Yannou Ravoet r0637112
Robrecht Peeters r0627461

Machine Learning Project

REPORT 2

1 Problem Statement

For the machine learning project course, we were asked to design agents for two imperfect-information, zero-sum, 2-player games: Kuhn and Leduc poker. Any learning algorithms could be used, so we were tasked with developing an intuitive understanding of the algorithm and any modifications could be added/changed to influence the success of the algorithm on the 2 poker games.

2 Research questions

To find the best learning algorithm for Leduc poker, we created two research questions.

1. Which base algorithm has the most potential to reach the lowest exploitability in Kuhn Poker?
2. What improvements can be made on the chosen base algorithm and how do they affect the exploitability?

We limited the first research question to Kuhn poker because of the smaller solution space, allowing us to make more algorithms (almost) converge. With a base algorithm, we mean an algorithm in its most default form (without any modifications). As an example, we consider $CFR+$ to be a modification of the base algorithm CFR. By limiting ourselves to Kuhn poker and base algorithms, we hoped to be able to look at multiple families of algorithms before deciding on which family we would research modifications of on Leduc poker.

For the second research questions we first ran the modifications on Kuhn Poker, again allowing us to make quicker comparisons. This way, we only had to implement the modifications that were worth implementing in Leduc poker.

3 Experiments

3.1 Finding a base algorithm to improve upon

To find a base algorithm on Kuhn poker, we decided to implement the following algorithms: Extensive-Form Fictitious Play (XFP) (the fictitious play version implemented by OpenSpiel), Counterfactual Regret Minimization (CFR) and Policy Gradient with Regret Loss (PG). As a metric to decide which algorithm was best, we used the exploitability and NashConv of the trained agents. The results are visible in Figure 1.

Yannou Ravoet r0637112
Robrecht Peeters r0627461

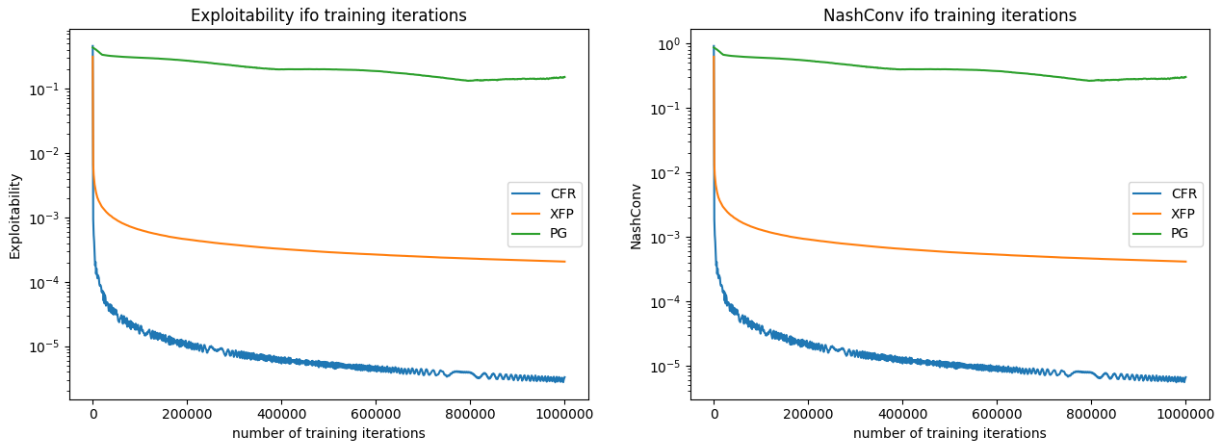


Figure 1: *Exploitability and NashConv of the base algorithms CFR, XFP and PG. We can clearly see that for Kuhn Poker, CFR has the most potential.*

The reason CFR performs better than XFP is that for each iteration, we do not only update the probabilities of the best response actions, but we update each action probability relative to its average regret. The reason both CFR and XFP perform better than $Policy\ Gradient$ (implemented with regret loss), is that CFR and XFP both try to find the value of actions in each state, whereas Policy Gradient tries to find the value of policies, which has a higher level of abstraction. For a game such as Kuhn poker, this abstraction is not necessary for performance reasons and thus only results in a more general and thus more exploitable policy.

Since some action probabilities seemed to converge towards certain constant values, we tested if rounding these probabilities towards these constants could further decrease the exploitability of a policy. Rounding values close to 0 or 1 to exactly 0 and 1 only very slightly decreased the exploitability. Rounding approximate values to exactly ⅓ and ⅔ however did decrease the exploitability to less than half what it was before. Unfortunately, this result did not translate to Leduc Poker as discussed in section 3.2.2.

3.2 Improving CFR with extensions

3.2.1 Testing on Kuhn Poker

We chose CFR as our base algorithm to improve upon. The modifications we implemented are $CFR+$ (as well as CFR with solely Regret Matching+, Linear averaging or alternating updates) and two versions of $Discounted\ CFR$: one with the proposed values of $\alpha = \frac{3}{2}, \beta = 0$ and $\gamma = 2$ – which were reported to perform well in the original paper [1]- as well as one with all three variables equal to 1 (also called $LCFR$). Each of these algorithms was trained on Kuhn Poker first and the best ones were selected for training on Leduc Poker.

Yannou Ravoet r0637112
Robrecht Peeters r0627461

We also considered a version of CFR where a Best Response policy is used as the opponent during self-play ($CFR - BR$). However,$CFR - BR$ was created to avoid CFR 'overfitting' to abstractions of games, leading to a higher exploitability in the unabstracted game, but not in the (abstracted) game it trains on [2]. Since we do not train on an abstraction of Kuhn or Leduc poker, this extension would result in a higher exploitability and would not offer an interesting comparison to the other extensions.

Intuitively, we know that later iterations of training are likely to contain better estimates of the optimal action-values. It then makes sense to give more weight to later iteration policies than to earlier policies. The family of Discounted CFR algorithms ($DCFR_{\alpha,\beta,\gamma}$) implements this weighting in two ways [1]:

- *Regret discounting on the cumulative regrets.*
  Parameters α and β are used to describe this behavior. At each iteration t, the accumulated positive regrets are multiplied by $\frac{t^\alpha}{t^\alpha + 1}$ whilst the accumulated negative regrets are multiplied by $\frac{t^\beta}{t^\beta + 1}$. Of course, the optimal values of α and β are game dependent but setting $\alpha = \frac{3}{2}$ and $\beta = 0$ has been reported to consistently outperform $CFR+$ and so these values were used. Combining $DCFR_{1,1,1}$ (also called $LCFR$) with $CFR+$ (adding Regret Matching +) has also been investigated but was found to diverge.

- *Linear averaging of the average policy*
  The parameter γ is used to average the average policy linearly over the iterations. At each iteration t, new contributions to the average strategy are multiplied by t^γ. As a result, linear averaging assigns more weight to more recent policies compared to the equal weight per iteration assigned by the default uniform average over all the previous policies. Linear averaging can thus be seen as a reverse discount factor: policies further in the past are giving less weight. A $\gamma = 2$ value is recommended by the original authors of $DCFR$, but this can be problem dependent.

Another flaw in CFR is that if an action suddenly becomes a good action (better than any other action) it can take a while before this change is represented in the policy updates, due to the action potentially having accumulated a lot of negative regret over the previous iterations. To prevent this, $CFR+$ extends CFR with the following modifications [3]:

- Regret Matching+ is used *instead of Regret Matching*
  Regret Matching updates action probabilities based on their accumulated regret. This means that any action having a negative regret, will have to return a better reward than the chosen action multiple times in a row to start being picked more often (when the regret gets positive). As a result, if the best action suddenly changes, Regret Matching must update multiple iterations before starting to increase the probability of choosing this best action. Regret Matching+ solves this issue by resetting any negative regrets to zero, meaning the best response action can immediately start being picked more often.

Yannou Ravoet r0637112
Robrecht Peeters r0627461

- Alternating updates are used instead *of simultaneous updates*
  Instead of updating the average policy of both players at the same time, alternating updates will first update the average policy of the current player and then use this update average policy to update the average policy of the second player.

- Linear averaging is used
  Linear averaging can sort of be a reverse discount factor: policies further in the past are giving less weight. Instead of using a uniform average over all the previous policies, linear averaging assigns more weight to more recent policies. More specifically for the OpenSpiel $CFR+$ implementation, the contributions of every iteration t are weighted proportionally to t^1. Which makes $CFR+$ equivalent to $DCFR_{\infty,-\infty,1}$.

In Figure 2 we plot the OpenSpiel implementation of the CFR, $CFR+$ and $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ algorithms. In Figure 3 we plot the result of adding each of the modification of $CFR+$ separately. Figure 4 shows us the difference between $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ and $DCFR_{\alpha=1,\beta=1,\gamma=1}$. Figure 2, 3 and 4 are all from training iterations on Kuhn Poker. In the case of Kuhn poker, we see that $DCFR_{\alpha=1,\beta=1,\gamma=1}$ is more stable and outperforms all the other algorithms. However, this result does not translate to Leduc poker as will be discussed in the next chapter. The reason $DCFR_{\alpha=1,\beta=1,\gamma=1}$ is much more stable than $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ is that it effectively weights each iteration t by $\frac{t}{T}$ where T is the total number of iterations ran instead of using different weighting schemes for positive and negative regrets.

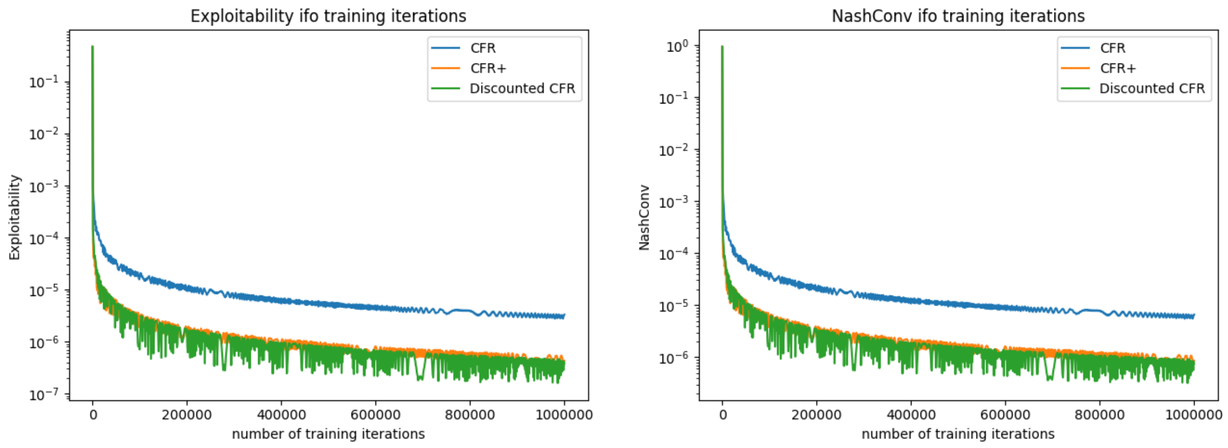


Figure 2: *OpenSpiel implementation results of CFR, CFR+ and* $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ *on Kuhn Poker.*

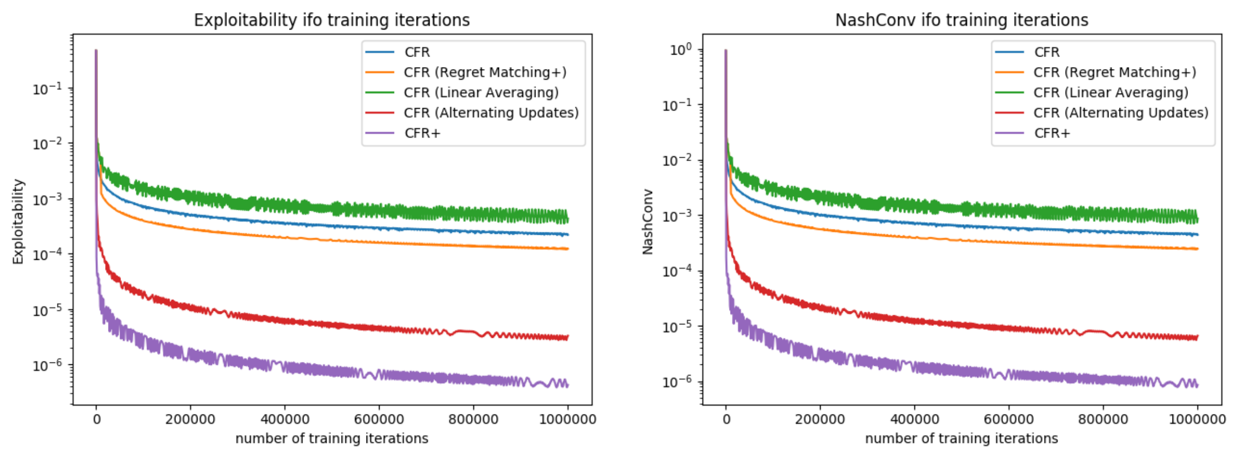


Figure 3: *The effect of the extensions of CFR+ on exploitability and NashConv shown separately. Although the reasoning behind alterning updates is never discussed in papers, the effect is clearly influencial.*

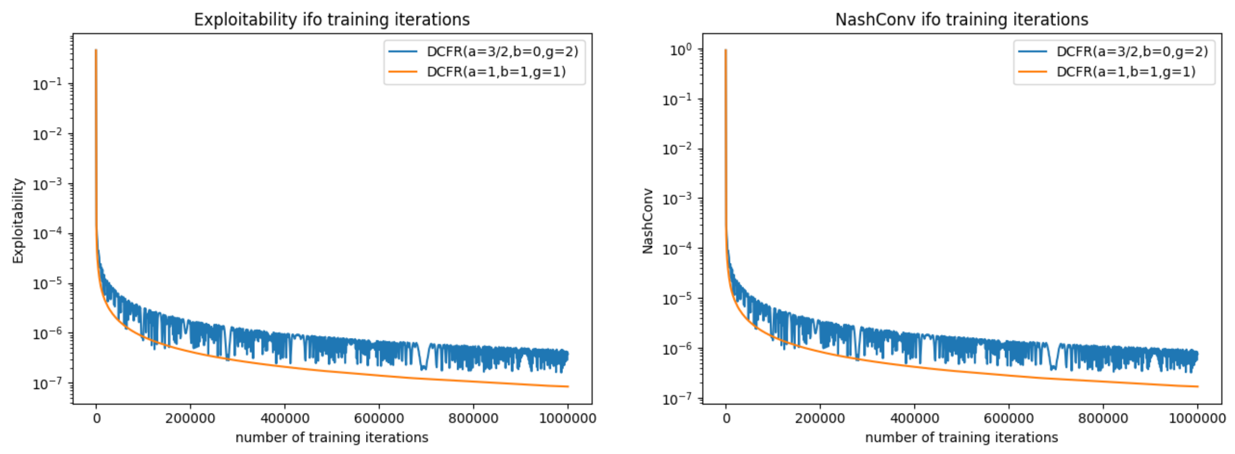
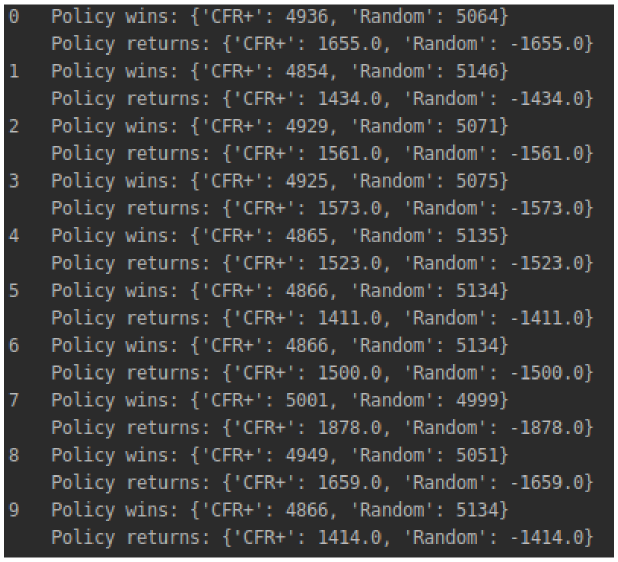


Figure 4: *Exploitability and NashConv of $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ and $DCFR_{\alpha=1,\beta=1,\gamma=1}$ on Kuhn Poker. $DCFR_{\alpha=1,\beta=1,\gamma=1}$ is clearly much more stable.*

We also ran some iterations of play with the $CFR+$ policy against random bots to see if the low exploitability results in a higher win rate or utility. The results can be seen in Figure 5. The win rate seems to be distributed equally between the two policies, which makes sense since this is mostly dependent on what cards are dealt to each player. However, the $CFR+$ policy is able to minimize losses on a loss and maximize profit on a win, resulting in a consistently much higher total return. Note that the agent is trained to maximize its return. If the agent would be trained to maximize the number of wins, the best policy would likely be to always bet.



```
0   Policy wins: {'CFR+': 4936, 'Random': 5064}
    Policy returns: {'CFR+': 1655.0, 'Random': -1655.0}
1   Policy wins: {'CFR+': 4854, 'Random': 5146}
    Policy returns: {'CFR+': 1434.0, 'Random': -1434.0}
2   Policy wins: {'CFR+': 4929, 'Random': 5071}
    Policy returns: {'CFR+': 1561.0, 'Random': -1561.0}
3   Policy wins: {'CFR+': 4925, 'Random': 5075}
    Policy returns: {'CFR+': 1573.0, 'Random': -1573.0}
4   Policy wins: {'CFR+': 4865, 'Random': 5135}
    Policy returns: {'CFR+': 1523.0, 'Random': -1523.0}
5   Policy wins: {'CFR+': 4866, 'Random': 5134}
    Policy returns: {'CFR+': 1411.0, 'Random': -1411.0}
6   Policy wins: {'CFR+': 4866, 'Random': 5134}
    Policy returns: {'CFR+': 1500.0, 'Random': -1500.0}
7   Policy wins: {'CFR+': 5001, 'Random': 4999}
    Policy returns: {'CFR+': 1878.0, 'Random': -1878.0}
8   Policy wins: {'CFR+': 4949, 'Random': 5051}
    Policy returns: {'CFR+': 1659.0, 'Random': -1659.0}
9   Policy wins: {'CFR+': 4866, 'Random': 5134}
    Policy returns: {'CFR+': 1414.0, 'Random': -1414.0}
```

***Figure 5:** Playing CFR+ vs a random policy (50% chance of bet, 50% chance of pass) for 10 episodes of 10.000 iterations each on Kuhn Poker. Clearly CFR+ has learned to manage risk properly.*

3.2.2 Choosing on Leduc Poker

Training on Leduc Poker takes much more time (10-11days for 750K iterations/algorithm). Therefore, we decide to only implement 4 algorithms: CFR as a baseline to compare against, $CFR+$ as a theoretical second place, $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ as a theoretical winner and $DCFR_{\alpha=1,\beta=1,\gamma=1}$ since it was the winning algorithm when testing on Kuhn Poker. Figure 6 shows us the resulting exploitability and NashConv for each of the algorithms when training for 750K iterations. Clearly, $CFR+$ is the best algorithm for the game.

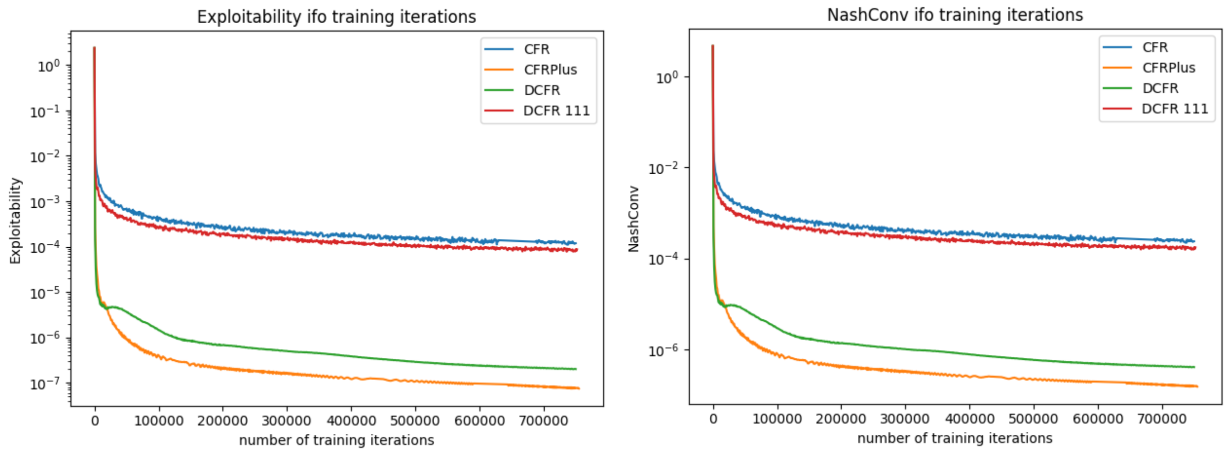
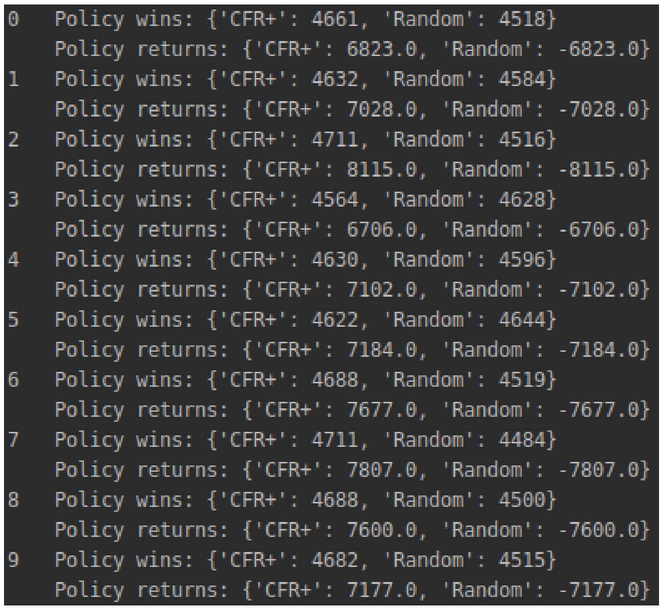


***Figure 6:** Exploitability and NashConv for the 4 algorithms implemented on Leduc poker.*

We tried rounding the values of the best policy as described in section 3.1, but unlike the results for Kuhn Poker, rounding values close to $0, 1, \frac{2}{3}$ or $\frac{1}{3}$ causes the exploitability and NashConv to go up drastically. This is likely because for simpler games like Kuhn Poker it makes more sense to have action probabilities that are exactly equal to intuitive percentages, but for more complex games with more rounds such as Leduc Poker it makes sense for action probabilities to be less comprehensible by intuition.

Finally, we also ran another 10 episodes of 10.000 iterations each of $CFR+$ against a random policy on Leduc Poker. The results can be seen in Figure 7. Similar to Kuhn Poker, the policy win rates are roughly equal, yet the $CFR+$ policy is able to minimize risk and maximize profit effectively. Since it is close to a Nash equilibrium, by definition, the best a policy could do against a trained $CFR+$ policy is (approximately) tie the returns.



```
0   Policy wins: {'CFR+': 4661, 'Random': 4518}
    Policy returns: {'CFR+': 6823.0, 'Random': -6823.0}
1   Policy wins: {'CFR+': 4632, 'Random': 4584}
    Policy returns: {'CFR+': 7028.0, 'Random': -7028.0}
2   Policy wins: {'CFR+': 4711, 'Random': 4516}
    Policy returns: {'CFR+': 8115.0, 'Random': -8115.0}
3   Policy wins: {'CFR+': 4564, 'Random': 4628}
    Policy returns: {'CFR+': 6706.0, 'Random': -6706.0}
4   Policy wins: {'CFR+': 4630, 'Random': 4596}
    Policy returns: {'CFR+': 7102.0, 'Random': -7102.0}
5   Policy wins: {'CFR+': 4622, 'Random': 4644}
    Policy returns: {'CFR+': 7184.0, 'Random': -7184.0}
6   Policy wins: {'CFR+': 4688, 'Random': 4519}
    Policy returns: {'CFR+': 7677.0, 'Random': -7677.0}
7   Policy wins: {'CFR+': 4711, 'Random': 4484}
    Policy returns: {'CFR+': 7807.0, 'Random': -7807.0}
8   Policy wins: {'CFR+': 4688, 'Random': 4500}
    Policy returns: {'CFR+': 7600.0, 'Random': -7600.0}
9   Policy wins: {'CFR+': 4682, 'Random': 4515}
    Policy returns: {'CFR+': 7177.0, 'Random': -7177.0}
```

Figure 7: *Running 10 episodes of 10.000 iterations between a trained CFR+ policy and a random policy on Leduc Poker. We achieved similar results to our training on Kuhn Poker.*

4 Conclusions

We clearly saw the advantage of adding Regret Matching+ and alternating updates to CFR in Figure 3. Although [1] stated that $DCFR_{\alpha=\frac{3}{2},\beta=0,\gamma=2}$ should outperform $CFR+$ in every game (at least the games they implemented), we saw this was not the case for Leduc poker in Figure 5. We managed to accomplish an exploitability value in the order of 10^{-7} by training our agents with $CFR+$ with self-play. Rounding action probabilities might make sense for smaller solution spaces but for a large solution space, it seems to be counterproductive.

Yannou Ravoet r0637112
Robrecht Peeters r0627461

5 How to use the code

The $ML\_project20$ and $open\_spiel$ directories are links to GitHub-repositories (they should be pulled separately when pulling the whole project from GitHub). The $resources$ directory contains all the plots and the two reports. The $src/POKERS$ directory is where you will find the files necessary for training and plotting agents for Kuhn and Leduc poker. Each of the main files contains a variable n to set the number of iterations and a $train\_policies$ function where you can (un)comment whichever solver you do/don't want to use. Training agents saves their intermediate policies in the *policies/<chosen algorithm>/<user determined subdirectory>* directory. Plotting a newly trained agent's convergence, requires you to set the $extract\_metrics$ argument of the $plot\_policies$ function to *True*. This can take quite a while ($\pm$45 minutes for 750 policy files of Leduc Poker). Once the metrics are extracted (and automatically saved in the *metrics/<chosen algorithm>/<user determined subdirectory>* directory), you can increase the plotting speed drastically by setting $extract\_metrics$ to *False.* Both $<poker>\_main.py$ files share the same two utils files: $utils\_poker.py$ for the training and plotting of agents and $policy\_handler.py$ for the loading and saving of policies.

6 Time Spent

We both spent a roughly equal amount of time on the project (around 15 hours/person). Most of this time was spent on the research of algorithms (7 hours/person) and development of the implementations for our needs (5 hours/person). Training the algorithms on Leduc Poker was done in the background on a laptop 24/7 for 19 days. Once the results were in, writing the report and cleaning up the code took the least amount of time (3 hours/person). We did not really keep track of time and think the estimations might be a bit low.

7 Sources

[1] N. Brown and T. Sandholm, "Solving Imperfect-Information Games via Discounted Regret Minimization," *Proc. AAAI Conf. Artif. Intell.*, vol. 33, pp. 1829–1836, Jul. 2019, doi: 10.1609/aaai.v33i01.33011829.
[2] M. Johanson, N. Bard, N. Burch, and M. Bowling, "Finding Optimal Abstract Strategies in Extensive-Form Games," p. 9.
[3] O. Tammelin, N. Burch, M. Johanson, and M. Bowling, "Solving Heads-up Limit Texas Hold'em," p. 8.
[4] T. W. Neller and M. Lanctot, "An Introduction to Counterfactual Regret Minimization," p. 38, 2013.
[5] J. Heinrich and M. Lanctot, "Fictitious Self-Play in Extensive-Form Games," p. 11.
[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, Second edition. Cambridge, Massachusetts: The MIT Press, 2018.
[7] F. Timbers, E. Lockhart, M. Schmid, M. Lanctot, and M. Bowling, "Approximate exploitability: Learning a best response in large games," p. 7.

Yannou Ravoet r0637112
Robrecht Peeters r0627461

8 APPENDIX: Intuitive definition of the used base algorithms and metrics

8.1 Counterfactual regret minimization [4]

In normal-form games, regret of not having chosen an action is defined as the difference in utility of that action and the action that was chosen. Counterfactual regret minimization (CFR) extends regret minimization to sequential games. “Counterfactual” means we treat the computations as if the player of interest played a deterministic path (action probabilities of 1) to reach the state. For sequential games, a counterfactual utility value of a strategy can be computed at a nonterminal history using counterfactual reach probabilities of this history. Counterfactual regret of an action at a history, and finally at a certain information state, can then be computed. Regret-matching can now be used to minimize expected regret trough self-play: starting from a uniformly random policy, we simulate a playthrough of the game. Then the regret for every state passed through in the playthrough is computed and accumulated. The current strategy is then updated (increasing the probability of actions that have a positive regret). Although the current policy does not necessarily converge to a Nash equilibrium, the average over all the previous policies does (in a zero-sum 2-player game).

8.2 Fictitious Play [5]

In fictitious play the idea is to always increase the probability of playing a best response to the opponent’s strategy. The version implemented by OpenSpiel is Extensive-Form Fictitious Play (XFP) where starting from a random average policy π_1 , we iteratively do two steps. In the first step, we compute the best response to the opponent’s current average policy π_j. In the second step, we then update the average policy of each player π_j based on this best response. This means we always only increase the probability of the best response to that iteration’s playthrough (compared to CFR which updates every action probability according to its regret in the playthrough). In a 2-player zero-sum game, this converges to the Nash equilibrium.

8.3 Regret Policy-Gradient [6]

The main idea is to not have to use an action-value function to compute an optimal policy. This means we immediately try to learn the policy parameters (on-policy learning) that result in the highest reward, offering a higher level of abstraction from classic action-value methods.

As such, instead of choosing actions based on their expected reward (through an exploration scheme such as epsilon-greedy exploration), policy gradient methods choose actions based on their probability in the current policy, as expressed by the policy’s parameters. After every episode of play, these parameters are adjusted with gradient descent in the policy space based on some chosen cost function (we chose regret-loss). The intuition is that the action-value function is often incredibly complex and thus hard to learn while the optimal policy might be much easier to approximate.

8.4 Exploitability and NashConv [7]

NashConv measures how close an average policy is to a Nash equilibrium. It is measured by summing the utility of a player’s policy when the opponent always plays the best response to this policy over all the policies. Exploitability divides this value by the number of policies we sum over. A Nash equilibrium has an exploitability of 0, since the best you can get from a best response policy, is a tie.