Report on DeepStack

Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker

Lasse Becker-Czarnetzki

czarnetzki@cl.uni-heidelberg.de

Abstract

This report focuses on the Paper ”DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker” [5]. The goal of this report is to convey the necessary knowledge to understand DeepStack and the methods used by DeepStack. The creation of overly redundant information for anyone who read the DeepStack paper is kept at a minimum, while detailed explanations are expanded on those parts of DeepStack that are not clearly explained in the paper itself. This includes needed background information and in particular a more detailed explanation of Counterfactual Regret Minimization, which is an underlying algorithm that is used by DeepStack.

Contents

1 Introduction/Need to knows

1.1 Imperfect vs Perfect information games

In research for artificial intelligence (AI) a lot of focus was and is put on developing systems to play games. the reason for this is that we see games as smaller observable and controllable version of the real world. Researchers develop AI to play games, to create methods and learn things about AI that could than be transferred to a real life setting. But if we look at perfect information games this assumption of analogy to the real world seems a little stretched. If we take chess as an example, for every board position you and your opponent have symmetric and complete information, what's more every move that came before to arrive at the current position does not matter for the current decisions. These are circumstances that are not present in real life decisions. In fact our information about the world or our believes about a "opponent" or their believes are incomplete and asymmetrical. At the same time it might be important to reason about how i got to a certain point to make a sound decision.
This is the reason why we want to develop AI for imperfect information games. They give us more inside in how AI could work in a real life setting. For that reason poker is a highly interesting game for AI research. It involves thinking about things your opponent might think about you if you act a certain way, it involves bluffing and reasoning about past actions. It seems to be the closest game to reality we can use for researching the reality in AI.

1.2 HUNL Texas Holdem

I won't explain the whole of poker here but i want to give some points to make this Report understandable and show why we talk about poker.
In our case we play the game called Heads-Up-No-Limit (HUNL) Texas Holdem. This means we play a two player zero-sum game of poker with the texas holdem rules. No-Limit means that there are no limits to the sizing of bets. A player can bet the minimum up to the players full stack size. These bets are wagers on who of the two players is holding the better cards. The two players get dealt two cards each and only each player gets to see their own hold cards. This way both players have asymmetric information about the game, they don't know what cards the opponent is holding. The game consists of up to 4 betting rounds. The players act in alternating fashion. A player can choose to bet or to check, which means to bet nothing. If a player is faced with a bet he can choose the meet it by putting the same amount of money in the middle or choose to give up (fold). The other option is to meet the faced bet with a even higher one. After the first betting round 3 public cards are revealed, which can make combinations with the players private cards. After each of the two following betting rounds another public card gets revealed. A game reaches a terminal state, if a player folds or if the 4. betting round is finished. If that happens the private cards are revealed and the player whose cards form a better combination with the public cards wins. To give an incentive to play more than the best hands at the start of every round minimum bets (blinds) have to be wagered by each player. The game consists of two players who try to maximize their winnings by tricking the other player into thinking they have bad cards when they don't, so they get payed off after the 4. betting round, and good cards if they have bad ones, to make them fold and give up their made bets. The players make these actions by reasoning

about believes and the history of actions, because they don't have perfect information. The game is also influenced by chance due to the randomness of the cards that are dealt. If in poker somebody talks about a players range, they mean the probability distribution over a players possible actions with there possible hold cards. This knowledge is very helpful if one wants to reason about a players action and the cards the player is most likely holding, if he chooses that action. See Figure 1 for a visualization of a poker game tree showing the public states. It shows player 1 and 2 action nodes. The green nodes are chane nodes.

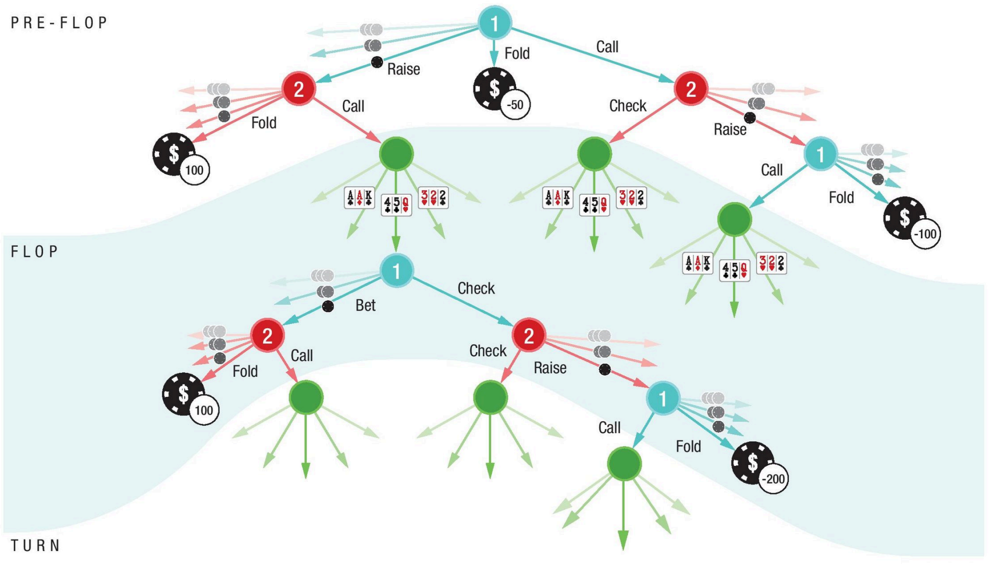

Figure 1: Poker public game tree [5]

1.3 Abstraction

Abstraction is a widely used concept, especially in poker research previous to DeepStack that one must know to dive in to the world of poker AI.
The basic goal of abstraction is to reduce the complexity of a search tree of a game by simplifying it. This smaller game tree than gets solved by some heuristic search algorithm. To use this solved simplified version of the game tree one than translates the actual game states and actions to the one closest in the abstraction and the corresponding strategy is used.
In case of poker there are two main forms of abstraction. By using action abstraction the number of bet sizes that are considered in the game tree get decreased. For example only bet sizes incremented by 50 are used. If the AI than is faced with a bet size not present in the game tree a translation/rounding of that bet size is necessary. This can result in possible strategies of exploitation by choosing specific bet sizes that result in a large inaccuracies through the translation or confuse the AI and general rounding errors.

The other form of abstraction is card abstraction. For this method pairs of cards that are similar get clustered together and treated the same way. This way far less card combinations have to be considered.

1.4 Heuristic search for imperfect information games

To adapt a heuristic search like method to imperfect information games, DeepStack overcomes 2 main problems.
The most significant difference in imperfect information games for solving a particular sub-game is that you can't forget the history of actions that resulted in the current state. The history plays an important role in the decision making progress. DeepStacks core game solving method is Continual re-solving (see section with Counterfactual Regret Minimization (see section 2). While using this method DeepStack keeps track of the believes of which cards the players are holding, which is the concluded information what one would get when thinking about the action history. For a detailed description see section 3.1.
The other big issue is the use of a suitable evaluation function. Heuristic search methods are not able to solve super large games such as poker with over 10^{160} decision points. To combat this only a depth-limited search is done, and if that limit is reached a evaluation function gives an estimate of the utility of the current state, which can be used. In the case of an imperfect information game like poker it is not possible to just evaluate one state with one utility value. We don't have a defined state because of the imperfect information circumstances and one public state (State that is definable through all publicly available information) can't simply be evaluated through by looking at it's state information. The reason is that there are many ways that that public state can be reached (different action histories) and to evaluate a set of possible states accurately one needs to reason about believes again. To learn such a complex evaluation DeepStack uses deep neural networks. For details see section 3.2.

2 Counterfactual Regret Minimization

To understand DeepStack one has to understand the underlying solving algorithm. For this reason i give an introduction to CFR in this section. Counterfactual Regret Minimization is a Algorithm that minimizes the overall regret for actions in a sequential game, a game where a sequence of actions from players is needed to reach a terminal state. By doing so we can compute a Nash Equilibrium strategy for that game or at least a close approximation. CFR was first developed by Zinkevich et al. [8] To understand the underlying algorithm one must first understand the concept of regret and how to compute it and how to derive a strategy from that same regret. For that reasons i first present the concept of regret matching in a simple normal form game before i explain how CFR can be used for Poker.

2.1 Regret Matching

In 2000 Hart and Mas-Colell [4] introduced a game theory algorithm that reaches equilibrium play by choosing actions proportional to positive regrets. They called it regret matching. First i need to establish the meaning of the word regret. The easiest way to

do this is by a simple example. Most commonly one chooses the zero-sum, normal-form game Rock-Paper-Scissors here and so will i. So imagine two players choosing one of three actions (R, P, S) at the same time and betting 1$ on the outcome. If both players choose the same action, they tie otherwise rock(R) beat scissorx, scissors(S) beats paper and paper(P) beats rock. If player 1 chooses paper(P) and player 2 chooses scissors(S), player 1 loses 1$ and player 2 gains 1$. A typical strategy optimizing algorithm would take this utility of -1$ player 1 received and handle it in some way. Regret matching doesn't do this but opens the perspective instead. We ask what could we have gotten if we were to choose another action at that moment.
Player 2 chose scissrs(S) so if we would have chosen rock(R) instead we would have gained 1$ and not lost 1$.
The utility for choosing paper(P) when player 2 chooses scissors(S) is
$u(paper, scissors) = -1$
The utility for choosing rock(R) when player 2 chooses scissors(S) is
$u(rock, scissors) = 1$
To get our regret for not choosing the action rock(R) we compute the difference in the utility we would have gotten versus the utility we actually got.
We regret not having played rock(R)
$u(rock, scissors) - (paper, scissors) = +1 - (-1) = 2$
The same way we can compute our regret for not choosing scissors.
We regret not having played scissors(S)
$u(scissors, scissors) - (paper, scissors) = 0 - (-1) = 1$
because tying would still have been better than losing 1$. Now we know how to compute regrets. After one iteration of playing where we randomly choose paper(R) and player 2 chooses scissors(S) our regrets for the actions (R, P, S) would be (2, 0, 1).
W«e can't compute regret for the action we actually took so the regret for that action is 0.
Now we can derive a strategy for RPS to use in the next playing iteration. We choose our actions proportional to our positive regrets. We do this by simply taking our positive regrets and normalizing them by dividing them with the sum of the total regrets in our game state. Therefore our resulting strategy after this first step would be $(\frac{2}{3}, 0, \frac{1}{3})$ In the next iteration we do the same thing and choose our actions according to the probabilities of our mixed strategy. The resulting regrets get added up to our existing regret values and normalized again. IF all regrets are negative we fall back to a random strategy.
If we do this procedure in self play, meaning the active player get switched after every iteration and their strategy update according to the described method, we converge to an equilibrium. In RPSes case the resulting Nash equilibrium strategy would be $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

2.2 CFR

If we want to use this method of regret matching on a sequential game such as poker we have to make a few additions to the algorithm. We have to not only consider the actions of the two players, but also chance actions (the public cards). We also have to model the fact that a player does not know in which game state he is, due to the incomplete information. If we create a game tree where each node is either a decision node or a chance node, the decision nodes edges are the probabilities of a player choosing an action resulting in another node or terminal state, the edges of a chance node the probabilities

of a chance action occurring, we model nodes that are not distinguishable by the player in information sets. A players strategy for all nodes in an information set is thereby the same. To get a grasp of the algorithm i will now define notation in a similar manner to [6].
Let A denote the set of all game actions (For example: Fold, Bet x amount,...). Let I denote an information set, and $A(I)$ denote the set of legal actions in that information set I. Let t and T denote time steps. t is always in regard to a specific information set and gets incremented every time that information set is visited. A strategy σ_i^t maps probabilities to a player i choosing a, $a \in A(I_i)$ in an Information set I_i at time t for every information set. All strategies together create a strategy profile σ^t. This includes a player 1 , a player 2 and a chance player c whose strategies are just the probabilities of the chance actions occurring in a chance node. If we want to reference a strategy profile that excludes player i's strategy, we denote σ_-i. If we want to say that using a particular strategy is equivalent to σ with the exception that at a particular information set I always action a is chosen we denote $\sigma_{I\to a}$. To define states we use a history h which is a sequence of actions (including chance) starting from the root of the game. The probability to reach a history h with a strategy profile σ is defined as $\pi^\sigma(I)$. Logically the probability to reach a information set I is the sum of the probabilities to reach all histories that are in that information set $\pi^\sigma(I) = \Sigma_{h\in I}\pi^\sigma(h)$. The counterfactual reach probability of an information set I, $\pi^\sigma_{-1}(I)$, is the probability of reaching this information set I with the strategy profile σ except we only take the probabilities of the chance actions and the actions of the other player into account by setting the probabilities of player is actions to reach this information set to 1. By doing this we get the probability of getting to an information set if player i purposely wants to get to that information set. Those are the situations i refer to as counterfactual. I denote Z as the set of all terminal game histories (All sequences of actions that end in a terminal state). I denote a nonterminal game history that is the proper prefix of a terminal one (in other words the game history that precedes a terminal game history) as hz for $z \in Z$ Finally let $u_i(z)$ be the utility to player i for the terminal game history z define the counterfactual value at non-terminal history h as:

$$v_i(\sigma, h) = \sum_{z\in Z, hz} \pi^{\sigma_{-i}}(h)\pi^\sigma(h,z)u_i(z) \tag{1}$$

Now we can define the counterfactual regret analogous to regret matching above
The counterfactual regret of not taking action a at history h is defined as:

$$r(h,a) = v_i(\sigma_{I\to a}, h) - v_i(\sigma, h) \tag{2}$$

Following that the counterfactual regret of not taking an action a at information set I is defined as:

$$r(I,a) = \sum_{h\in I} r(h,a) \tag{3}$$

Now let $r_i^t(I, a)$ be the reference for the regret of player i of not taking action a at information set I, if players use strategy σ^t. Than the cumulative regret is defined as:

$$R_i^T(I, a) = \sum_{t=1}^{T} r_i^t(i, a) \tag{4}$$

The regret can be understood as the difference between the exspected value for always choosing action a and using the strategy σ. This is weighted by the probability to reach the node in which the action is taken only using the probabilities of the opponent playing to that node and the chance actions getting to it. We want to get a strategy proportional to our positive counterfactual regrets. Therefore we define the nonnegative counterfactual regret $R_i^{T,+}(i, a) = max(R_i^T(i, a), 0)$. Now we can apply the regret matching method to compute the new strategy as follows:

$$\sigma_i^{T+1}(i, a) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\sum_{a \in A(I)} R_i^{T,+}(i,a)} & \text{if} \sum_{a \in A(I)} R_i^{T,+}(i, a) > 0 \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases} \tag{5}$$

This equation is used to compute to compute the action probabilities in each information set proportional to the positive cumulative regrets. Again we fall back to a random strategy, if the regrets are negative. In Algorithm 1 the complete algorithm is shown with chance sampling. The algorithm has 5 parameters. The history of actions h, the currently learning player i, the time step t and the reach probabilities for both players 1 and 2 σ_1, σ_2.
Some additional information is needed to fully understand the algorithm. All variables that start with a v are local variables and are not computed by any above mentioned equations. The σ_c strategy is the already mentioned chance player strategy. $P(h)$ just stands for the active player after any history h. ha denotes a history that is identical to history h after action a is taken in that history h. $\emptyset$ just defines a empty history. One detail to keep in mind is that not the final strategy profile that is achieved after a number of iterations converges to an equilibrium but the average strategy for an information set I, $\bar{\sigma}^T$ converges to a Nash equilibrium as $T \to \infty$. This algorithm now can be used to compute a strategy for poker. Every possible states for a public state, that a player could be in, if all cards that the players could be holding are considered, are put in information sets. The thinking of believes over which cards the players are holding is modeled through the probability weighting of the players strategy to reach this information set with certain cards. This way a approximate Nash equilibrium strategy for poker.
DeepStack uses a variation of this vanilla version of CFR (see 3.1).

Algorithm 1 Counterfactual Regret Minimization (with chance sampling)

```
Initialize cumulative regret tables: $\forall I, r_I[a] \leftarrow 0$.
Initialize cumulative strategy tables: $\forall I, s_I[a] \leftarrow 0$.
Initialize initial profile: $\sigma^1(I,a) \leftarrow 1/|A(I)|$

function CFR($h$, $i$, $t$, $\pi_1$, $\pi_2$):
if $h$ is terminal then
    return $u_i(h)$
else if $h$ is a chance node then
    Sample a single outcome $a \sim \sigma_c(h,a)$
    return CFR($ha$, $i$, $t$, $\pi_1$, $\pi_2$)
end if
Let $I$ be the information set containing $h$.
$v_\sigma \leftarrow 0$
$v_{\sigma_{I\rightarrow a}}[a] \leftarrow 0$ for all $a \in A(I)$
for $a \in A(I)$ do
    if $P(h) = 1$ then
        $v_{\sigma_{I\rightarrow a}}[a] \leftarrow$ CFR($ha$, $i$, $t$, $\sigma^t(I,a) \cdot \pi_1$, $\pi_2$)
    else if $P(h) = 2$ then
        $v_{\sigma_{I\rightarrow a}}[a] \leftarrow$ CFR($ha$, $i$, $t$, $\pi_1$, $\sigma^t(I,a) \cdot \pi_2$)
    end if
    $v_\sigma \leftarrow v_\sigma + \sigma^t(I,a) \cdot v_{\sigma_{I\rightarrow a}}[a]$
end for
if $P(h) = i$ then
    for $a \in A(I)$ do
        $r_I[a] \leftarrow r_I[a] + \pi_{-i} \cdot (v_{\sigma_{I\rightarrow a}}[a] - v_\sigma)$
        $s_I[a] \leftarrow s_I[a] + \pi_i \cdot \sigma^t(I,a)$
    end for
    $\sigma^{t+1}(I) \leftarrow$ regret-matching values computed using Equation 5 and regret table $r_I$
end if
return $v_\sigma$

function Solve():
for $t = \{1, 2, 3, \ldots, T\}$ do
    for $i \in \{1, 2\}$ do
        CFR($\emptyset$, $i$, $t$, 1, 1)
    end for
end for
```

Source: Neller and Lanctot [6]

3 DeepStack

3.1 Continual re-solving

The center of DeepStack is the mechanism of Continual re-solving. The basic concept is that DeepStack solves the currently existing sub-game online if the AI has to act. If i say solve i mean a approximated Nash equilibrium strategy is computed via a depth-limited search. Also only a set amount of time is used to make the thinking time comparable to a human if not faster in a lot of cases. To make this feasible depth and breadth limiting methods are applied to the search tree. The search can be done depth-limited because of the evaluation function the Deep Counterfactual neural networks provide (see 3.2).
In addition the breadth of the search space gets limited by action abstraction. For details see 3.3.
CFR like it is explained above applies for solving a whole game tree. Continual re-solving needs to solve sub-game trees without having to solve the preceeding game trunk. A expansion of CFR called CFR-D [2] makes this possible. To be able to solve a particular sub-game continual resolving needs three things, The public state, the players range at that public state, and the opponents counterfactual values. With this information a strategy for the remainder of the game can be reconstructed. Re-solving just means that we don't keep our strategy for the whole game, but rather compute a best strategy in every situation. Since we use CFR as our solver the opponents counterfactual values can easily be taken from that and our range can easily be computed using the player strategy profile. There are three possible action scenarios, where we have to update or information vectors to represent our current game state.
(1) After our own action the current strategy can be disregarded. The opponents counterfactual values are set to those computed in the re-solve resulting from our action. Our range can easily be updated by using Bayes' rule on the previous strategy and the currently computed one.
(2) A chance action occurs: The opponents counterfactual values get updated in the same way by using the new computed values for the re-solve after the chance. For our range hands that are no longer possible just get assigned the probability zero.
(3) Our opponent takes an action: No update is required. This is a very central design point in DeepStack which allows the AI to react exactly to an opponents action. No translation is necessary. DeepStack doesn't face that weaknesses that action abstraction on the opponents bets can bring mentioned in 1.3
To implement this continual resolving DeepStack uses a variation of the vanilla CFR, how it is often called, that is described above. In particular uses a technique known as CFR$^+$ which uses a more efficient version of regret-matching called regret-matching$^+$. The main difference in effect being the efficiency and ability to solve a larger game. For details refer to Tammelin [7].
The biggest issues with vanilla CFR however is that it can only function properly when it is used on a complete game tree. This is why CFR-D [2] is used. A extension of the algorithm that uses decomposition to make solving of a sub-game with CFR possible. This version of CFR uses augmented information sets that also include possible parent nodes. To make solving of a sub-game theoretically sound. In Burch, Johanson, and Bowling [2] and more explicitly in Moravcík et al. [5] it is shown that by using the information, that we update in continual re-solving (Our players range and the opponents counterfactual values), we can still compute a sound and Nash equilibrium approximat-

ing strategy with CFR for a sub-game. For details on the CFR-D, decomposition and the augmented information sets refer to Burch, Johanson, and Bowling [2]. CFR-D also requires that the counterfactual values of our opponent, that we keep in our information vector, are an upper bound of the value the opponent can achieve with each hand in the current public state. At the same time they can't be larger than the value the opponent could have achieved, if they deviated from reaching the current public state. The heart of the solver remains the same as in Vanilla CFR.

3.2 Deep Counterfactual Neural Networks

The second big component that allows DeepStack to use a heuristic search like method are the Deep Counterfactual Neural networks. To avoid redundancy i will not describe these networks in great detail, and the way they are trained and used as they are described clearly and in great detail in the DeepStack paper [5].
The main challenge that these networks overcome is that a public state, that a depth-limited search arrives in, can't be evaluated by some pre-computed value. In re-solving these public states will be reached in many different ways with many different player ranges and counterfactual values. To solve this these networks, which are trained on millions of randomly created poker situations, get the information vectors used in continual re-solving (players range and opponents counterfactual values) and the pot and stack sizes as well as the public cards to define the public state as input. They output counterfactual value vectors for each player which intuitively value each pair of cards a player could be holding. This way a good evaluation of the public state, that is reached in a particular re-solve iteration, can be given to the CFR algorithm. For the input of card information DeepStack uses some card abstraction (see section 1.3. Using this method the search complexity of HUNL can be decreased from 10^{160} to $10^{1}7$. Training these networks offline and especially creating the random poker situations is by far the most computational taxing part of DeepStack. See Figure 2 for an architecture overview of the deep neural nets used for the counterfactual value networks.

3.3 Sparse look-ahead trees

To lessen the breadth of the search tree DeepStack uses action abstraction to limit the amount of bet sizes to consider immensely. It shows to be sufficient to only use a limited amount of bet sizes, only 2-4 different ones, to make the computation much less complex and a lot faster, while having enough options for a good strategy, so that it is applicable for real time matches.

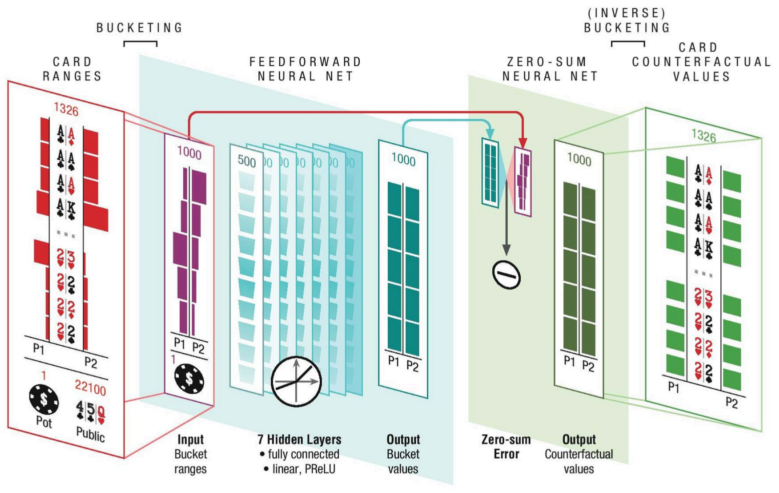

Figure 2: Counterfactual Value Network architecture [5]

4 Evaluation

In this section i want to quickly show the evaluation results of DeepStack. I also want to avoid redundancy here so i refer everyone to Moravcík et al. [5] for a detailed description of DeepStacks evaluation. DeepStack used two variations of evaluation. The played a number of games against pro poker players to see if they can beat them with statiscal significane. They also created a method of measuring the exploitability of DeepStack.

4.1 DeepStack vs Human Pros

The main problem for evaluating against human players is that the chance factor in poker makes it very difficult to clearly see if one player is better than the other through a small number off games. In fact one needs to play up to 100.000 matches to see if one player is better than the other with statistical significance. This creates a problem because it is not feasible to pay pro poker for that number of games or give them some other incentive. The team behind DeepStack therefore created a method for reducing the variance in the result of a poker game. The method is called AIVAT. The basic idea of AIVAT is not to evaluate the results of a poker game by the amount of money that was exchanged, but by the value DeepStack could have gotten in expectation. The difference between this expectation and the actual result can be used to lower the variance. The practical thing about this is that this expectation in a given poker situation can be computed by DeepStacks Counterfactual Value networks (3.2). For any terminal state of a poker match we know DeepStacks range and thereby can get an expected value in the given situation. The variance that arises in the outcome can be reduced by taking

the difference between the expected value and the actual outcome. In the same way AIVAT can reduce variance at every chance action. The differences in expected values before and after a chance action are not in the players or opponents control, so they can be discarded as variance. These are some examples on how AIVAT can achieve variance reduction on poker, for a more detailed description refer to Burch et al. [3].
By using AIVAT the number of games that have to be played to measure a result with statistical significance is reduced to 3000. They let pros play these 3000 games against DeepStack giving an incentive by rewarding the 10 best performing. In the poker community a measurement for a player exists called mbb/g (milli big blinds per game). It basically is a normalized value to measure the winnings of a player over a span of multiple games. A poker pros aims to get 50 mbb/g consistantly to turn a profit. Always folding would result in -750 mbb/g. To make it quick, DeepStack successfully beat the contestants with statistical signifcance. The results can be seen in Figure 3

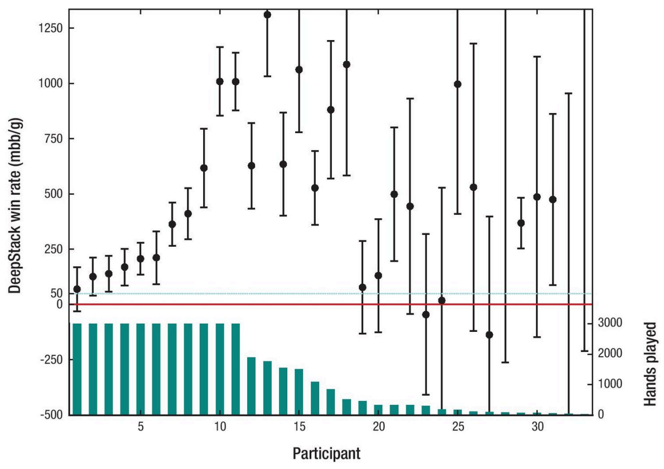

Figure 3: DeepStacks performance against human pro poker players [5]

One detail that can be criticized in this experiment is that they did choose poker pros to player against, but not experts in HUNL. Poker is usually played in a multiplayer fashion up to 10 players. The heads-up variant requires specific skills and understanding of the game at the highest level, which some of the participants might not possess but experts in HUNL might.

4.2 Exploitability of DeepStack

In game theory exploitability is a measurement that checks a strategy's difference to a Nash equilibrium strategy. The best response to a strategy would result in a tie of one is playing in a Nash equilibrium. The amount of losses that are created different to that tie is the exploitability of a strategy that only approximates a Nash equilibrium. Because no-limit poker is to complex no best respone strategy can be computed. The authors of DeepStack try to approximate a best response by looking at DeepStacks action probabilities in a poker situation and creating a local best response. In terms of results they can only show that this method can't exploit DeepStack. This shows that EeepStack has achieved a not trivial approximation of a Nash equilibrium strategy but since it is still an approximation the method of local best response seemst not sufficient enough to properly measure true exploitability but it makes it possible to compare to other AI systems. They show that earlier Poker AIs lose to a local best respone strategy but comparisons to other sota systems such as Libratus is not available.

5 Discussion

5.1 DeepStack vs Libratus

Almost at the same time as DeepStack a different independently developed HUNL Poker AI was developed. It is called Libratus [1] and most notably it also makes use of CFR and a similar technique to continues Re-solving. Libratus beat HUNL Poker Pros with statistical significance and also beat other previously released top Poker AIs.Notably the played against experts in HUNL unlike DeepStack It was never shown if DeepStack was able to beat previous Poker AIs 1v1 or even Libratus itself. To give a wider view on different attempts to solve the problem of HUNL Poker i want to call out some differences between Libratus and DeepStack.
Libratus does have more additional features. For example it can account for possible mistakes the opponent made in previous actions. DeepStake always assumes equally good play. Libratus acts in the first two betting rounds according to a pre-computed blueprint strategy. Like discussed above, DeepStack doesn't have the problem of abstracting the opponents action because it computes a real time response specific to that action. Libratus on the other hand needs to round the opponents bet to a bet that is on the known game tree of the blueprint strategy. Libratus combatants this problem by using a self improvement module, which considers the seen bet sizes of the opponent and adds problematic ones to the game tree. Unlike DeepStack Libratus doesn't use any Deep Learning method. A big practical difference to consider is that DeepStack can run on a simple GPU at test time, while Libratus does need a lot more computing power.

5.2 Conclusion

DeepStack is a HUNL Poker AI that successfully beats human professionals with statistical significance in the game. DeepStack shows that the adaption of methods used in AI for perfect information is possible and usefull.
The fact that DeepStack is breaking into the super human level of play for a popular game such as poker and that the AI can technically be run on a simple personal laptop raises an issue for the online poker community. Since further findings in this line of research in the next years are to be expected and recreating such an AI without needing a lot of resources seems now more possible than ever, gambling sites need to be aware of the possible misuse of such an AI. But even if DeepStack is closer than Equilibrium strategy than professional players, it seems unlikely that the use of such a strategy is more profitable than those of the pros yet. Since humans aren't perfect and especially

amateurs won't be playing game theory optimal, the strategies of the professionals to exploit these mistakes, might still be superior in the sense off accumulating profit. Nevertheless achieves DeepStack a big step in AI research and imperfect information games in general since no to domain specific learning was used and methods like continual re-solving could get transferred to other areas.

6 References

[1] Noam Brown and Tuomas Sandholm. "Libratus: The Superhuman AI for No-Limit Poker". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 5226–5228. DOI: 10.24963/ijcai.2017/772. URL: https://doi.org/10.24963/ijcai.2017/772.

[2] Neil Burch, Michael Johanson, and Michael Bowling. "Solving Imperfect Information Games Using Decomposition". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI'14. Qu&#233;bec City, Qu&#233;bec, Canada: AAAI Press, 2014, pp. 602–608. URL: http://dl.acm.org/citation.cfm?id=2893873.2893967.

[3] Neil Burch et al. "AIVAT: A New Variance Reduction Technique for Agent Evaluation in Imperfect Information Games". In: *CoRR* abs/1612.06915 (2016). arXiv: 1612.06915. URL: http://arxiv.org/abs/1612.06915.

[4] Sergiu Hart and Andreu Mas-Colell. "A Simple Adaptive Procedure Leading to Correlated Equilibrium". In: *Econometrica* 68.5 (2000), pp. 1127–1150. DOI: 10.1111/1468-0262.00153. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1468-0262.00153. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-0262.00153.

[5] Matej Moravcík et al. "DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker". In: *CoRR* abs/1701.01724 (2017). arXiv: 1701.01724. URL: http://arxiv.org/abs/1701.01724.

[6] Todd W. Neller and Marc Lanctot. "An Introduction to Counterfactual Regret Minimization". In: 2013.

[7] Oskari Tammelin. "Solving Large Imperfect Information Games Using CFR+". In: (July 2014).

[8] 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: 978-1-60560-352-0. URL: http://dl.acm.org/citation.cfm?id=2981562.2981779.