make a reasonable bet. But if we make this bet only with pocket aces, then our opponent would know our cards and they would act accordingly by folding. Therefore to play optimally, we have to balance our strategy over all hands we could have.

Small imperfect information games can be solved using Linear Programming. This approach is not feasible for bigger games. In recent years AI Libratus [2] and Pluribus [3] defeated top professional poker players. Both AIs used a version of Counterfactual Regret Minimization (CFR) algorithm as base of their strategy. Instead of trying to exploit mistakes of opponent, they approximate the unexploitable strategy called Nash equilibrium.

2 Algorithm description

CFR [7] is an iterative self-play algorithm. It learns provably optimal strategy. At the core of the algorithm lies the notion of counterfactual regret. The

counterfactual regret for an action A in an infoset I specifies how much we regret playing the current strategy compared to a strategy that is the same for all infosets except for infoset I, where it plays the action A with probability 1. The counterfactual regret for action A in infoset I is accumulated across all iterations. The strategy in current iteration is determined using Regret Matching. In regret matching the probability of playing action is proportional to the accumulated counterfactual regret of that action.

The total regret for a strategy is bounded by the sum of counterfactual regrets in all infosets. By iterative self-play using the regret matching, we minimize counterfactual regrets and therefore we minimize the total regret. It is proved that average strategy obtained by using this approach converges to approximate Nash equilibrium. By increasing the number of iterations, we get closer approximations.

More CFR algorithms were proposed over the years. These algorithms share same asymptotical bounds on regret, but they were shown to converge faster in practice. The original CFR algorithm is often referred to as Vanilla CFR.

The algorithm used to solve Heads-up Limit Texas Hold'em is called CFR+ [6]. It changes Vanilla CFR in a few ways. It resets negative accumulated regret to zero on each iteration and it uses weights for average strategy updates.

Because the average strategy is computed from all iterations, the initial random strategy has same weight as the strategy computed in last iteration. It makes sense to add more weight to recent iterations. This can be also done by multiplying accumulated values by number less than 1 each iteration. This approach was proposed in Linear and Discounted CFR [1] paper. Linear CFR scales both accumulated regret and accumulated average strategy linearly. Discounted CFR allows different weighting for positive regret, negative regret and average strategy.

Previous CFR algorithms always traversed whole game tree on each iteration. This becomes impossible with big games like No-Limit poker, where the game tree is too big to traverse even once. A method called Monte Carlo CFR [5] traverses only a part of the game tree on each iteration. It is proved that MCCFR converges to the Nash equilibrium. In the paper they proposed two sampling strategies. The external sampling traverses all actions for the player whose strategy is updated on current iteration and it samples chance and opponent's actions. This version of sampling was used in AIs Libratus and Pluribus. The outcome sampling samples all actions and therefore only one history is traversed on each iteration. One iteration is much faster, but it suffers from high variance of the update.

Another proposed approach called Deep CFR [4] trains a neural network that predicts counterfactual regrets. In usual CFR the regrets for infosets are usually stored in a hash table. Therefore any similarities between infosets cannot be captured and for big games we have to create abstractions of those games. For example in poker, similar card combinations are clustered together and treated as the same combination. Using a neural network, we allow the algorithm to learn these abstractions on its own.

3 Implementation

The implementation of CFR algorithms that I created in python3 can solve any extensive-form game specified by implementing the IState interface. These functions must be provided:

- is_chance() → bool: Returns true iff the chance (nature) decides action in this state.
- is_terminal() → bool: Returns true iff the state is terminal.
- get_player() → int: Returns integer id of the player that acts in this state
- get_infoset() → Infoset: Returns the view of the game for the acting player. It can be implemented as a string. All games that I implemented using IState interface use string as representation of Infoset.
- get_payoff(int id) → int: Returns payoff for player with 'id'. Can only be called on terminal state.
- get_available_actions() → List: Returns array of actions available in the state.
- next_state(action) → IState: Simulates the game rules and returns state after playing 'action'. If is_chance() is true, then it samples the chance action randomly.

I implemented four games using IState interface:

- Modified Rock-Paper-Scissors: When using extensive-form games, we have to use information sets if we want to simulate simultaneous action of both players. The game is modified by assigning any game, where scissors were played, value 2. This modification is used because in normal version of RPS the Nash equilibrium is same as the initial (random) strategy used in CFR. Therefore to see if the algorithm learns anything, I used this modified game.
- E-Card: I saw this game in movie Kaiji: The Ultimate Gambler and I wandered what is the optimal strategy. I implemented this game using IState interface and computed its Nash equilibrium using CFR algorithm. The optimal strategy is to pick random card from the cards in our hand.
- Kuhn poker: This is an extremely simplified version of poker with only 13 information sets. Analytically computed Nash equilibrium for this game is known and I used it to check that implemented CFR algorithms converge to it.
- Leduc poker: This is another simplified version of poker. It contains 288 information sets. The deck consists of 6 cards and there are two betting rounds. Complete rules are described in the docstring of LeducPokerState class. During algorithm evaluation I used also Leduc poker with 20 cards.

I implemented CFR algorithms described in section 2. To implement them I followed the mathematical description of algorithms in their original papers. Each algorithm has its corresponding Trainer class. All trainer classes follow same interface. By calling function train(iterations: int), the trainer runs corresponding CFR algorithm for 'iterations' iterations. This allows us to stop after some number of iterations, then check whether the strategy is improving and then continue with training.

Class CFRUtility contains helper functions that can evaluate strategies and simulate games.

3.1 Deep CFR

I tried to implement Deep CFR using PyTorch. I did not manage to make it work. Usually, after several hundreds of iterations a NaN values appear between weights of the neural network. I checked whether the input is correct and it was. I found that the main reason for NaN values between weights is the problem of exploding gradients. I added gradient clipping and it fixed problems with NaN weights. Unfortunately, the learned strategy did not converge to Nash equilibrium. There are many hyperparameters and options in Deep CFR and I think that it would require much more experimenting to make it work.

The incomplete implementation of Deep CFR is in "Leduc Deep CFR.ipynb" Jupyter notebook.

3.2 Examples

I created several Jupyter notebooks to show examples of how the implementation can be used. In first notebook I train CFR+ strategy for E-Card game. In second notebook I train DiscountedCFR for Leduc poker and it is possible to play against the trained strategy directly in the Jupyter notebook. The third notebook contains script that was used for evaluation of algorithms. The fourth notebook is the same as third notebook, but it uses bigger game (Leduc poker with 20 cards).

3.3 Installation

My implementation uses NumPy for numerical computations and matplotlib for plotting of graphs. I used PyTorch for neural networks.

All packages can be installed through pip using the requirements.txt file.

4 Evaluation

There are two standard ways of evaluating strategies for imperfect-information games. The first practical way is to compare a strategy to other strategies. By playing many games and computing sample mean, we can determine whether one strategy is significantly better than other. The other more theoretical approach is to compute exploitability. The exploitability measures how well the

strategy plays against the best response strategy. The best response strategy is a strategy that achieves highest expected value against our strategy. We can see it as if the opponent knew precisely what our strategy is and using this knowledge he optimized his strategy. For example if we would play rock with probability 1 in rock-paper-scissors game, then best response is to always play paper. Nash equilibrium strategy has exploitability 0.

High exploitability was a problem of previous poker AIs that could beat human players in the beginning, but started losing when human players found out how to exploit AI weakness.

4.1 Exploitability

To compute exploitability I used the fact that best response can be pure strategy. As opposed to equilibrium strategy that has to be mixed strategy. The best response computation recursively computes best actions for each information set by weighting expected values of states in given information set by their reach probabilities. I compared my exploitability computation with exploitability computed by CFR with one strategy fixed and they had same results. My exploitability computation doesn't have any parameters and is faster than computing best response using CFR.

4.1.1 Leduc poker

I trained all algorithms for one hour. All algorithms except OutcomeMCCFR converged to a strategy with low exploitability. OutcomeMCCFR exploitability was 0.75. In figure 2 we can see how the exploitability decreased over time. In figure 1 we can see that the biggest decrease in exploitability occured in first ten minutes of training. The exploitability of random strategy is 4.75. I did not include OutcomeMCCFR in the graph as it would make it harder to read.

The Vanilla CFR algorithm converged to the lowest exploitability.

4.1.2 Leduc poker with 20 cards

Because the Leduc poker with 6 cards is quite small game, I decided to compute exploitability for Leduc poker that has 20 cards. I trained all algorithms for 120 minutes. In figure 3 we can see that ExternalMCCFR converged to lowest exploitability and much faster than other algorithms. The reason is that ExternalMCCFR traverses game tree much faster, because it samples chance and opponent's actions. Therefore it doesn't traverse bad actions of opponent.

In figure 4 we can see exploitability in first 10 minutes of training. In the beginning the exploitability of VanillaCFR decreases slower than for CFR+, LinearCFR and DiscountedCFR. CFR+, LinearCFR and DiscountedCFR all give higher weight to recent strategy updates. Therefore they faster forget older regrets and can improve faster at least in the beginning. If the game was much bigger, then the exploitability graph for whole training could look as first 2

Figure 1: Leduc poker exploitability

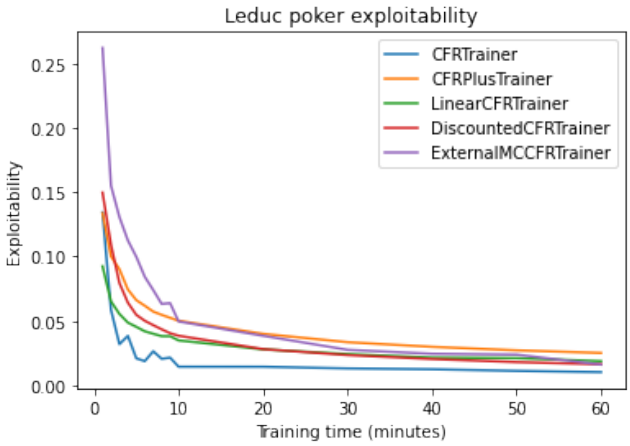
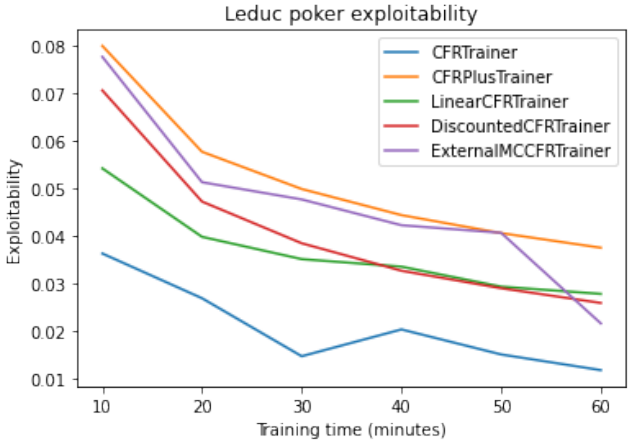


Figure 2: Leduc poker exploitability (from minute 10)



minutes for Leduc poker with 20 cards and VanillaCFR would be outperformed by other algorithms.

4.2 1v1 comparison

I simulated 250,000 games of Leduc poker between all trained algorithms. All algorithms significantly won against OutcomeMCCFR. There was no clear winner between remaining algorithms.

Figure 3: Leduc poker with 20 cards exploitability

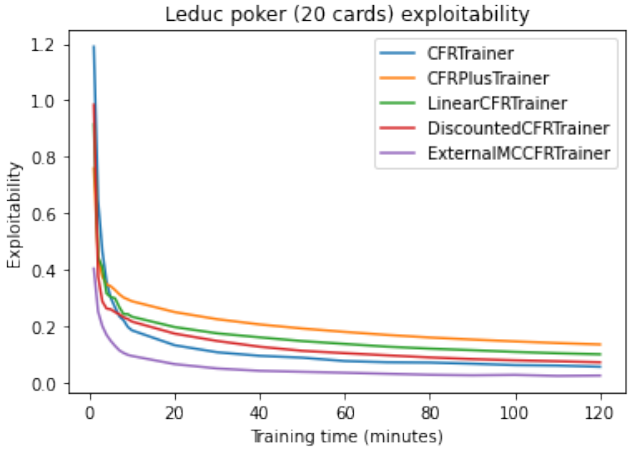
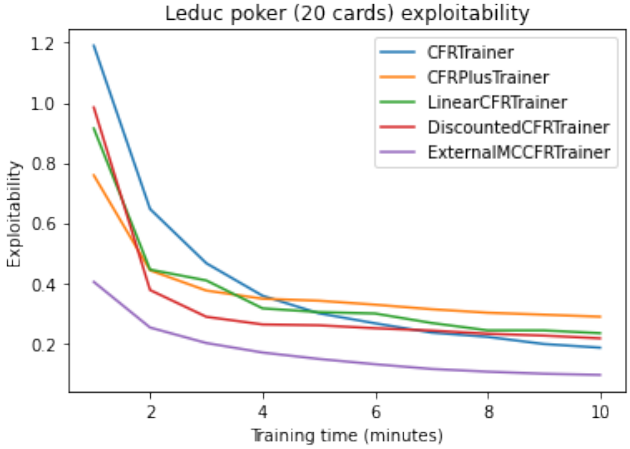


Figure 4: Leduc poker with 20 cards exploitability (first 10 minutes)



4.3 Evaluation of simple strategies

To compare clever strategies found by CFR to something simple, I used three simple strategies: random strategy, always call strategy and check-fold strategy. Check-fold strategy checks whenever it can and when it faces a bet, it folds. The table 1 shows exploitabilities of these simple strategies and their result against Vanilla CFR for Leduc poker. We can see that Vanilla CFR confidently beats simple strategies.

	Exploitability	vs. Vanilla CFR
Random	4.75	-1.45
Always call	2.93	-1.36
Check-fold	2.0	-1.13

Table 1: Evaluation of simple strategies

5 Conclusion

I studied Counterfactual Regret Minimization algorithm and its variants. I read papers that introduced these algorithms. I implemented many variants of CFR in python3.6. They are implemented in a way that allows us to easily define a game and run them to find its Nash equilibrium. I compared implemented CFR algorithms on Leduc poker and on Leduc poker with 20 cards. I also briefly commented on reasons why some CFR versions outperformed others.

6 References

[1] Noam Brown and Tuomas Sandholm. *Solving Imperfect-Information Games via Discounted Regret Minimization*. 2019. arXiv: 1809.04040 [cs.GT].

[2] Noam Brown and Tuomas Sandholm. "Superhuman AI for heads-up no-limit poker: Libratus beats top professionals". In: *Science* 359.6374 (2018), pp. 418–424. ISSN: 0036-8075. DOI: 10.1126/science.aao1733. eprint: https://science.sciencemag.org/content/359/6374/418.full.pdf. URL: https://science.sciencemag.org/content/359/6374/418.

[3] Noam Brown and Tuomas Sandholm. "Superhuman AI for multiplayer poker". In: *Science* 365.6456 (2019), pp. 885–890. ISSN: 0036-8075. DOI: 10.1126/science.aay2400. eprint: https://science.sciencemag.org/content/365/6456/885.full.pdf. URL: https://science.sciencemag.org/content/365/6456/885.

[4] Noam Brown et al. *Deep Counterfactual Regret Minimization*. 2019. arXiv: 1811.00164 [cs.AI].

[5] Marc Lanctot et al. "Monte Carlo Sampling for Regret Minimization in Extensive Games". In: *Advances in Neural Information Processing Systems*. Ed. by Y. Bengio et al. Vol. 22. Curran Associates, Inc., 2009. URL: https://proceedings.neurips.cc/paper/2009/file/00411460f7c92d2124a67ea0f4cb5f85-Paper.pdf.

[6] Oskari Tammelin et al. "Solving Heads-up Limit Texas Hold'em". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 645–652. ISBN: 9781577357384.

[7] Martin Zinkevich et al. “Regret Minimization in Games with Incomplete Information”. In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS’07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, pp. 1729–1736. ISBN: 9781605603520.