Combining Reinforcement Learning, Deep Learning and Game theory approaches in building a poker bot

equal contribution*

Mathis Pink*[1] Janosch Bajorath*[2]

Abstract

In this project, we conceptually examine central ideas in computer poker, their relation to Reinforcement Learning and finally provide a re-implementation of Deep Counterfactual Regret Minimization, which combines approaches from Deep Learning, Reinforcement Learning and draws on concepts from Game Theory. The aim is to gain insight into that particular approach to computer poker from a student's perspective. That means, by re-implementing the paper we will examine implicit techniques and ambiguities in both theory and implementation. The report begins with providing the necessary background concepts, ideas and terminology to then explain how these are applied and combined in Deep Counterfactual Regret Minimization. We then show some limited experimental results and discuss the project's insights.

1. Introducing Ideas and Approaches in Computer Poker

In this section, we provide ideas relevant to applying Reinforcement Learning, Deep Learning and game-theory concepts to poker games. We first describe how poker differs from other Reinforcement Learning problems. Then we explain key game theoretic concepts. We discuss the type of observation a poker agent bases its decisions on, the problem of large state spaces as well as sampling methods used to obtain estimates analogous to state-action values but in the context of two-player zero-sum imperfect information games.

1.1. Applying Reinforcement Learning to Poker

Reinforcement learning as a broad field aims at having computer agents that are capable of learning behavior in an environment that maximizes expected returns by interacting with the environment. An environment in which an agent is trained to follow a specific behaviour can be anything from multi-armed bandits to complex three dimensional games and simulations. The returns are typically defined as the sum of discounted rewards, which are defined by the experimenter. In the 2013 mobile hit *Flappy Bird*, the reward is 1 for each flap the bird makes while staying in flight. Such reward is immediately given after an action. The future success depends on the immediate actions taken before, more specifically, the timing of tapping the screen. In other games however, a series of decisions leads to the degree of success and rewards are only given at the end of an episode, i.e. they are sparse rewards. The game of poker, to which this project is dedicated, is one game to which this applies. A poker player does not get immediate feedback from the game about whether a bet was good or bad. It is only after a succession of betting rounds, involving multiple decision points, that the player gets feedback and it is upon the poker player to decide if it is worth risking money to make it to the showdown and obtain reward. This property is not something which makes the game environment of poker a special case for Reinforcement Learning. What separates poker from a game like 'Flappy Bird' is first and perhaps most importantly, that it is a game played by multiple players, all of which we assume to be adaptive agents. Even if it is only one opponent (as in heads-up poker), this alone complicates the process of letting a computer- agent learn to play poker. We simply do not know in advance which agents we will play against. This means we can not just hard code the opponent as a part of the game/environment. A way to deal with this is to successively train an agent in self-play, as it is done for other games such as chess and Go (Silver et al., 2017). The idea is to train the agent by iteratively playing against a previous version of itself, which we then treat as if it were a part of the environment. This way an agent learns first to beat a randomly initialized agent and after that, it learns how to beat such an agent etc. Another complication in applying Reinforcement Learning approaches to letting an agent learn to play poker is that the environment‘s state is unknown to the poker player. Each player only knows its own cards (called hole-cards) along with the publicly observable community cards. While in Flappy Bird, a RL agent would have access to the complete state of the game, that is all of the pixels, what the poker player observes is not the complete state of the environment but only a part of the state. Poker is thus not to be understood as a Markov Decision Process (MDP) but as a Partially Observable Markov Decision Process (POMDP).

1.2. Game Theoretic Concepts in Computer Poker

An important theoretical framework to finding optimal decisions in games is provided by game theory. What is called a reward in RL is called utility or pay-off in game theory. Games can be formalized in two forms. The normal form of a game is usually a table with all the strategy combinations of both players and their respective resulting payoffs. It allows to reason about finding a set of strategies from which there would be no incentive for either player to deviate from their strategies. To represent a game in its normal form requires mutual perfect information between the agents. This is not given in poker, it is thus called an imperfect- information game. In accord with this, two questions arise: How do we frame imperfect-information games in game theory and how does this relate to or help in designing a RL approach to solving poker? Furthermore what does it mean to *solve* poker?

Imperfect information games can be construed as extensive-form games. In extensive-form, a game is interpreted as a game tree. In poker, each node of the tree represents either a decision made by a player or by chance. The root node is the cards being dealt and the leaf nodes are the terminal states of a game (either a showdown or a fold, in which the other player wins the money in the pot without having to show cards). Unlike in normal-form games, extensive-form games assume sequential decisions. An agent however, does not know at which node in the complete game tree it is currently making a decision. This would require knowledge of the cards dealt to the opponent at the root node. We say that the agent makes a decision for an entire information set, that is the set of all game states (decision nodes for that player) that are congruent with the information available to the agent. Following (Brown et al., 2020) in terminology we will refer to the observation that is associated with an information set as the infostate since this highlights its meaning in the context of Reinforcement Learning.

1.3. A Poker Agent's Observations

Contrary to common belief, winning at poker is not about processing information conveyed by the opponent‘s body language, but about patterns in betting, probability and risk assessment. Professional poker players thus not only play successfully on average at live events but also at online tables and tournaments. For a computer-poker agent, the infostate given to the agent contains not only its private and the community cards, but also the betting/action history in this game. Further information that could play a role are stack sizes as well as betting patterns from previous hands. The more information we provide the agent with, the larger the infostate space is. This leads to a problem also faced by an RL agent that learns to play Flappy Bird given the current and previous pixels. Tabular methods from classical Reinforcement Learning become infeasible.

We now give an example of the infostate size in a game we will describe in more detail later. If there are three possible actions 0 to fold/check and 2 or 4 to bet/raise/call, and there is a maximum of 10 possible bets in a game, the total number of possible infostates can be calculated as follows: Let's say there are 52 different cards $+1$ for a not yet dealt card (represented as -1) and a maximum of 10 bets in the bet history with values 0, 2, 4 and again -1 for bets that have not yet occurred. The number of possible infostates then is $53^5 * 4^{10} = 4.39 * 10^{14}$.

We thus need function approximation techniques such as Deep Neural Networks (DNN) to learn the infostate-action values. This is very much similar to how one would train a Deep Q network (Mnih et al., 2013) instead of using a table in Q-Learning. The network takes the betting history for the given hand/game as its input along with information about hole- and community-cards and should output values associated with the possible actions. If folding or checking is optimal for a given info- state, the argmax of the DNN's output should be the action index 0 (or whichever index of the output vector is associated with folding/checking).

1.4. Additional Techniques to Reduce the Infostate Space Size

Another way to reduce the number of possible infostates is to put multiple card combinations into one bucket/cluster. For instance, it does not matter in poker whether the first of the hole-cards is a two (in poker-lingo called deuce) of spades and the second card is a seven of clubs or vice versa. In any pre-flop situation it also does not matter what the colors of the unsuited deuce and seven are. All these infostates could be put into one cluster. Techniques that reduce the number of possible infostates in this way are called (card) abstraction techniques (Brown et al., 2019). A wide variety of past poker bots such as Libratus used abstraction techniques (Brown & Sandholm, 2017). While card abstractions (e.g. using K-means clustering) help in reducing the infostate space, they also provide less granularity and risk equating different situations that call for different strategies.

1.5. Types of Strategies in Poker

In poker it is important to have a stochastic strategy and not a deterministic strategy. Having a deterministic strategy would render the strategy highly exploitable as the agent always acts identically given a particular situation. Acting according to a stochastic policy means not only that the agent explores more infostate - action pairs, it also means that there is higher uncertainty in interpreting the agent's actions and thus in inferencing about its private information. The stochastic strategy depends on the infostate – action values (or advantages, if the infostate value is subtracted as a baseline). This leads to the question of how to determine the infostate action values.

1.6. Estimating Infostate Action Values

One method to achieve this would be to simply take a single Monte-Carlo estimate of the game's outcome (positive or negative payoff for the agent given that it takes action a in infostate s). This however leads to a very large variance, which is not just due to the chance nodes traversed, but also due to the fact that the agents play a stochastic strategy. The opposite of a single monte-carlo sample would be to traverse the entire game-tree and evaluate for every possible outcome. One way to do this is Counterfactual Regret Minimization (abbr. CFR), variants of which are used in almost every poker bot.

1.7. Counterfactual Regret Minimization

In vanilla CFR a game tree is iteratively fully traversed to compute counterfactual regrets for not having taken an action (Neller & Lanctot, 2013). Then a lookup table is filled with these values, that can be understood as infostate action values. In a given situation, we want to take the action that has the highest regret value. By taking the actions that we would regret the most if we would not take them, we minimize the counterfactual regret.

To compute the counterfactual regret of an action, we need the value of an infostate-action pair (given by the payoff) and the values of taking other actions in the same infostate. The regret value then is the infostate action value and subtract the overall expected value when being in this infostate (infostate value). The regret for an action can thus be understood as an advantage-value.

1.8. Exploitability and the Average Strategy in CFR

The iterative component of CFR is that we do not just compute the counterfactual regret values only once. Instead, we repeat the process with the new strategies that we obtain from regret-matching (Brown et al., 2019). Regret-matching is a way to obtain a stochastic policy from regret values. It works by ignoring the negative regrets (actions that are worse than the average action) and normalizing the remaining action regret values to sum up to one. Having obtained the policy after the first iteration (of playing against a randomly acting player), we only know that our strategy works against a player having a random strategy. We repeat playing against the previously trained agent, effectively learning to beat an agent that can beat a random player. This goes on for a given number of CFR iterations. The aim is to have a strategy that is not exploitable, i.e., that represents a Nash Equilibrium if played by both players. Finding a Nash Equilibrium for a poker game equates to having solved the game. A Nash Equilibrium here means that there is no change in strategy that can lead to higher expected payoffs. It is proven that in CFR, it is the average strategy along all iterations that converges to a Nash-Equilibrium (Burch et al., 2012).

1.9. Monte-Carlo CFR

There are numerous variations and improvements upon vanilla CFR. One general variation is Monte- Carlo CFR (Lanctot et al., 2009). In Monte-Carlo CFR, we do not traverse the complete game-tree in a CFR iteration. Instead we sample a number of partial traversals. MC-CFR can itself be implemented to different degrees. Closest to a full-traversal is external sampling MC-CFR. In ES-MC- CFR, we take random samples for chance nodes and for nodes at which the traverser (the player for which we calculate the regrets) is not the one who acts (called non-traverser nodes). For traverser nodes however we evaluate every possible action. A more sparse traversal of the game tree is presented by Outcome-Sampling MC-CFR. In

OS-MC-CFR, we also sample the traverser's actions instead of sampling all actions, making it a more feasible option for large action-spaces (Brown et al., 2019).

1.10. Linear CFR

In Linear CFR, the estimated regret values from different iterations are not weighted equally but with the CFR-iteration as the weight with which the iteration's regret estimates are taken into account (Brown & Sandholm, 2019).

2. Methods

Having taken a brief look at the relevant themes and options in building a poker bot, we go on to discuss the particular methods we implemented and tested in this project. For instructive purposes we chose to re-implement a paper that combines the previously discussed aspects of Poker RL: Deep Counterfactual Regret-Minimization (Brown et al., 2019). In this section, we present the particulars of this algorithm along with decisions that came up in re-implementing the paper that may or may not deviate from the original author's implementation, the code for which remains unpublished. In providing a more explicit description of the algorithm and our implementation of it, along the previously discussed theoretical background, we hope this report to be of help to future students getting into Deep Reinforcement Learning applied to poker. We continue by giving a brief explanation of where Deep CFR fits in the previously discussed context and then provide an extended version of the algorithm‘s pseudocode.

2.1. Deep Counterfactual Regret Minimization (Deep CFR)

Deep CFR uses function approximation instead of card abstraction techniques and (linear) external sampling Monte-Carlo CFR to estimate infostate action regret values (Brown et al., 2019) on which the neural network model is then trained from scratch in each iteration. Two agents take turns in learning to minimize regrets when playing against the other agent's latest trained policy (derived via regret-matching from the regret values).

The algorithm works by sampling for a given number K of (incomplete) game-tree traversals and training the network from scratch afterwards. One traversal consists of traversing the tree for one specific set of dealt hole-cards. We explore all actions of the traverser (whose advantage model we want to train afterwards) but not for the opponent. For the opponent we sample actions based on her current strategy (latest trained model). Chance nodes (community cards) are also sampled and not fully traversed. While traversing the game tree, the opponent's strategy from which actions are sampled is stored in a strategy memory. This is important such that we can afterwards train a network that learns the average strategy, which is shown to converge to the epsilon Nash-Equilibrium (Neller & Lanctot, 2013).

2.2. Reservoir Sampling

After each traversal, we store tuples of (infostate, CFR-iteration, advantages) via reservoir sampling (Vitter, 1985) to a hdf5 file that has a reservoir size of 40 million tuples. In reservoir sampling, the memory is never reset. Instead, the memory keeps track of the number of tuples it has processed so far and once the memory is full, it sometimes replaces a random old datum in favor of the new datum. At other times, it does not store the new data. This works by sampling a random index between 0 and the total number of elements processed so far. If the memory has seen 10 times the data it is capable of storing, the chance to overwrite an old sample with a new sample is $\frac{1}{10}$. Reservoir sampling is proven to result in a sample distribution that on average has the same statistics as what we would get if we would sample without replacement from all elements processed so far (Vitter, 1985).

The use of reservoir sampling instead of a sliding window memory in Deep CFR is empirically shown to lead to lower exploitability and thus a better approximation of the Nash Equilibrium.

2.3. Neural Network Architecture

The network that takes the infostate and outputs either action probabilities or action advantages (regrets) is composed of two parts: A set of card embedding Layers, one for each card type (hole-cards, flop-cards, turn card, river card), a bet-history encoder and a fully connected network that processes both. We replicate the architecture used in Deep CFR without any modifications except for the fact that we use Tensorflow for the implementation while (Brown et al., 2019) used PyTorch. A visualization of the architecture that we use can be found there as well as PyTorch model code.

2.3.1. Card Embeddings Instead of Card Abstraction

The card embedding layer takes card indices and decomposes their meaning into a rank, suit and specific card embedding vector, all of which are then linearly combined to form the embedding of a singular card. A group of cards (e.g. flop cards) is combined by again linearly combining the individual embeddings. Despite the claim made by (Brown et al., 2019) that the algorithm uses less domain knowledge than algorithms employing abstraction techniques, in our opinion this constitutes a non-trivial assumption about the game that is ingrained in the algorithm.

2.3.2. Decomposing and Encoding the bet history

The bet history is a vector of the bet sizes that occurred in the game (a single hand) so far padded with negative values such that it always has the same size. The network has to learn how many bets have already happened and whether it is supposed to call a bet or decide to bet itself. The negative values are clipped to 0 but also used to construct a boolean vector that just contains information about whether a bet already occurred in the game or not. This is important for the model to know how deep into the game it already is and also whether it is the small blind. Again, the idea to feed both bet amounts and bets occurred is domain knowledge ingrained in the model.

2.3.3. Combining bet and card branch to obtain action advantages

Bet and card branches are concatenated into the combined feature vector to be further processed with fully connected layers and ReLU activation functions. Residual connections are used where applicable. The combined and processed feature vector is then standardized to have a mean and standard deviation of zero. We assume this is crucial because the advantages need to sum to zero. The normalization however, as we find makes it harder to fit the model to a game with very high payoffs. An action head takes the normalized features and linearly maps them to the action indices.

For the strategy network, we use regret-matching as part of the model (without any trainable variables).

Since it is not clear in the original paper (Brown et al., 2019) what actions were used, we used two variants. In one, the sampled indices are directly passed to the environment in which a bet of 0 is either interpreted as a fold or a check. In another version (which we ended up evaluating), the indices have fixed meanings. 0 means fold/check, 1 means check/call, 2 means bet/raise one big blind and 3 means bet/raise two big blinds.

2.4. Training the DNN

To train the network, we use a weighted mean squared error loss function where the weights are the CFR-iterations to which the samples in the advantage and strategy memories belong.

(Brown et al., 2019) use a large batch size of 10.000, which we also find to be leading to very high but necessary training times. Same as (Brown et al., 2019) we also train the models with 4000 batches.

Since we weight the loss by the CFR iteration in which the training example was created, we have to use gradient norm clipping, which we also set to 1.

2.5. Pseudocode for Deep CFR

Since a large part of this project was to implement the pseudocode given in (Brown et al., 2019) and we found that strictly following the pseudo-code leads to inconsistencies, we re-print a modified/extended version of the pseudocode for Deep CFR. Along with the pseudo-code we explain aspects of it that are not necessarily clear from the paper.

The outer function in which the different CFR iterations are done and in which the traversals are called starts with initializing the advantage networks for each player to give a 0 output for the action advantages, regardless of its input. Via regret-matching this leads to a uniform probability distribution over the actions. We achieve the negative output by initializing the bias for the action head to be -5. This only affects the sampling in the first iteration. We always train the network with the bias initialized to 0, i.e. random output advantages and not zero outputs for any input.

2.5.1. Recursive Traversal Function

The traversal of the game tree happens recursively. The function takes the environment and its current state h and checks if the game is over (showdown or a player has folded). If the game state is terminal, it returns the payoff for the traverser. This

Algorithm 1 Deep Counterfactual Regret Minimization. Reproduced from (Brown et al., 2019)

```
1:  function DEEPCFR
2:      Initialize each player's advantage network $V(I, a|\theta_p)$ with parameters $\theta_p$ so that it returns 0 for all inputs.
3:      Initialize reservoir-sampled advantage memories $\mathcal{M}_{V,1}, \mathcal{M}_{V,2}$ and strategy memory $\mathcal{M}_\Pi$.
4:      for CFR iteration $t = 1$ to $T$ do
5:          for each player $p$ do
6:              for traversal $k = 1$ to $K$ do
7:                  TRAVERSE($\emptyset, p, \theta_1, \theta_2, \mathcal{M}_{V,p}, \mathcal{M}_\Pi$)    ▷ Collect data from a game traversal with external sampling
8:              Train $\theta_p$ from scratch on loss $\mathcal{L}(\theta_p) = \mathbb{E}_{(I,t',\tilde{r}^{t'})\sim\mathcal{M}_{V,p}}\left[t' \sum_a \left(\tilde{r}^{t'}(a) - V(I, a|\theta_p)\right)^2\right]$
9:      Train $\theta_\Pi$ on loss $\mathcal{L}(\theta_\Pi) = \mathbb{E}_{(I,t',\sigma^{t'})\sim\mathcal{M}_\Pi}\left[t' \sum_a \left(\sigma^{t'}(a) - \Pi(I, a|\theta_\Pi)\right)^2\right]$
10:     return $\theta_\Pi$
```

is where the actual payoff values that are taken into account for the calculation of instantaneous regrets (action advantages). If it's the traverser's turn, the game state is taken as a starting point by copying the environment and re-shuffling the deck. This is not stated in (Brown et al., 2019) but we figure it leads to a less biased estimate of the regret values towards specific environment chance node actions. Not shuffling the cards would be closer to how human (unprofessional) poker players would think about counterfactuals when regretting not having taken an action after seeing the particular flop turnout that happened. For instance a player could fold a deuce-seven off-suit pre-flop only to find out that the flop cards are two sevens and a deuce - giving him a full-house. In most cases it would be a mistake to regret such an action based on the specific rather than the expected outcome of the chance nodes.

We then recursively traverse the game for all possible actions at this decision point and store the results to a value list. Then we subtract the expected value from each value to obtain the action advantages, also called instantaneous regrets (Brown et al., 2019) and add the tuple (infostate, normalized action advantages, CFR iteration) to the player's memory. We normalize the action advantages by dividing by their standard deviation because the high variance in payoffs otherwise leads to difficulties in training the deep learning model since that would require very large weights in the model's action head.

Since in the originally given pseudocode (Brown et al., 2019) there is no return statement given after calculating the instantaneous regrets, we would append $None$ to the values. For this reason we decide to extend the pseudo-code (see line 17) and algorithm by returning the expected value for the infostate. This way we make sure that meaningful values are passed back up through the recursions.

If it's the opponents turn, we sample an action from her strategy and store the strategy for this infostate to the strategy memory that both players share and continue with the traversing the game tree.

3. Experiments and Results

In this section we first explain the poker game environment settings that we used and give preliminary results that we obtained from our Deep CFR implementation. Since the specifics of the poker game used in (Brown et al., 2019) are not disclosed, we can only speculate about how our setup differs from the one used in the original publication. After explaining the game we ran our implementation on, we show plots of the resulting agents in the first iterations.

We evaluate the implementation of the algorithm on a simplified poker variant for only two iterations due to amount of training that is needed in each iteration.

The poker variant consists of two betting rounds: pre-flop and flop. Only bet/raise sizes of 2 and 4 are allowed, effectively reducing the otherwise very high payoff variance. The maximum number of raises in each betting round is set to three, which also limits the payoff variance.

To examine the performance of the trained agents after the first iteration, we pit them against a completely random agent that folds/checks, calls/checks, bets/raises 2 or 4 chips. Since first player 0 learns to beat a random strategy and then player 1 learns to beat the trained player 0, we evaluate player 0 versus a random agent, player 1 versus player 0 and player 1 versus a random agent.

Algorithm 2 CFR Traversal with External Sampling. Adapted from (Brown et al., 2019)

```
 1: function TRAVERSE($h, p, \theta_1, \theta_2, \mathcal{M}_V, \mathcal{M}_\Pi$, t)
 2:     Input: History $h$, traverser player $p$, regret network parameters $\theta$ for each player, advantage memory $\mathcal{M}_V$ for player
        $p$, strategy memory $\mathcal{M}_\Pi$, CFR iteration $t$.
 3:
 4:     if $h$ is terminal then
 5:         return the payoff to player $p$
 6:     else if $h$ is a chance node then
 7:         $a \sim \sigma(h)$
 8:         return TRAVERSE($h \cdot a, p, \theta_1, \theta_2, \mathcal{M}_V, \mathcal{M}_\Pi$, t)
 9:     else if $P(h) = p$ then                                              ▷ If it's the traverser's turn to act
10:         Compute strategy $\sigma^t(I)$ from predicted advantages $V(I(h), a|\theta_p)$ using regret matching.
11:         Shuffle remaining card deck in $h$
12:         for $a \in A(h)$ do
13:             $v(a) \leftarrow$ TRAVERSE($h \cdot a, p, \theta_1, \theta_2, \mathcal{M}_V, \mathcal{M}_\Pi$, t)                ▷ Traverse each action
14:         for $a \in A(h)$ do
15:             $\tilde{r}(I, a) \leftarrow v(a) - \sum_{a' \in A(h)} \sigma(I, a') \cdot v(a')$                 ▷ Compute advantages
16:         Insert the infostate and its normalized action advantages $(I, t, \tilde{r}^t(I))$ into the advantage memory $\mathcal{M}_V$
17:         return expected value $\sum_{a' \in A(h)} \sigma(I, a') \cdot v(a')$                 ▷ missing in (Brown et al., 2019)
18:     else                                                               ▷ If it's the opponent's turn to act
19:         Compute strategy $\sigma^t(I)$ from predicted advantages $V(I(h), a|\theta_{3-p})$ using regret matching.
20:         Insert the infostate and its action probabilities $(I, t, \sigma^t(I))$ into the strategy memory $\mathcal{M}_\Pi$
21:         Sample an action $a$ from the probability distribution $\sigma^t(I)$.
22:         return TRAVERSE($h \cdot a, p, \theta_1, \theta_2, \mathcal{M}_V, \mathcal{M}_\Pi$, t)
```

We evaluate the trained models for 40.000 games. Afterwards we plot the cumulative payoffs as well as the action probabilities for pre-flop and flop.

The figures can be found in the supplementary material.

4. Discussion

We see that training the first agent trained in the first iteration is capable of exploiting strategies. However unlike what would be expected, the second agent to be trained in the first CFR iteration performs worse than the first agent that was solely trained on training examples obtained from playing against a random strategy. The reason for this might be a difference in how many infostates were stored to the memories. When playing against a random agent, games usually last longer and hence there are more infostates to be sampled in one traversal. Since we tried to keep our implementation close to the original paper's, we did not adjust the number of traversals accordingly. Other possible reasons for this behavior might lie in the choices that we had to make on top of what was written by (Brown et al., 2019). Shuffling cards, while theoretically sound, might require more traversals and an even higher batch size for the networks to learn useful patterns. It is also possible that this is actually not an unexpected result and it was the same in the original paper.

Running the algorithm for a higher number of iterations, e.g. > 150 as in (Brown et al., 2019), and training a strategy network, it might be that we still get results that are similar to those in the original paper. Though it is possible that an expensive hyperparameter search is necessary to find a suitable number of traversals, batch size and SGD updates. However training Deep Learning models on poker suffers from very long training times due to the fact that high batch sizes are needed, even for a game in which payoff variance is limited through multiple means. As Brown et al. state, it takes additional means to further reduce payoff variance for the method to work on larger games. It might be that even the simple game that we trained the model on has variance that is too high.

A very important conclusion about computer poker is that it takes a lot of computational power in both CPU and GPU hours. Training a single model took us approx. 5 hours, meaning that 150 CFR iterations require 1505 hours of model training, the last 5 hours of which are for the strategy network. Considering its resource-intensity in time and energy, training Deep CFR is infeasible for most people interested in computer poker. Evaluating and researching such frameworks (by means of ablation experiments, hyperparameter search etc.) is something that only big-tech companies are capable of doing, one of which (Brown et al., 2019) are associated with (Facebook).

The main merit of this project is thus not that the implemented agent achieves extraordinary performance, but that this has been an opportunity to reflect on how the ideas from reinforcement learning can be applied to imperfect-information games.

References

Brown, N. and Sandholm, T. Libratus: The superhuman ai for no-limit poker. In *IJCAI*, 2017.

Brown, N. and Sandholm, T. Solving imperfect-information games via discounted regret minimization, 2019.

Brown, N., Lerer, A., Gross, S., and Sandholm, T. Deep counterfactual regret minimization, 2019. URL https://arxiv.org/abs/1811.00164.

Brown, N., Bakhtin, A., Lerer, A., and Gong, Q. Combining deep reinforcement learning and search for imperfect-information games, 2020.

Burch, N., Lanctot, M., Szafron, D., and Gibson, R. Efficient monte carlo counterfactual regret minimization in games with many player actions. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/3df1d4b96d8976ff5986393e8767f5b2-Paper.pdf.

Lanctot, M., Waugh, K., Zinkevich, M., and Bowling, M. Monte carlo sampling for regret minimization in extensive games. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'09, pp. 1078–1086, Red Hook, NY, USA, 2009. Curran Associates Inc. ISBN 9781615679119.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Neller, T. W. and Lanctot, M. An introduction to counterfactual regret minimization. In *Proceedings of Model AI Assignments, The Fourth Symposium on Educational Advances in Artificial Intelligence (EAAI-2013)*, volume 11, 2013.

Silver, D., Schrittwieser, J., and Simonyan, K. Mastering the game of go without human knowledge. 550, 2017.

Vitter, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985. ISSN 0098-3500. doi: 10.1145/3147.3165. URL https://doi.org/10.1145/3147.3165.

A. Supplementary Material

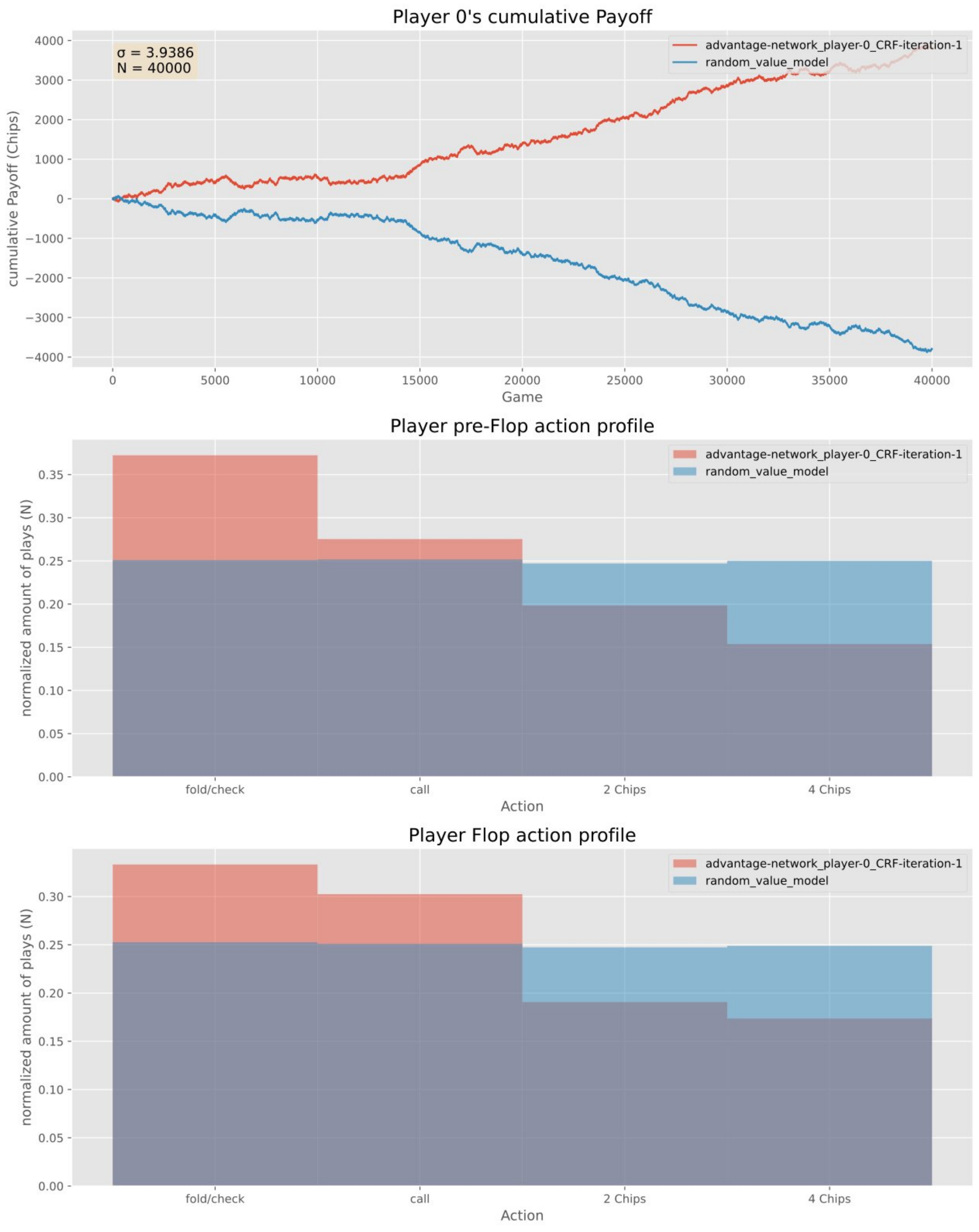

Figure 1. Results for 40000 poker hands between the model of **player 0 at the first CFR iteration** and the **random model**. The first figure shows a plot of each players cumulative reward and the mean of player 0. The Second and third one are showing histograms of the players actions during the pre-Flop and Flop respectively.

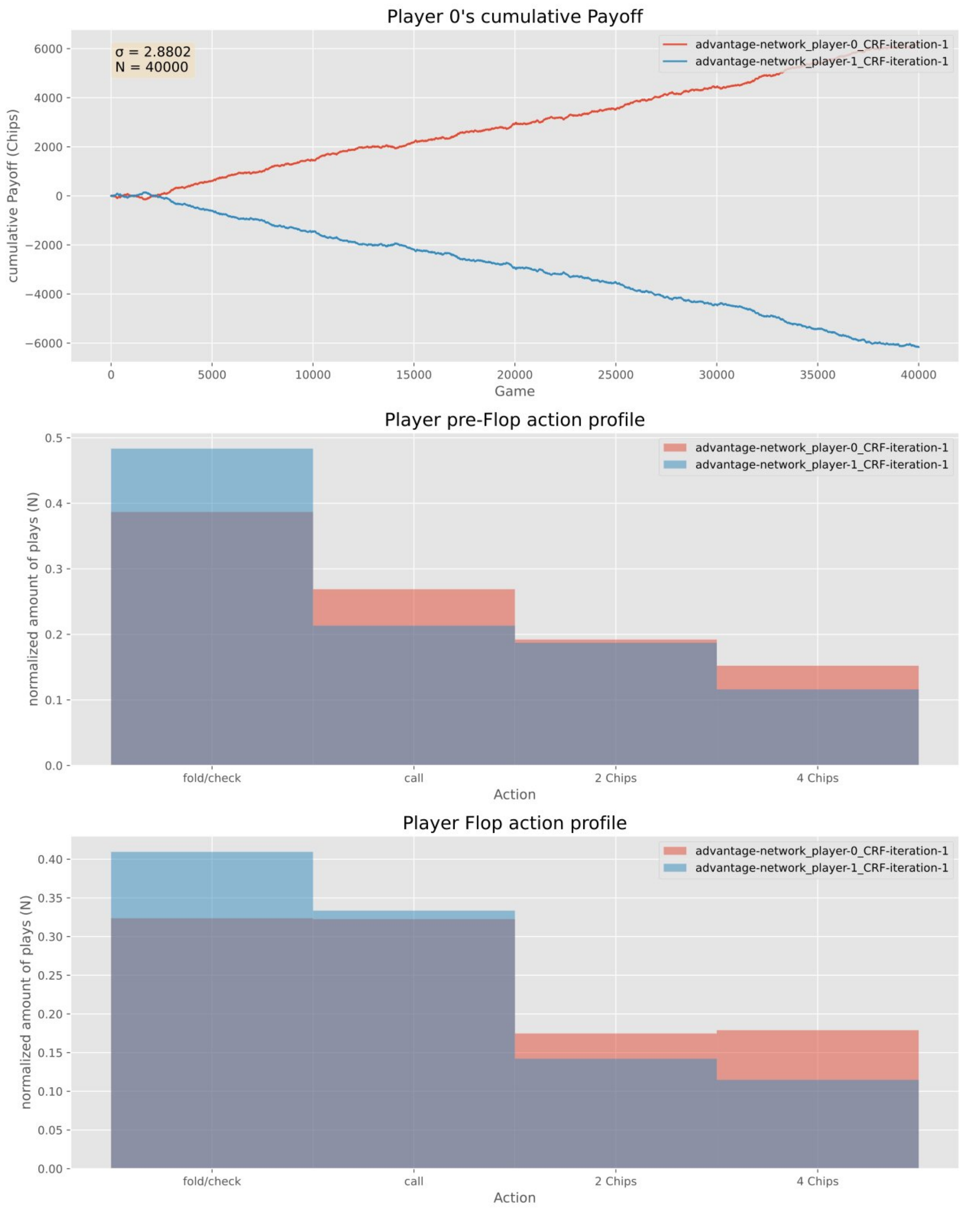

Figure 2. Results for 40000 poker hands between the model of **player 0 at the first CFR iteration** and the **player 1 at the first CFR iteration**. The first figure shows a plot of each players cumulative reward and the mean of player 0. The Second and third one are showing histograms of the players actions during the pre-Flop and Flop respectively.

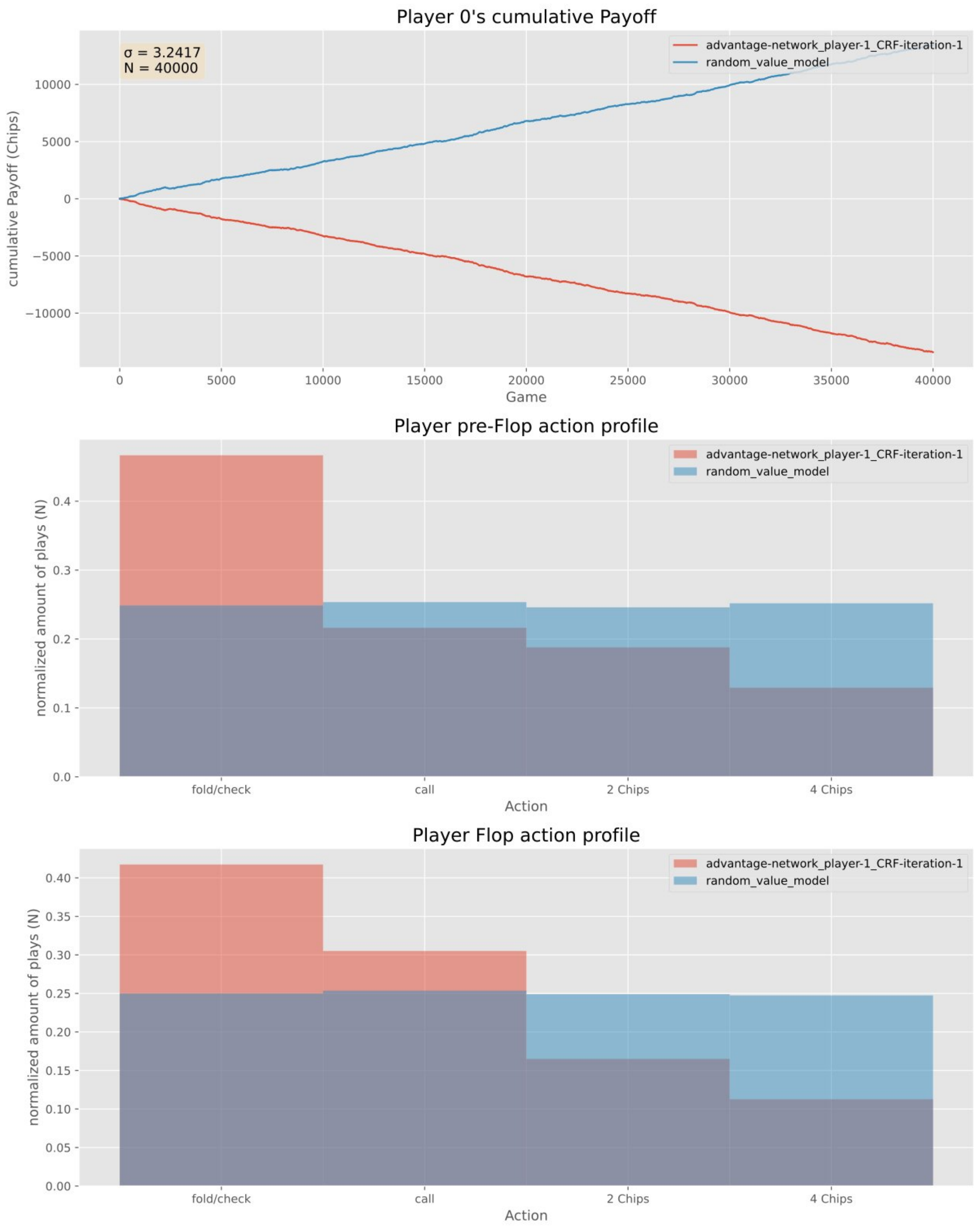

Figure 3. Results for 40000 poker hands between the model of **player 1 at the first CFR iteration** and the **random model**. The first figure shows a plot of each players cumulative reward and the mean of player 0. The Second and third one are showing histograms of the players actions during the pre-Flop and Flop respectively.