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

Matej Moravčík[♠,♥,†], Martin Schmid[♠,♥,†], Neil Burch[♠], Viliam Lisý[♠,♣],
Dustin Morrill[♠], Nolan Bard[♠], Trevor Davis[♠],
Kevin Waugh[♠], Michael Johanson[♠], Michael Bowling[♠,∗]

[♠]Department of Computing Science, University of Alberta,
Edmonton, Alberta, T6G2E8, Canada

[♥]Department of Applied Mathematics, Charles University,
Prague, Czech Republic

[♣]Department of Computer Science, FEE, Czech Technical University,
Prague, Czech Republic

[†]These authors contributed equally to this work and are listed in alphabetical order.

[∗]To whom correspondence should be addressed; E-mail: bowling@cs.ualberta.ca

Artificial intelligence has seen several breakthroughs in recent years, with games often serving as milestones. A common feature of these games is that players have perfect information. Poker is the quintessential game of imperfect information, and a longstanding challenge problem in artificial intelligence. We introduce DeepStack, an algorithm for imperfect information settings. It combines recursive reasoning to handle information asymmetry, decomposition to focus computation on the relevant decision, and a form of intuition that is automatically learned from self-play using deep learning. In a study involving 44,000 hands of poker, DeepStack defeated with statistical significance professional poker players in heads-up no-limit Texas hold'em. The approach is theoretically sound and is shown to produce more difficult to exploit strategies than prior approaches. [1]

Games have long served as benchmarks and marked milestones of progress in artificial intelligence (AI). In the last two decades, computer programs have reached a performance that

---

1

exceeds expert human players in many games, e.g., backgammon (*1*), checkers (*2*), chess (*3*), Jeopardy! (*4*), Atari video games (*5*), and go (*6*). These successes all involve games with information symmetry, where all players have identical information about the current state of the game. This property of perfect information is also at the heart of the algorithms that enabled these successes, e.g., local search during play (*7, 8*).

The founder of modern game theory and computing pioneer, von Neumann, envisioned reasoning in games without perfect information. "Real life is not like that. Real life consists of bluffing, of little tactics of deception, of asking yourself what is the other man going to think I mean to do. And that is what games are about in my theory." (*9*) One game that fascinated von Neumann was poker, where players are dealt private cards and take turns making bets or bluffing on holding the strongest hand, calling opponents' bets, or folding and giving up on the hand and the bets already added to the pot. Poker is a game of imperfect information, where players' private cards give them asymmetric information about the state of game.

Heads-up no-limit Texas hold'em (HUNL) is a two-player version of poker in which two cards are initially dealt face-down to each player, and additional cards are dealt face-up in three subsequent rounds. No limit is placed on the size of the bets although there is an overall limit to the total amount wagered in each game (*10*). AI techniques have previously shown success in the simpler game of heads-up limit Texas hold'em, where all bets are of a fixed size resulting in just under 10^{14} decision points (*11*). By comparison, computers have exceeded expert human performance in go (*6*), a perfect information game with approximately 10^{170} decision points (*12*). The imperfect information game HUNL is comparable in size to go, with the number of decision points exceeding 10^{160} (*13*).

Imperfect information games require more complex reasoning than similarly sized perfect information games. The correct decision at a particular moment depends upon the probability distribution over private information that the opponent holds, which is revealed through their past actions. However, how our opponent's actions reveal that information depends upon their knowledge of our private information and how our actions reveal it. This kind of recursive reasoning is why one cannot easily reason about game situations in isolation, which is at the heart of heuristic search methods for perfect information games. Competitive AI approaches in imperfect information games typically reason about the entire game and produce a complete strategy prior to play (*14–16*). Counterfactual regret minimization (CFR) (*14,17,18*) is one such technique that uses self-play to do recursive reasoning through adapting its strategy against itself over successive iterations. If the game is too large to be solved directly, the common response is to solve a smaller, abstracted game. To play the original game, one translates situations and actions from the original game to the abstract game.

Although this approach makes it feasible for programs to reason in a game like HUNL, it does so by squeezing HUNL's 10^{160} situations down to the order of 10^{14} abstract situations. Likely as a result of this loss of information, such programs are behind expert human play. In 2015, the computer program Claudico lost to a team of professional poker players by a margin of 91 mbb/g (*19*), which is a "huge margin of victory" (*20*). Furthermore, it has been recently shown that abstraction-based programs from the Annual Computer Poker Competition have

massive flaws (*21*). Four such programs (including top programs from the 2016 competition) were evaluated using a local best-response technique that produces an approximate lower-bound on how much a strategy can lose. All four abstraction-based programs are beatable by over 3,000 mbb/g, which is four times as large as simply folding each game.

DeepStack takes a fundamentally different approach. It continues to use the recursive reasoning of CFR to handle information asymmetry. However, it does not compute and store a complete strategy prior to play and so has no need for explicit abstraction. Instead it considers each particular situation as it arises during play, but not in isolation. It avoids reasoning about the entire remainder of the game by substituting the computation beyond a certain depth with a fast approximate estimate. This estimate can be thought of as DeepStack's intuition: a gut feeling of the value of holding any possible private cards in any possible poker situation. Finally, DeepStack's intuition, much like human intuition, needs to be trained. We train it with deep learning (*22*) using examples generated from random poker situations. We show that DeepStack is theoretically sound, produces strategies substantially more difficult to exploit than abstraction-based techniques, and defeats professional poker players at HUNL with statistical significance.

DeepStack

DeepStack is a general-purpose algorithm for a large class of sequential imperfect information games. For clarity, we will describe its operation in the game of HUNL. The state of a poker game can be split into the players' private information, hands of two cards dealt face down, and the public state, consisting of the cards laying face up on the table and the sequence of betting actions made by the players. Possible sequences of public states in the game form a public tree with every public state having an associated public subtree (Fig. 1).

A player's strategy defines a probability distribution over valid actions for each decision point, where a decision point is the combination of the public state and the hand for the acting player. Given a player's strategy, for any public state one can compute the player's range, which is the probability distribution over the player's possible hands given that the public state is reached.

Fixing both players' strategies, the utility for a particular player at a terminal public state, where the game has ended, is a bilinear function of both players' ranges using a payoff matrix determined by the rules of the game. The expected utility for a player at any other public state, including the initial state, is the expected utility over reachable terminal states given the players' fixed strategies. A best-response strategy is one that maximizes a player's expected utility against an opponent strategy. In two-player zero-sum games, like HUNL, a solution or Nash equilibrium strategy (*23*) maximizes the expected utility when playing against a best-response opponent strategy. The exploitability of a strategy is the difference in expected utility against its best-response opponent and the expected utility under a Nash equilibrium.

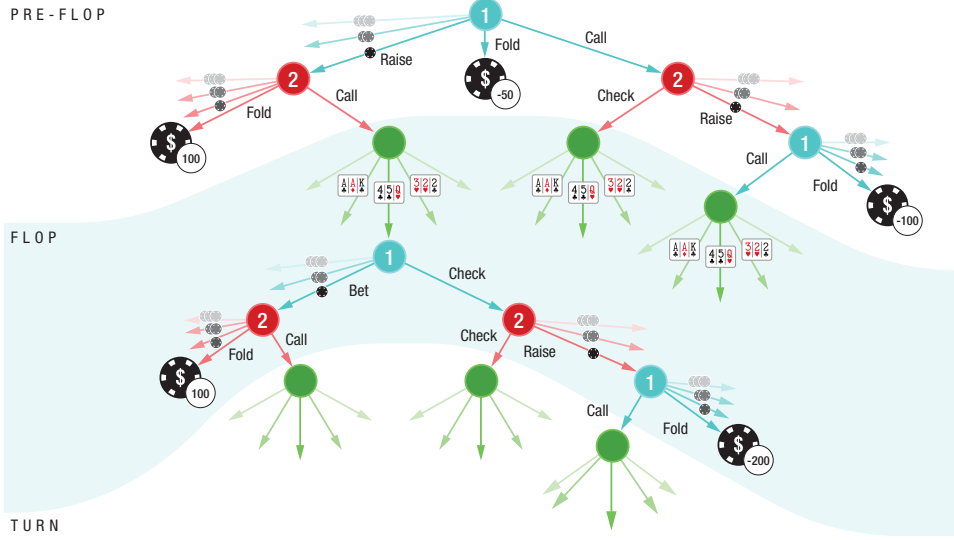The DeepStack algorithm seeks to compute and play a low-exploitability strategy for the

Figure 1: A portion of the public tree in HUNL. Nodes represent public states, whereas edges represent actions: red and turquoise showing player betting actions, and green representing public cards revealed by chance. The game ends at terminal nodes, shown as a chip with an associated value. For terminal nodes where no player folded, the player whose private cards form a stronger poker hand receives the value of the state.

game, i.e., solve for an approximate Nash equilibrium. DeepStack computes this strategy during play only for the states of the public tree that actually arise. Although computed during play, DeepStack's strategy is static, albeit stochastic, because it is the result of a deterministic computation that produces a probability distribution over the available actions.

The DeepStack algorithm (Fig. 2) is composed of three ingredients: a sound local strategy computation for the current public state, depth-limited lookahead using a learned value function to avoid reasoning to the end of the game, and a restricted set of lookahead actions. At a conceptual level these three ingredients describe heuristic search, which is responsible for many of AI's successes in perfect information games. Until DeepStack, no theoretically sound application of heuristic search was known in imperfect information games. The heart of heuristic search methods is the idea of "continual re-searching", where a sound local search procedure is invoked whenever the agent must act without retaining any memory of how or why it acted to reach the current state. At the heart of DeepStack is continual re-solving, a sound local strategy computation which only needs minimal memory of how and why it acted to reach the current public state.

Continual re-solving. Suppose we have taken actions according to a particular solution strategy but then in some public state forget this strategy. Can we reconstruct a solution strategy for the subtree without having to solve the entire game again? We can, through the process of

4

re-solving (*17*). We need to know both our range at the public state and a vector of expected values achieved by the opponent under the previous solution for each opponent hand (*24*). With these values, we can reconstruct a strategy for only the remainder of the game, which does not increase our overall exploitability. Each value in the opponent's vector is a counterfactual value, a conditional "what-if" value that gives the expected value if the opponent reaches the public state with a particular hand. The CFR algorithm also uses counterfactual values, and if we use CFR as our solver, it is easy to compute the vector of opponent counterfactual values at any public state.

Re-solving, however, begins with a strategy, whereas our goal is to avoid ever maintaining a strategy for the entire game. We get around this by doing continual re-solving: reconstructing a strategy by re-solving every time we need to act; never using the strategy beyond our next action. To be able to re-solve at any public state, we need only keep track of our own range and a suitable vector of opponent counterfactual values. These values must be an upper bound on the value the opponent can achieve with each hand in the current public state, while being no larger than the value the opponent could achieve had they deviated from reaching the public state. This is an important relaxation of the counterfactual values typically used in re-solving, with a proof of sufficiency included in our proof of Theorem 1 below (*10*).

At the start of the game, our range is uniform and the opponent counterfactual values are initialized to the value of being dealt each private hand. When it is our turn to act we re-solve the subtree at the current public state using the stored range and opponent values, and act according to the computed strategy, discarding the strategy before we act again. After each action, either by a player or chance dealing cards, we update our range and opponent counterfactual values according to the following rules: (i) Own action: replace the opponent counterfactual values with those computed in the re-solved strategy for our chosen action. Update our own range using the computed strategy and Bayes' rule. (ii) Chance action: replace the opponent counterfactual values with those computed for this chance action from the last re-solve. Update our own range by zeroing hands in the range that are impossible given new public cards. (iii) Opponent action: no change to our range or the opponent values are required.

These updates ensure the opponent counterfactual values satisfy our sufficient conditions, and the whole procedure produces arbitrarily close approximations of a Nash equilibrium (see Theorem 1). Notice that continual re-solving never keeps track of the opponent's range, instead only keeping track of their counterfactual values. Furthermore, it never requires knowledge of the opponent's action to update these values, which is an important difference from traditional re-solving. Both will prove key to making this algorithm efficient and avoiding any need for the translation step required with action abstraction methods (*25, 26*).

Continual re-solving is theoretically sound, but by itself impractical. While it does not ever maintain a complete strategy, re-solving itself is intractable except near the end of the game. In order to make continual re-solving practical, we need to limit the depth and breadth of the re-solved subtree.

Limited depth lookahead via intuition. As in heuristic search for perfect information games, we would like to limit the depth of the subtree we have to reason about when re-solving. However, in imperfect information games we cannot simply replace a subtree with a heuristic or precomputed value. The counterfactual values at a public state are not fixed, but depend on how players play to reach the public state, i.e., the players' ranges (*17*). When using an iterative algorithm, such as CFR, to re-solve, these ranges change on each iteration of the solver.

DeepStack overcomes this challenge by replacing subtrees beyond a certain depth with a learned counterfactual value function that approximates the resulting values if that public state were to be solved with the current iteration's ranges. The inputs to this function are the ranges for both players, as well as the pot size and public cards, which are sufficient to specify the public state. The outputs are a vector for each player containing the counterfactual values of holding each hand in that situation. In other words, the input is itself a description of a poker game: the probability distribution of being dealt individual private hands, the stakes of the game, and any public cards revealed; the output is an estimate of how valuable holding certain cards would be in such a game. The value function is a sort of intuition, a fast estimate of the value of finding oneself in an arbitrary poker situation. With a depth limit of four actions, this approach reduces the size of the game for re-solving from 10^{160} decision points at the start of the game down to no more than 10^{17} decision points. DeepStack uses a deep neural network as its learned value function, which we describe later.

Sound reasoning. DeepStack's depth-limited continual re-solving is sound. If DeepStack's intuition is "good" and "enough" computation is used in each re-solving step, then DeepStack plays an arbitrarily close approximation to a Nash equilibrium.

Theorem 1 *If the values returned by the value function used when the depth limit is reached have error less than ϵ, and T iterations of CFR are used to re-solve, then the resulting strategy's exploitability is less than $k_1\epsilon + k_2/\sqrt{T}$, where k_1 and k_2 are game-specific constants. For the proof, see (10).*

Sparse lookahead trees. The final ingredient in DeepStack is the reduction in the number of actions considered so as to construct a sparse lookahead tree. DeepStack builds the lookahead tree using only the actions fold (if valid), call, 2 or 3 bet actions, and all-in. This step voids the soundness property of Theorem 1, but it allows DeepStack to play at conventional human speeds. With sparse and depth-limited lookahead trees, the re-solved games have approximately 10^7 decision points, and are solved in under five seconds using a single NVIDIA GeForce GTX 1080 graphics card. We also use the sparse and depth-limited lookahead solver from the start of the game to compute the opponent counterfactual values used to initialize DeepStack's continual re-solving.

Relationship to heuristic search in perfect information games. There are three key challenges that DeepStack overcomes to incorporate heuristic search ideas in imperfect information
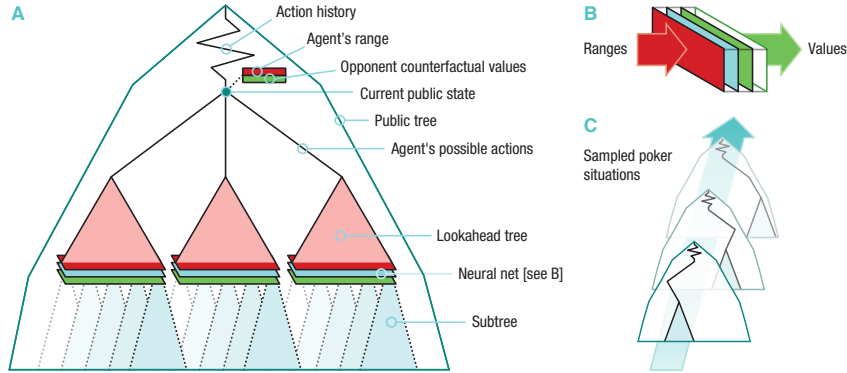
Figure 2: **DeepStack overview.** (**A**) DeepStack reasons in the public tree always producing action probabilities for all cards it can hold in a public state. It maintains two vectors while it plays: its own range and its opponent's counterfactual values. As the game proceeds, its own range is updated via Bayes' rule using its computed action probabilities after it takes an action. Opponent counterfactual values are updated as discussed under "Continual re-solving". To compute action probabilities when it must act, it performs a re-solve using its range and the opponent counterfactual values. To make the re-solve tractable it restricts the available actions of the players and lookahead is limited to the end of the round. During the re-solve, counterfactual values for public states beyond its lookahead are approximated using DeepStack's learned evaluation function. (**B**) The evaluation function is represented with a neural network that takes the public state and ranges from the current iteration as input and outputs counterfactual values for both players (Fig. 3). (**C**) The neural network is trained prior to play by generating random poker situations (pot size, board cards, and ranges) and solving them to produce training examples. Complete pseudocode can be found in Algorithm S1 (*10*).

games. First, sound re-solving of public states cannot be done without knowledge of how and why the players acted to reach the public state. Instead, two additional vectors, the agent's range and opponent counterfactual values, must be maintained to be used in re-solving. Second, re-solving is an iterative process that traverses the lookahead tree multiple times instead of just once. Each iteration requires querying the evaluation function again with different ranges for every public state beyond the depth limit. Third, the evaluation function needed when the depth limit is reached is conceptually more complicated than in the perfect information setting. Rather than returning a single value given a single state in the game, the counterfactual value function needs to return a vector of values given the public state and the players' ranges. Because of this complexity, to learn such a value function we use deep learning, which has also been successful at learning complex evaluation functions in perfect information games (*6*).

Relationship to abstraction-based approaches. Although DeepStack uses ideas from abstraction, it is fundamentally different from abstraction-based approaches. DeepStack restricts the number of actions in its lookahead trees, much like action abstraction (*25, 26*). However,

each re-solve in DeepStack starts from the actual public state and so it always perfectly understands the current situation. The algorithm also never needs to use the opponent's actual action to obtain correct ranges or opponent counterfactual values, thereby avoiding translation of opponent bets. We used hand clustering as inputs to our counterfactual value functions, much like explicit card abstraction approaches (*27, 28*). However, our clustering is used to estimate counterfactual values at the end of a lookahead tree rather than limiting what information the player has about their cards when acting. We later show that these differences result in a strategy substantially more difficult to exploit.

Deep Counterfactual Value Networks

Deep neural networks have proven to be powerful models and are responsible for major advances in image and speech recognition (*29, 30*), automated generation of music (*31*), and game-playing (*5, 6*). DeepStack uses deep neural networks with a tailor-made architecture, as the value function for its depth-limited lookahead (Fig. 3). Two separate networks are trained: one estimates the counterfactual values after the first three public cards are dealt (flop network), the other after dealing the fourth public card (turn network). An auxiliary network for values before any public cards are dealt is used to speed up the re-solving for early actions (*10*).

Architecture.   DeepStack uses a standard feedforward network with seven fully connected hidden layers each with 500 nodes and parametric rectified linear units (*32*) for the output. This architecture is embedded in an outer network that forces the counterfactual values to satisfy the zero-sum property. The outer computation takes the estimated counterfactual values, and computes a weighted sum using the two players' input ranges resulting in separate estimates of the game value. These two values should sum to zero, but may not. Half the actual sum is then subtracted from the two players' estimated counterfactual values. This entire computation is differentiable and can be trained with gradient descent. The network's inputs are the pot size as a fraction of the players' total stacks and an encoding of the players' ranges as a function of the public cards. The ranges are encoded by clustering hands into 1,000 buckets, as in traditional abstraction methods (*27, 28, 33*), and input as a vector of probabilities over the buckets. The output of the network are vectors of counterfactual values for each player and hand, interpreted as fractions of the pot size.

Training.   The turn network was trained by solving 10 million randomly generated poker turn games. These turn games used randomly generated ranges, public cards, and a random pot size (*10*). The target counterfactual values for each training game were generated by solving the game with players' actions restricted to fold, call, a pot-sized bet, and an all-in bet, but no card abstraction. The flop network was trained similarly with 1 million randomly generated flop games. However, the target counterfactual values were computed using our depth-limited
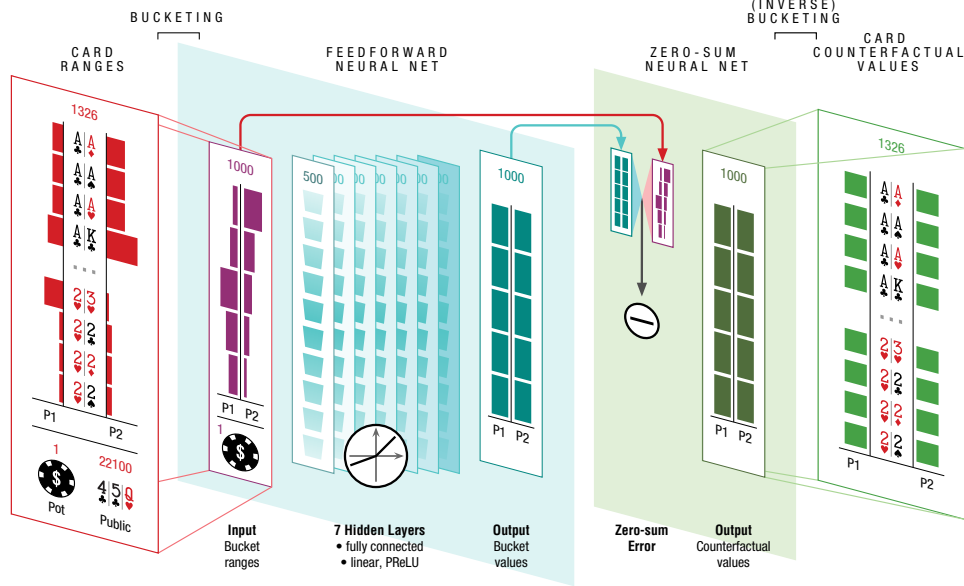
Figure 3: **Deep counterfactual value network.** The inputs to the network are the pot size, public cards, and the player ranges, which are first processed into hand clusters. The output from the seven fully connected hidden layers is post-processed to guarantee the values satisfy the zero-sum constraint, and then mapped back into a vector of counterfactual values.

solving procedure and our trained turn network. The networks were trained using the Adam gradient descent optimization procedure (*34*) with a Huber loss (*35*).

Evaluating DeepStack

We evaluated DeepStack by playing it against a pool of professional poker players recruited by the International Federation of Poker (*36*). Thirty-three players from 17 countries were recruited. Each was asked to complete a 3,000 game match over a period of four weeks between November 7th and December 12th, 2016. Cash incentives were given to the top three performers ($5,000, $2,500, and $1,250 CAD).

Evaluating performance in HUNL is challenging because of the large variance in per-game outcomes owing to randomly dealt cards and stochastic choices made by the players. The better player may lose in a short match simply because they were dealt weaker hands or their rare bluffs were made at inopportune times. As seen in the Claudico match (*20*), even 80,000 games may not be enough to statistically significantly separate players whose skill differs by a considerable margin. We evaluate performance using AIVAT (*37*), a provably unbiased low-variance technique for evaluating performance in imperfect information games based on carefully constructed control variates. AIVAT requires an estimated value of holding each hand in each public state, and then uses the expected value changes that occur due to chance actions and actions of
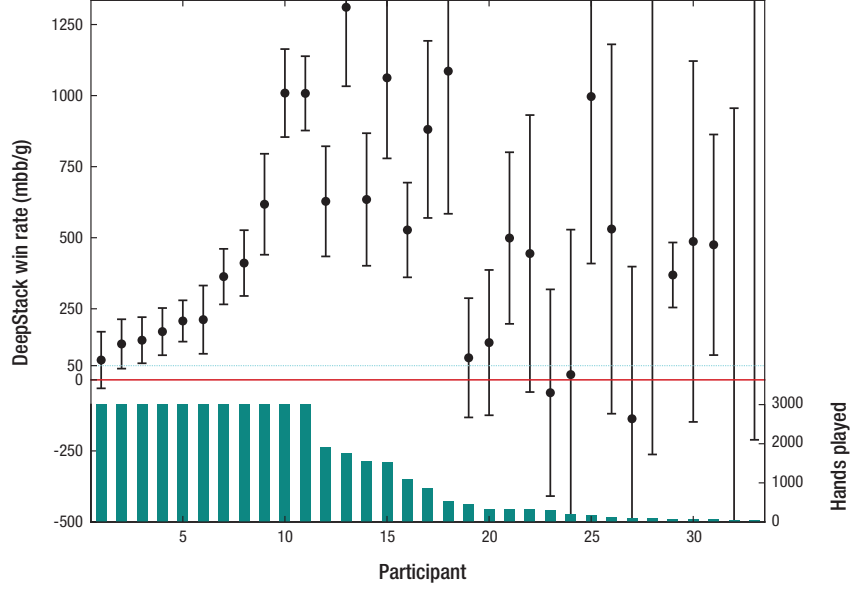
9

Figure 4: **Performance of professional poker players against DeepStack.** Performance estimated with AIVAT along with a 95% confidence interval. The solid bars at the bottom show the number of games the participant completed.

players with known strategies (i.e., DeepStack) to compute the control variate. DeepStack's own value function estimate is perfectly suited for AIVAT. Indeed, when used with AIVAT we get an unbiased performance estimate with an impressive 85% reduction in standard deviation. Thanks to this technique, we can show statistical significance (*38*) in matches with as few as 3,000 games.

In total 44,852 games were played by the thirty-three players with 11 players completing the requested 3,000 games. Over all games played, DeepStack won 492 mbb/g. This is over 4 standard deviations away from zero, and so, highly significant. Note that professional poker players consider 50 mbb/g a sizable margin. Using AIVAT to evaluate performance, we see DeepStack was overall a bit lucky, with its estimated performance actually 486 mbb/g. However, as a lower variance estimate, this margin is over 20 standard deviations from zero.

The performance of individual participants measured with AIVAT is summarized in Figure 4. Amongst those players that completed the requested 3,000 games, DeepStack is estimated to be winning by 394 mbb/g, and individually beating 10 out of 11 such players by a statistically significant margin. Only for the best performing player, still estimated to be losing by 70 mbb/g, is the result not statistically significant. More details on the participants and their results are presented in (*10*).

10

Table 1: **Exploitability bounds from Local Best Response.** For all listed programs, the value reported is the largest estimated exploitability when applying LBR using a variety of different action sets. Table S2 gives a more complete presentation of these results (*10*). ‡: LBR was unable to identify a positive lower bound for DeepStack's exploitability.

Program	LBR (mbb/g)
Hyperborean (2014)	4675
Slumbot (2016)	4020
Act1 (2016)	3302
Always Fold	750
DeepStack	**0** ‡

Exploitability

The main goal of DeepStack is to approximate Nash equilibrium play, i.e., minimize exploitability. While the exact exploitability of a HUNL poker strategy is intractable to compute, the recent local best-response technique (LBR) can provide a lower bound on a strategy's exploitability (*21*) given full access to its action probabilities. LBR uses the action probabilities to compute the strategy's range at any public state. Using this range it chooses its response action from a fixed set using the assumption that no more bets will be placed for the remainder of the game. Thus it best-responds locally to the opponent's actions, providing a lower bound on their overall exploitability. As already noted, abstraction-based programs from the Annual Computer Poker Competition are highly exploitable by LBR: four times more exploitable than folding every game (Table 1). However, even under a variety of settings, LBR fails to exploit DeepStack at all — itself losing by over 350 mbb/g to DeepStack (*10*). Either a more sophisticated lookahead is required to identify DeepStack's weaknesses or it is substantially less exploitable.

Discussion

DeepStack defeated professional poker players at HUNL with statistical significance (*39*), a game that is similarly sized to go, but with the added complexity of imperfect information. It achieves this goal with little domain knowledge and no training from expert human games. The implications go beyond being a milestone for artificial intelligence. DeepStack represents a paradigm shift in approximating solutions to large, sequential imperfect information games. Abstraction and offline computation of complete strategies has been the dominant approach for almost 20 years (*33, 40, 41*). DeepStack allows computation to be focused on specific situations that arise when making decisions and the use of automatically trained value functions. These are two of the core principles that have powered successes in perfect information games, albeit conceptually simpler to implement in those settings. As a result, the gap between the largest

perfect and imperfect information games to have been mastered is mostly closed.

With many real world problems involving information asymmetry, DeepStack also has implications for seeing powerful AI applied more in settings that do not fit the perfect information assumption. The abstraction paradigm for handling imperfect information has shown promise in applications like defending strategic resources (*42*) and robust decision making as needed for medical treatment recommendations (*43*). DeepStack's continual re-solving paradigm will hopefully open up many more possibilities.

References and Notes

1. G. Tesauro, *Communications of the ACM* **38**, 58 (1995).

2. J. Schaeffer, R. Lake, P. Lu, M. Bryant, *AI Magazine* **17**, 21 (1996).

3. M. Campbell, A. J. Hoane Jr., F. Hsu, *Artificial Intelligence* **134**, 57 (2002).

4. D. Ferrucci, *IBM Journal of Research and Development* **56**, 1:1 (2012).

5. V. Mnih, *et al.*, *Nature* **518**, 529 (2015).

6. D. Silver, *et al.*, *Nature* **529**, 484 (2016).

7. A. L. Samuel, *IBM Journal of Research and Development* **3**, 210 (1959).

8. L. Kocsis, C. Szepesvári, *Proceedings of the Seventeenth European Conference on Machine Learning* (2006), pp. 282–293.

9. J. Bronowski, The ascent of man, Documentary (1973). Episode 13.

10. See Supplementary Materials.

11. In 2008, Polaris defeated a team of professional poker players in heads-up limit Texas hold'em (*44*). In 2015, Cepheus essentially solved the game (*18*).

12. V. L. Allis, Searching for solutions in games and artificial intelligence, Ph.D. thesis, University of Limburg (1994).

13. M. Johanson, Measuring the size of large no-limit poker games, *Technical Report TR13-01*, Department of Computing Science, University of Alberta (2013).

14. M. Zinkevich, M. Johanson, M. Bowling, C. Piccione, *Advances in Neural Information Processing Systems 20* (2008), pp. 905–912.

15. A. Gilpin, S. Hoda, J. Peña, T. Sandholm, *Proceedings of the Third International Workshop On Internet And Network Economics* (2007), pp. 57–69.

16. End-game solving (*17, 45, 46*) is one exception to computation occurring prior to play. When the game nears the end, a new computation is invoked over the remainder of the game. Thus, the program need not store this part of the strategy or can use a finer-grained abstraction aimed to improve the solution quality. We discuss this as re-solving when we introduce DeepStack's technique of continual re-solving.

17. N. Burch, M. Johanson, M. Bowling, *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence* (2014), pp. 602–608.

18. M. Bowling, N. Burch, M. Johanson, O. Tammelin, *Science* **347**, 145 (2015).

19. We use milli-big-blinds per game (mbb/g) to measure performance in poker, where a milli-big-blind is one thousandth of the forced big blind bet amount that starts the game. This normalizes performance for the number of games played and the size of stakes. For comparison, a win rate of 50 mbb/g is considered a sizable margin by professional players and 750 mbb/g is the rate that would be lost if a player folded each game. The poker community commonly uses big blinds per one hundred games (bb/100) to measure win rates, where 10 mbb/g equals 1 bb/100.

20. J. Wood, Doug Polk and team beat Claudico to win $100,000 from Microsoft & The Rivers Casino, *Pokerfuse,* http://pokerfuse.com/news/media-and-software/26854-doug-polk-and-team-beat-claudico-win-100000-microsoft/ (2015).

21. V. Lisý, M. Bowling, *Proceedings of the AAAI-17 Workshop on Computer Poker and Imperfect Information Games* (2017). https://arxiv.org/abs/1612.07547.

22. DeepStack is not the first application of deep learning to the game of poker. Previous applications of deep learning, though, are either not known to be theoretically sound (*47*), or have only been applied in small games (*48*).

23. J. F. Nash, *Proceedings of the National Academy of Sciences* **36**, 48 (1950).

24. When the previous solution is an approximation, rather than an exact Nash equilibrium, sound re-solving needs the expected values from a best-response to the player's previous strategy. In practice, though, using expected values from the previous solution in self-play often works as well or better than best-response values (*10*).

25. A. Gilpin, T. Sandholm, T. B. Sørensen, *Proceedings of the Seventh International Conference on Autonomous Agents and Multi-Agent Systems* (2008), pp. 911–918.

26. D. Schnizlein, M. Bowling, D. Szafron, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence* (2009), pp. 278–284.

27. A. Gilpin, T. Sandholm, T. B. Sørensen, *Proceedings of the Twenty-Second Conference on Artificial Intelligence* (2007), pp. 50–57.

28. M. Johanson, N. Burch, R. Valenzano, M. Bowling, *Proceedings of the Twelfth International Conference on Autonomous Agents and Multi-Agent Systems* (2013), pp. 271–278.

29. A. Krizhevsky, I. Sutskever, G. E. Hinton, *Advances in Neural Information Processing Systems 25* (2012), pp. 1106–1114.

30. G. Hinton, *et al.*, *IEEE Signal Processing Magazine* **29**, 82 (2012).

31. A. van den Oord, *et al.*, *CoRR* **abs/1609.03499** (2016).

32. K. He, X. Zhang, S. Ren, J. Sun, *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)* (2015), pp. 1026–1034.

33. J. Shi, M. L. Littman, *Proceedings of the Second International Conference on Computers and Games* (2000), pp. 333–345.

34. D. P. Kingma, J. Ba, *Proceedings of the Third International Conference on Learning Representations* (2014).

35. P. J. Huber, *Annals of Mathematical Statistics* **35**, 73 (1964).

36. International Federation of Poker. http://pokerfed.org/about/ (Accessed January 1, 2017).

37. N. Burch, M. Schmid, M. Moravcik, M. Bowling, *Proceedings of the AAAI-17 Workshop on Computer Poker and Imperfect Information Games* (2017). http://arxiv.org/abs/1612.06915.

38. Statistical significance where noted was established using a two-tailed Student's t-test at the 0.05 level with $N \geq 3000$.

39. Subsequent to our study, the computer program Libratus, developed at CMU by Tuomas Sandholm and Noam Brown, defeated a team of four professional heads-up poker specialists in a HUNL competition held January 11-30, 2017. Libratus has been described as using "nested endgame solving" (*49*), a technique with similarities to continual re-solving, but developed independently. Libratus employs this re-solving when close to the end of the game rather than at every decision, while using an abstraction-based approach earlier in the game.

40. D. Billings, *et al.*, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (2003), pp. 661–668.

41. T. Sandholm, *AI Magazine* **31**, 13 (2010).

42. V. Lisy, T. Davis, M. Bowling, *Proceedings of the Thirtieth Conference on Artificial Intelligence* (2016), pp. 544–550.

43. K. Chen, M. Bowling, *Advances in Neural Information Processing Systems 25* (2012), pp. 2078–2086.

44. J. Rehmeyer, N. Fox, R. Rico, *Wired* **16.12**, 186 (2008).

45. S. Ganzfried, T. Sandholm, *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multi-Agent Systems* (2015), pp. 37–45.

46. M. Moravcik, M. Schmid, K. Ha, M. Hladík, S. J. Gaukrodger, *Proceedings of the Thirtieth Conference on Artificial Intelligence* (2016), pp. 572–578.

47. N. Yakovenko, L. Cao, C. Raffel, J. Fan, *Proceedings of the Thirtieth Conference on Artificial Intelligence* (2016), pp. 360–367.

48. J. Heinrich, D. Silver, *arXiv preprint arXiv:1603.01121* (2016).

49. N. Brown, T. Sandholm, *Proceedings of the AAAI-17 Workshop on Computer Poker and Imperfect Information Games* (2017).

50. M. Zinkevich, M. Littman, *Journal of the International Computer Games Association* **29**, 166 (2006). News item.

51. D. Morrill, Annual computer poker competition poker GUI client, https://github.com/dmorrill10/acpc_poker_gui_client/tree/v1.2 (2012).

52. O. Tammelin, N. Burch, M. Johanson, M. Bowling, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence* (2015), pp. 645–652.

53. R. Collobert, K. Kavukcuoglu, C. Farabet, *BigLearn, NIPS Workshop* (2011).

54. S. Ganzfried, T. Sandholm, *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence* (2014), pp. 682–690.

Supplementary Materials for
DeepStack: Expert-Level AI in No-Limit Poker

Game of Heads-Up No-Limit Texas Hold'em

Heads-up no-limit Texas hold'em (HUNL) is a two-player poker game. It is a repeated game, in which the two players play a match of individual games, usually called hands, while alternating who is the dealer. In each of the individual games, one player will win some number of chips from the other player, and the goal is to win as many chips as possible over the course of the match.

Each individual game begins with both players placing a number of chips in the pot: the player in the dealer position puts in the small blind, and the other player puts in the big blind, which is twice the small blind amount. During a game, a player can only wager and win up to a fixed amount known as their stack. In the particular format of HUNL used in the Annual Computer Poker Competition (*50*) and this article, the big blind is 100 chips and the stack is 20,000 chips or 200 big blinds. Resetting the stacks after each game is called "Doyle's Game", named for the professional poker player Doyle Brunson who publicized this variant (*25*). It is used in the Annual Computer Poker Competitions because it allows for each game to be an independent sample of the same game.

A game of HUNL progresses through four rounds: the pre-flop, flop, turn, and river. Each round consists of cards being dealt followed by player actions in the form of wagers as to who will hold the strongest hand at the end of the game. In the pre-flop, each player is given two private cards, unobserved by their opponent. In the later rounds, cards are dealt face-up in the center of the table, called public cards. A total of five public cards are revealed over the four rounds: three on the flop, one on the turn, and one on the river.

After the cards for the round are dealt, players alternate taking actions of three types: fold, call, or raise. A player folds by declining to match the last opponent wager, thus forfeiting to the opponent all chips in the pot and ending the game with no player revealing their private cards. A player calls by adding chips into the pot to match the last opponent wager, which causes the next round to begin. A player raises by adding chips into the pot to match the last wager followed by adding additional chips to make a wager of their own. At the beginning of a round when there is no opponent wager yet to match, the raise action is called bet, and the call action is called check, which only ends the round if both players check. An all-in wager is one involving all of the chips remaining the player's stack. If the wager is called, there is no further wagering in later rounds. The size of any other wager can be any whole number of chips remaining in the player's stack, as long as it is not smaller than the last wager in the current round or the big blind.

The dealer acts first in the pre-flop round and must decide whether to fold, call, or raise the opponent's big blind bet. In all subsequent rounds, the non-dealer acts first. If the river round ends with no player previously folding to end the game, the outcome is determined by a showdown. Each player reveals their two private cards and the player that can form the strongest five-card poker hand (see "List of poker hand categories" on Wikipedia; accessed January 1, 2017) wins all the chips in the pot. To form their hand each player may use any cards from their two private cards and the five public cards. At the end of the game, whether ended by fold or showdown, the players will swap who is the dealer and begin the next game.

Since the game can be played for different stakes, such as a big blind being worth $0.01 or $1 or $1000, players commonly measure their performance over a match as their average number of big blinds won per game. Researchers have standardized on the unit milli-big-blinds per game, or mbb/g, where one milli-big-blind is one thousandth of one big blind. A player that always folds will lose 750 mbb/g (by losing 1000 mbb as the big blind and 500 as the small blind). A human rule-of-thumb is that a professional should aim to win at least 50 mbb/g from their opponents. Milli-big-blinds per game is also used as a unit of exploitability, when it is computed as the expected loss per game against a worst-case opponent. In the poker community, it is common to use big blinds per one hundred games (bb/100) to measure win rates, where 10 mbb/g equals 1 bb/100.

Poker Glossary

all-in A wager of the remainder of a player's stack. The opponent's only response can be call or fold.

bet The first wager in a round; putting more chips into the pot.

big blind Initial wager made by the non-dealer before any cards are dealt. The big blind is twice the size of the small blind.

call Putting enough chips into the pot to match the current wager; ends the round.

check Declining to wager any chips when not facing a bet.

chip Marker representing value used for wagers; all wagers must be a whole numbers of chips.

dealer The player who puts the small blind into the pot. Acts first on round 1, and second on the later rounds. Traditionally, they would distribute public and private cards from the deck.

flop The second round; can refer to either the 3 revealed public cards, or the betting round after these cards are revealed.

fold Give up on the current game, forfeiting all wagers placed in the pot. Ends a player's participation in the game.

hand Many different meanings: the combination of the best 5 cards from the public cards and private cards, just the private cards themselves, or a single game of poker (for clarity, we avoid this final meaning).

milli-big-blinds per game (mbb/g) Average winning rate over a number of games, measured in thousandths of big blinds.

pot The collected chips from all wagers.

pre-flop The first round; can refer to either the hole cards, or the betting round after these cards are distributed.

private cards Cards dealt face down, visible only to one player. Used in combination with public cards to create a hand. Also called hole cards.

public cards Cards dealt face up, visible to all players. Used in combination with private cards to create a hand. Also called community cards.

raise Increasing the size of a wager in a round, putting more chips into the pot than is required to call the current bet.

river The fourth and final round; can refer to either the 1 revealed public card, or the betting round after this card is revealed.

showdown After the river, players who have not folded show their private cards to determine the player with the best hand. The player with the best hand takes all of the chips in the pot.

small blind Initial wager made by the dealer before any cards are dealt. The small blind is half the size of the big blind.

stack The maximum amount of chips a player can wager or win in a single game.

turn The third round; can refer to either the 1 revealed public card, or the betting round after this card is revealed.

Performance Against Professional Players

To assess DeepStack relative to expert humans, players were recruited with assistance from the International Federation of Poker (*36*) to identify and recruit professional poker players

through their member nation organizations. We only selected participants from those who self-identified as a "professional poker player" during registration. Players were given four weeks to complete a 3,000 game match. To incentivize players, monetary prizes of $5,000, $2,500, and $1,250 (CAD) were awarded to the top three players (measured by AIVAT) that completed their match. The participants were informed of all of these details when they registered to participate. Matches were played between November 7th and December 12th, 2016, and run using an online user interface (*51*) where players had the option to play up to four games simultaneously as is common in online poker sites. A total of 33 players from 17 countries played against DeepStack. DeepStack's performance against each individual is presented in Table 2, with complete game histories available as part of the supplementary online materials.

Local Best Response of DeepStack

The goal of DeepStack, and much of the work on AI in poker, is to approximate a Nash equilibrium, i.e., produce a strategy with low exploitability. The size of HUNL makes an explicit best-response computation intractable and so exact exploitability cannot be measured. A common alternative is to play two strategies against each other. However, head-to-head performance in imperfect information games has repeatedly been shown to be a poor estimation of equilibrium approximation quality. For example, consider an exact Nash equilibrium strategy in the game of Rock-Paper-Scissors playing against a strategy that almost always plays "rock". The results are a tie, but their playing strengths in terms of exploitability are vastly different. This same issue has been seen in heads-up limit Texas hold'em as well (Johanson, IJCAI 2011), where the relationship between head-to-head play and exploitability, which is tractable in that game, is indiscernible. The introduction of local best response (LBR) as a technique for finding a lower-bound on a strategy's exploitability gives evidence of the same issue existing in HUNL. Act1 and Slumbot (second and third place in the previous ACPC) were statistically indistinguishable in head-to-head play (within 20 mbb/g), but Act1 is 1300mbb/g less exploitable as measured by LBR. This is why we use LBR to evaluate DeepStack.

LBR is a simple, yet powerful, technique to produce a lower bound on a strategy's exploitability in HUNL (*21*) . It explores a fixed set of options to find a "locally" good action against the strategy. While it seems natural that more options would be better, this is not always true. More options may cause it to find a locally good action that misses out on a future opportunity to exploit an even larger flaw in the opponent. In fact, LBR sometimes results in larger lower bounds when not considering any bets in the early rounds, so as to increase the size of the pot and thus the magnitude of a strategy's future mistakes. LBR was recently used to show that abstraction-based agents are significantly exploitable (see Table 3). The first three strategies are submissions from recent Annual Computer Poker Competitions. They all use both card and action abstraction and were found to be even more exploitable than simply folding every game in all tested cases. The strategy "Full Cards" does not use any card abstraction, but uses only the sparse fold, call, pot-sized bet, all-in betting abstraction using hard translation (*26*). Due

Table 2: Results against professional poker players estimated with AIVAT (Luck Adjusted Win Rate) and chips won (Unadjusted Win Rate), both measured in mbb/g. Recall 10mbb/g equals 1bb/100. Each estimate is followed by a 95% confidence interval. ‡ marks a participant who completed the 3000 games after their allotted four week period.

Player		Rank	Hands	Luck Adjusted Win Rate	Unadjusted Win Rate
Martin Sturc	🇦🇹	1	3000	70 ± 119	-515 ± 575
Stanislav Voloshin	🇺🇦	2	3000	126 ± 103	-65 ± 648
Prakshat Shrimankar	🇮🇳	3	3000	139 ± 97	174 ± 667
Ivan Shabalin	🇷🇺	4	3000	170 ± 99	153 ± 633
Lucas Schaumann	🇩🇪	5	3000	207 ± 87	160 ± 576
Phil Laak	🇺🇸	6	3000	212 ± 143	774 ± 677
Kaishi Sun	🇨🇳	7	3000	363 ± 116	5 ± 729
Dmitry Lesnoy		8	3000	411 ± 138	-87 ± 753
Antonio Parlavecchio	🇮🇹	9	3000	618 ± 212	1096 ± 962
Muskan Sethi	🇮🇳	10	3000	1009 ± 184	2144 ± 1019
Pol Dmit‡	🇷🇺	–	3000	1008 ± 156	883 ± 793
Tsuneaki Takeda	🇯🇵	–	1901	628 ± 231	-332 ± 1228
Youwei Qin	🇦🇺	–	1759	1311 ± 331	1958 ± 1799
Fintan Gavin	🇮🇪	–	1555	635 ± 278	-26 ± 1647
Giedrius Talacka	🇱🇹	–	1514	1063 ± 338	459 ± 1707
Juergen Bachmann	🇩🇪	–	1088	527 ± 198	1769 ± 1662
Sergey Indenok	🇷🇺	–	852	881 ± 371	253 ± 2507
Sebastian Schwab	🇩🇪	–	516	1086 ± 598	1800 ± 2162
Dara O'Kearney	🇮🇪	–	456	78 ± 250	223 ± 1688
Roman Shaposhnikov	🇷🇺	–	330	131 ± 305	-898 ± 2153
Shai Zurr	🇮🇱	–	330	499 ± 360	1154 ± 2206
Luca Moschitta	🇮🇹	–	328	444 ± 580	1438 ± 2388
Stas Tishekvich	🇮🇱	–	295	-45 ± 433	-346 ± 2264
Eyal Eshkar	🇮🇱	–	191	18 ± 608	715 ± 4227
Jefri Islam	🇦🇹	–	176	997 ± 700	3822 ± 4834
Fan Sun	🇨🇳	–	122	531 ± 774	-1291 ± 5456
Igor Naumenko	🇺🇦	–	102	-137 ± 638	851 ± 1536
Silvio Pizzarello	🇮🇹	–	90	1500 ± 2100	5134 ± 6766
Gaia Freire	🇧🇷	–	76	369 ± 136	138 ± 694
Alexander Bös	🇦🇹	–	74	487 ± 756	1 ± 2628
Victor Santos	🇧🇷	–	58	475 ± 462	-1759 ± 2571
Mike Phan	🇦🇷	–	32	-1019 ± 2352	-11223 ± 18235
Juan Manuel Pastor	🇪🇸	–	7	2744 ± 3521	7286 ± 9856
Human Professionals			44852	486 ± 40	492 ± 220

Table 3: Exploitability lower bound of different programs using local best response (LBR). LBR evaluates only the listed actions in each round as shown in each row. F, C, P, A, refer to fold, call, a pot-sized bet, and all-in, respectively. 56bets includes the actions fold, call and 56 equidistant pot fractions as defined in the original LBR paper (*21*). ‡: Always Fold checks when not facing a bet, and so it cannot be maximally exploited without a betting action.

		Local best response performance (mbb/g)			
	Pre-flop	F, C	C	C	C
LBR Settings	Flop	F, C	C	C	56bets
	Turn	F, C	F, C, P, A	56bets	F, C
	River	F, C	F, C, P, A	56bets	F, C
Hyperborean (2014)		721 ± 56	3852 ± 141	4675 ± 152	983 ± 95
Slumbot (2016)		522 ± 50	4020 ± 115	3763 ± 104	1227 ± 79
Act1 (2016)		407 ± 47	2597 ± 140	3302 ± 122	847 ± 78
Always Fold		$‡250 \pm 0$	750 ± 0	750 ± 0	750 ± 0
Full Cards [100 BB]		-424 ± 37	-536 ± 87	2403 ± 87	1008 ± 68
DeepStack		-428 ± 87	-383 ± 219	-775 ± 255	-602 ± 214

to computation and memory requirements, we computed this strategy only for a smaller stack of 100 big blinds. Still, this strategy takes almost 2TB of memory and required approximately 14 CPU years to solve. Naturally, it cannot be exploited by LBR within the betting abstraction, but it is heavily exploitable in settings using other betting actions that require it to translate its opponent's actions, again losing more than if it folded every game.

As for DeepStack, under all tested settings of LBR's available actions, it fails to find any exploitable flaw. In fact, it is losing 350 mbb/g or more to DeepStack. Of particular interest is the final column aimed to exploit DeepStack's flop strategy. The flop is where DeepStack is most dependent on its counterfactual value networks to provide it estimates through the end of the game. While these experiments do not prove that DeepStack is flawless, it does suggest its flaws require a more sophisticated search procedure than what is needed to exploit abstraction-based programs.

DeepStack Implementation Details

Here we describe the specifics for how DeepStack employs continual re-solving and how its deep counterfactual value networks were trained.

Table 4: Lookahead re-solving specifics by round. The abbreviations of F, C, ½P, P, 2P, and A refer to fold, call, half of a pot-sized bet, a pot-sized bet, twice a pot-sized bet, and all in, respectively. The final column specifies which neural network was used when the depth limit was exceeded: the flop, turn, or the auxiliary network.

Round	CFR Iterations	Omitted Iterations	First Action	Second Action	Remaining Actions	NN Eval
Pre-flop	1000	980	F, C, ½P, P, A	F, C, ½P, P, 2P, A	F, C, P, A	Aux/Flop
Flop	1000	500	F, C, ½P, P, A	F, C, P, A	F, C, P, A	Turn
Turn	1000	500	F, C, ½P, P, A	F, C, P, A	F, C, P, A	—
River	2000	1000	F, C, ½P, P, 2P, A	F, C, ½P, P, 2P, A	F, C, P, A	—

Continual Re-Solving

As with traditional re-solving, the re-solving step of the DeepStack algorithm solves an augmented game. The augmented game is designed to produce a strategy for the player such that the bounds for the opponent's counterfactual values are satisfied. DeepStack uses a modification of the original CFR-D gadget (*17*) for its augmented game, as discussed below. While the max-margin gadget (*46*) is designed to improve the performance of poor strategies for abstracted agents near the end of the game, the CFR-D gadget performed better in early testing.

The algorithm DeepStack uses to solve the augmented game is a hybrid of vanilla CFR (*14*) and CFR$^+$ (*52*), which uses regret matching$^+$ like CFR$^+$, but does uniform weighting and simultaneous updates like vanilla CFR. When computing the final average strategy and average counterfactual values, we omit the early iterations of CFR in the averages.

A major design goal for DeepStack's implementation was to typically play at least as fast as a human would using commodity hardware and a single GPU. The degree of lookahead tree sparsity and the number of re-solving iterations are the principle decisions that we tuned to achieve this goal. These properties were chosen separately for each round to achieve a consistent speed on each round. Note that DeepStack has no fixed requirement on the density of its lookahead tree besides those imposed by hardware limitations and speed constraints.

The lookahead trees vary in the actions available to the player acting, the actions available for the opponent's response, and the actions available to either player for the remainder of the round. We use the end of the round as our depth limit, except on the turn when the remainder of the game is solved. On the pre-flop and flop, we use trained counterfactual value networks to return values after the flop or turn card(s) are revealed. Only applying our value function to public states at the start of a round is particularly convenient in that that we don't need to include the bet faced as an input to the function. Table 4 specifies lookahead tree properties for each round.

The pre-flop round is particularly expensive as it requires enumerating all 22,100 possible public cards on the flop and evaluating each with the flop network. To speed up pre-flop play, we trained an additional auxiliary neural network to estimate the expected value of the flop network

Table 5: Absolute (L_1), Euclidean (L_2), and maximum absolute (L_∞) errors, in mbb/g, of counterfactual values computed with 1,000 iterations of CFR on sparse trees, averaged over 100 random river situations. The ground truth values were estimated by solving the game with 9 betting options and 4,000 iterations (first row).

Betting	Size	L_1	L_2	L_∞
F, C, Min, ¼P, ½P, ¾P, P, 2P, 3P, 10P, A [4,000 iterations]	555k	0.0	0.0	0.0
F, C, Min, ¼P, ½P, ¾P, P, 2P, 3P, 10P, A	555k	18.06	0.891	0.2724
F, C, 2P, A	48k	64.79	2.672	0.3445
F, C, ½P, A	100k	58.24	3.426	0.7376
F, C, P, A	61k	25.51	1.272	0.3372
F, C, ½P, P, A	126k	41.42	1.541	0.2955
F, C, P, 2P, A	204k	27.69	1.390	0.2543
F, C, ½P, P, 2P, A	360k	20.96	1.059	0.2653

over all possible flops. However, we only used this network during the initial omitted iterations of CFR. During the final iterations used to compute the average strategy and counterfactual values, we did the expensive enumeration and flop network evaluations. Additionally, we cache the re-solving result for every observed pre-flop situation. When the same betting sequence occurs again, we simply reuse the cached results rather than recomputing. For the turn round, we did not use a neural network after the final river card, but instead solved to the end of the game. However, we used a bucketed abstraction for all actions on the river. For acting on the river, the re-solving includes the remainder of the game and so no counterfactual value network was used.

Actions in Sparse Lookahead Trees. DeepStack's sparse lookahead trees use only a small subset of the game's possible actions. The first layer of actions immediately after the current public state defines the options considered for DeepStack's next action. The only purpose of the remainder of the tree is to estimate counterfactual values for the first layer during the CFR algorithm. Table 5 presents how well counterfactual values can be estimated using sparse lookahead trees with various action subsets.

The results show that the F, C, P, A, actions provide an excellent tradeoff between computational requirements via the size of the solved lookahead tree and approximation quality. Using more actions quickly increases the size of the lookahead tree, but does not substantially improve errors. Alternatively, using a single betting action that is not one pot has a small effect on the size of the tree, but causes a substantial error increase.

To further investigate the effect of different betting options, Table 6 presents the results of evaluating DeepStack with different action sets using LBR. We used setting of LBR that proved most effective against the default set of DeepStack actions (see Table 4). While the extent of the variance in the 10,000 hand evaluation shown in Table 6 prevents us from declaring a best

Table 6: Performance of LBR exploitation of DeepStack with different actions allowed on the first level of its lookahead tree using the best LBR configuration against the default version of DeepStack. LBR cannot exploit DeepStack regardless of its available actions.

First level actions	LBR performance
F, C, P, A	-479 ± 216
Default	-383 ± 219
F, C, ½P, P, 1½P, 2P, A	-406 ± 218

set of actions with certainty, the crucial point is that LBR is significantly losing to each of them, and that we can produce play that is difficult to exploit even choosing from a small number of actions. Furthermore, the improvement of a small number of additional actions is not dramatic.

Opponent Ranges in Re-Solving. Continual re-solving does not require keeping track of the opponent's range. The re-solving step essentially reconstructs a suitable range using the bounded counterfactual values. In particular, the CFR-D gadget does this by giving the opponent the option, after being dealt a uniform random hand, of terminating the game (T) instead of following through with the game (F), allowing them to simply earn that hand's bound on its counterfactual value. Only hands which are valuable to bring into the subgame will then be observed by the re-solving player. However, this process of the opponent learning which hands to follow through with can make re-solving require many iterations. An estimate of the opponent's range can be used to effectively warm-start the choice of opponent ranges, and help speed up the re-solving.

One conservative option is to replace the uniform random deal of opponent hands with any distribution over hands as long as it assigns non-zero probability to every hand. For example, we could linearly combine an estimated range of the opponent from the previous re-solve (with weight b) and a uniform range (with weight $1 - b$). This approach still has the same theoretical guarantees as re-solving, but can reach better approximate solutions in fewer iterations. Another option is more aggressive and sacrifices the re-solving guarantees when the opponent's range estimate is wrong. It forces the opponent with probability b to follow through into the game with a hand sampled from the estimated opponent range. With probability $1 - b$ they are given a uniform random hand and can choose to terminate or follow through. This could prevent the opponent's strategy from reconstructing a correct range, but can speed up re-solving further when we have a good opponent range estimate.

DeepStack uses an estimated opponent range during re-solving only for the first action of a round, as this is the largest lookahead tree to re-solve. The range estimate comes from the last re-solve in the previous round. When DeepStack is second to act in the round, the opponent has already acted, biasing their range, so we use the conservative approach. When DeepStack is first to act, though, the opponent could only have checked or called since our last re-solve. Thus, the lookahead has an estimated range following their action. So in this case, we use the

24

Table 7: Thinking times for both humans and DeepStack. DeepStack's extremely fast pre-flop speed shows that pre-flop situations often resulted in cache hits.

| | Thinking Time (s) | | | |
| | Humans | | DeepStack | |
Round	Median	Mean	Median	Mean
Pre-flop	10.3	16.2	0.04	0.2
Flop	9.1	14.6	5.9	5.9
Turn	8.0	14.0	5.4	5.5
River	9.5	16.2	2.2	2.1
Per Action	9.6	15.4	2.3	3.0
Per Hand	22.0	37.4	5.7	7.2

aggressive approach. In both cases, we set $b = 0.9$.

Speed of Play. The re-solving computation and neural network evaluations are both implemented in Torch7 (*53*) and run on a single NVIDIA GeForce GTX 1080 graphics card. This makes it possible to do fast batched calls to the counterfactual value networks for multiple public subtrees at once, which is key to making DeepStack fast.

Table 7 reports the average times between the end of the previous (opponent or chance) action and submitting the next action by both humans and DeepStack in our study. DeepStack, on average, acted considerably faster than our human players. It should be noted that some human players were playing up to four games simultaneously (although few players did more than two), and so the human players may have been focused on another game when it became their turn to act.

Deep Counterfactual Value Networks

DeepStack uses two counterfactual value networks, one for the flop and one for the turn, as well as an auxiliary network that gives counterfactual values at the end of the pre-flop. In order to train the networks, we generated random poker situations at the start of the flop and turn. Each poker situation is defined by the pot size, ranges for both players, and dealt public cards. The complete betting history is not necessary as the pot and ranges are a sufficient representation. The output of the network are vectors of counterfactual values, one for each player. The output values are interpreted as fractions of the pot size to improve generalization across poker situations.

The training situations were generated by first sampling a pot size from a fixed distribution which was designed to approximate observed pot sizes from older HUNL programs.[2] The

---

[2]The fixed distribution selects an interval from the set of intervals $\{[100, 100), [200, 400), [400, 2000),$

player ranges for the training situations need to cover the space of possible ranges that CFR might encounter during re-solving, not just ranges that are likely part of a solution. So we generated pseudo-random ranges that attempt to cover the space of possible ranges. We used a recursive procedure $R(S, p)$, that assigns probabilities to the hands in the set S that sum to probability p, according to the following procedure.

1. If $|S| = 1$, then $\Pr(s) = p$.

2. Otherwise,

   (a) Choose p_1 uniformly at random from the interval $(0, p)$, and let $p_2 = p - p_1$.
   (b) Let $S_1 \subset S$ and $S_2 = S \setminus S_1$ such that $|S_1| = \lfloor |S|/2 \rfloor$ and all of the hands in S_1 have a hand strength no greater than hands in S_2. Hand strength is the probability of a hand beating a uniformly selected random hand from the current public state.
   (c) Use $R(S_1, p_1)$ and $R(S_2, p_2)$ to assign probabilities to hands in $S = S_1 \bigcup S_2$.

Generating a range involves invoking $R(\textit{all hands}, 1)$. To obtain the target counterfactual values for the generated poker situations for the main networks, the situations were approximately solved using 1,000 iterations of CFR$^+$ with only betting actions fold, call, a pot-sized bet, and all-in. For the turn network, ten million poker turn situations (from after the turn card is dealt) were generated and solved with 6,144 CPU cores of the Calcul Québec MP2 research cluster, using over 175 core years of computation time. For the flop network, one million poker flop situations (from after the flop cards are dealt) were generated and solved. These situations were solved using DeepStack's depth limited solver with the turn network used for the counterfactual values at public states immediately after the turn card. We used a cluster of 20 GPUS and one-half of a GPU year of computation time. For the auxiliary network, ten million situations were generated and the target values were obtained by enumerating all 22,100 possible flops and averaging the counterfactual values from the flop network's output.

Neural Network Training. All networks were trained using built-in Torch7 libraries, with the Adam stochastic gradient descent procedure (*34*) minimizing the average of the Huber losses (*35*) over the counterfactual value errors. Training used a mini-batch size of 1,000, and a learning rate 0.001, which was decreased to 0.0001 after the first 200 epochs. Networks were trained for approximately 350 epochs over two days on a single GPU, and the epoch with the lowest validation loss was chosen.

Neural Network Range Representation. In order to improve generalization over input player ranges, we map the distribution of individual hands (combinations of public and private cards) into distributions of buckets. The buckets were generated using a clustering-based abstraction

---

$[2000, 6000), [6000, 19950]\}$ with uniform probability, followed by uniformly selecting an integer from within the chosen interval.
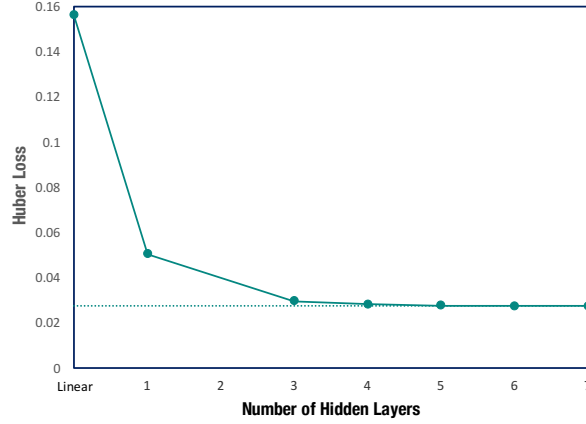
Figure 5: Huber loss with different numbers of hidden layers in the neural network.

technique, which cluster strategically similar hands using k-means clustering with earth mover's distance over hand-strength-like features (*28, 54*). For both the turn and flop networks we used 1,000 clusters and map the original ranges into distributions over these clusters as the first layer of the neural network (see Figure 3 of the main article). This bucketing step was not used on the auxiliary network as there are only 169 strategically distinct hands pre-flop, making it feasible to input the distribution over distinct hands directly.

Neural Network Accuracies.   The turn network achieved an average Huber loss of 0.016 of the pot size on the training set and 0.026 of the pot size on the validation set. The flop network, with a much smaller training set, achieved an average Huber loss of 0.008 of the pot size on the training set, but 0.034 of the pot size on the validation set. Finally, the auxiliary network had average Huber losses of 0.000053 and 0.000055 on the training and validation set, respectively. Note that there are, in fact, multiple Nash equilibrium solutions to these poker situations, with each giving rise to different counterfactual value vectors. So, these losses may overestimate the true loss as the network may accurately model a different equilibrium strategy.

Number of Hidden Layers.   We observed in early experiments that the neural network had a lower validation loss with an increasing number of hidden layers. From these experiments, we chose to use seven hidden layers in an attempt to tradeoff accuracy, speed of execution, and the available memory on the GPU. The result of a more thorough experiment examining the turn network accuracy as a function of the number of hidden layers is in Figure 5. It appears that seven hidden layers is more than strictly necessary as the validation error does not improve much beyond five. However, all of these architectures were trained using the same ten million turn situations. With more training data it would not be surprising to see the larger networks see a further reduction in loss due to their richer representation power.

Proof of Theorem 1

The formal proof of Theorem 1, which establishes the soundness of DeepStack's depth-limited continual re-solving, is conceptually easy to follow. It requires three parts. First, we establish that the exploitability introduced in a re-solving step has two linear components; one due to approximately solving the subgame, and one due to the error in DeepStack's counterfactual value network (see Lemmas 1 through 5). Second, we enable estimates of subgame counterfactual values that do not arise from actual subgame strategies (see Lemma 6). Together, parts one and two enable us to use DeepStack's counterfactual value network for a single re-solve.[3] Finally, we show that using the opponent's values from the best action, rather than the observed action, does not increase overall exploitability (see Lemma 7). This allows us to carry forward estimates of the opponent's counterfactual value, enabling continual re-solving. Put together, these three parts bound the error after any finite number of continual re-solving steps, concluding the proof. We now formalize each step.

There are a number of concepts we use throughout this section. We use the notation from Burch et al. (*17*) without any introduction here. We assume player 1 is performing the continual re-solving. We call player 2 the opponent. We only consider the re-solve player's strategy σ, as the opponent is always using a best response to σ. All values are considered with respect to the opponent, unless specifically stated. We say σ is ϵ-exploitable if the opponent's best response value against σ is no more than ϵ away from the game value for the opponent.

A public state S corresponds to the root of an imperfect information subgame. We write $\mathcal{I}_2^S$ for the collection of player 2 information sets in S. Let $G\langle S, \sigma, w \rangle$ be the subtree gadget game (the re-solving game of Burch et al. (*17*)), where S is some public state, σ is used to get player 1 reach probabilities $\pi_{-2}^\sigma(h)$ for each $h \in S$, and w is a vector where w_I gives the value of player 2 taking the terminate action (T) from information set $I \in \mathcal{I}_2^S$. Let

$$\mathrm{BV}_I(\sigma) = \max_{\sigma_2^*} \sum_{h \in I} \pi_{-2}^\sigma(h) u^{\sigma, \sigma_2^*}(h) / \pi_{-2}^\sigma(I),$$

be the counterfactual value for I given we play σ and our opponent is playing a best response. For a subtree strategy σ^S, we write $\sigma \to \sigma^S$ for the strategy that plays according to σ^S for any state in the subtree and according to σ otherwise. For the gadget game $G\langle S, \sigma, w \rangle$, the gadget value of a subtree strategy σ^S is defined to be:

$$\mathrm{GV}_{w,\sigma}^S(\sigma^S) = \sum_{I \in \mathcal{I}_2^S} \max(w_I, \mathrm{BV}_I(\sigma \to \sigma^S)),$$

and the underestimation error is defined to be:

$$\mathrm{U}_{w,\sigma}^S = \min_{\sigma^S} \mathrm{GV}_{w,\sigma}^S(\sigma^S) - \sum_{I \in \mathcal{I}_2^S} w_I.$$

---

[3] The first part is a generalization and improvement on the re-solving exploitability bound given by Theorem 3 in Burch et al. (*17*), and the second part generalizes the bound on decomposition regret given by Theorem 2 of the same work.

Lemma 1 *The game value of a gadget game $G\langle S, \sigma, w \rangle$ is*

$$\sum_{I \in \mathcal{I}_2^S} w_I + U_{w,\sigma}^S.$$

Proof. Let $\tilde{\sigma}_2^S$ be a gadget game strategy for player 2 which must choose from the F and T actions at starting information set I. Let $\tilde{u}$ be the utility function for the gadget game.

$$\min_{\sigma_1^S} \max_{\tilde{\sigma}_2^S} \tilde{u}(\sigma_1^S, \tilde{\sigma}_2^S) = \min_{\sigma_1^S} \max_{\sigma_2^S} \sum_{I \in \mathcal{I}_2^S} \frac{\pi_{-2}^\sigma(I)}{\sum_{I' \in \mathcal{I}_2^S} \pi_{-2}^\sigma(I')} \max_{a \in \{F, T\}} \tilde{u}^{\sigma^S}(I, a)$$

$$= \min_{\sigma_1^S} \max_{\sigma_2^S} \sum_{I \in \mathcal{I}_2^S} \max(w_I, \sum_{h \in I} \pi_{-2}^\sigma(h) u^{\sigma^S}(h))$$

A best response can maximize utility at each information set independently:

$$= \min_{\sigma_1^S} \sum_{I \in \mathcal{I}_2^S} \max(w_I, \max_{\sigma_2^S} \sum_{h \in I} \pi_{-2}^\sigma(h) u^{\sigma^S}(h))$$

$$= \min_{\sigma_1^S} \sum_{I \in \mathcal{I}_2^S} \max(w_I, \mathbf{BV}_I(\sigma \to \sigma_1^S))$$

$$= U_{w,\sigma}^S + \sum_{I \in \mathcal{I}_2^S} w_I$$

∎

Lemma 2 *If our strategy σ^S is ϵ-exploitable in the gadget game $G\langle S, \sigma, w \rangle$, then $GV_{w,\sigma}^S(\sigma^S) \leq \sum_{I \in \mathcal{I}_2^S} w_I + U_{w,\sigma}^S + \epsilon$*

Proof. This follows from Lemma 1 and the definitions of ϵ-Nash, $U_{w,\sigma}^S$, and $GV_{w,\sigma}^S(\sigma^S)$. ∎

Lemma 3 *Given an ϵ_O-exploitable σ in the original game, if we replace a subgame with a strategy σ^S such that $BV_I(\sigma \to \sigma^S) \leq w_I$ for all $I \in \mathcal{I}_2^S$, then the new combined strategy has an exploitability no more than $\epsilon_O + EXP_{w,\sigma}^S$ where*

$$EXP_{w,\sigma}^S = \sum_{I \in \mathcal{I}_2^S} \max(BV_I(\sigma), w_I) - \sum_{I \in \mathcal{I}_2^S} BV_I(\sigma)$$

29

Proof. We only care about the information sets where the opponent's counterfactual value increases, and a worst case upper bound occurs when the opponent best response would reach every such information set with probability 1, and never reach information sets where the value decreased.

Let $Z[S] \subseteq Z$ be the set of terminal states reachable from some $h \in S$ and let v_2 be the game value of the full game for player 2. Let σ_2 be a best response to σ and let σ_2^S be the part of σ_2 that plays in the subtree rooted at S. Then necessarily σ_2^S achieves counterfactual value $\mathrm{BV}_I(\sigma)$ at each $I \in \mathcal{I}_2^S$.

$$
\max_{\sigma_2^*}(u(\sigma \to \sigma^S, \sigma_2^*))
$$

$$
= \max_{\sigma_2^*}\left[\sum_{z \in Z[S]} \pi_{-2}^{\sigma \to \sigma^S}(z)\pi_2^{\sigma_2^*}(z)u(z) + \sum_{z \in Z\setminus Z[S]} \pi_{-2}^{\sigma \to \sigma^S}(z)\pi_2^{\sigma_2^*}(z)u(z) \right]
$$

$$
= \max_{\sigma_2^*}\left[\sum_{z \in Z[S]} \pi_{-2}^{\sigma \to \sigma^S}(z)\pi_2^{\sigma_2^*}(z)u(z) - \sum_{z \in Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^* \to \sigma_2^S}(z)u(z) \right.
$$

$$
\left. + \sum_{z \in Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^* \to \sigma_2^S}(z)u(z) + \sum_{z \in Z\setminus Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^*}(z)u(z) \right]
$$

$$
\leq \max_{\sigma_2^*}\left[\sum_{z \in Z[S]} \pi_{-2}^{\sigma \to \sigma^S}(z)\pi_2^{\sigma_2^*}(z)u(z) - \sum_{z \in Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^* \to \sigma_2^S}(z)u(z) \right]
$$

$$
+ \max_{\sigma_2^*}\left[\sum_{z \in Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^* \to \sigma_2^S}(z)u(z) + \sum_{z \in Z\setminus Z[S]} \pi_{-2}^{\sigma}(z)\pi_2^{\sigma_2^*}(z)u(z) \right]
$$

$$
\leq \max_{\sigma_2^*}\left[\sum_{I \in \mathcal{I}_2^S} \sum_{h \in I} \pi_{-2}^{\sigma}(h)\pi_2^{\sigma_2^*}(h)u^{\sigma^S, \sigma_2^*}(h) \right.
$$

$$
\left. - \sum_{I \in \mathcal{I}_2^S} \sum_{h \in I} \pi_{-2}^{\sigma}(h)\pi_2^{\sigma_2^*}(h)u^{\sigma, \sigma_2^S}(h) \right] + \max_{\sigma_2^*}(u(\sigma, \sigma_2^*))
$$

By perfect recall $\pi_2(h) = \pi_2(I)$ for each $h \in I$:

$$
\leq \max_{\sigma_2^*}\left[\sum_{I \in \mathcal{I}_2^S} \pi_2^{\sigma_2^*}(I)\left(\sum_{h \in I} \pi_{-2}^{\sigma}(h)u^{\sigma^S, \sigma_2^*}(h) - \sum_{h \in I} \pi_{-2}^{\sigma}(h)u^{\sigma, \sigma_2^S}(h) \right) \right]
$$

$$
+ v_2 + \epsilon_O
$$

$$
= \max_{\sigma_2^*}\left[\sum_{I \in \mathcal{I}_2^S} \pi_2^{\sigma_2^*}(I)\pi_{-2}^{\sigma}(I)\left(\mathrm{BV}_I(\sigma \to \sigma^S) - \mathrm{BV}_I(\sigma) \right) \right] + v_2 + \epsilon_O
$$

$$
\leq \left[\sum_{I \in \mathcal{I}_2^S} \max(\mathrm{BV}_I(\sigma \to \sigma^S) - \mathrm{BV}_I(\sigma), 0) \right] + v_2 + \epsilon_O
$$

$$\leq \quad \left[\sum_{I\in\mathcal{I}_2^S} \max(w_I - \mathbf{BV}_I(\sigma), \mathbf{BV}_I(\sigma) - \mathbf{BV}_I(\sigma))\right] + v_2 + \epsilon_O$$

$$= \quad \left[\sum_{I\in\mathcal{I}_2^S} \max(\mathbf{BV}_I(\sigma), w_I) - \sum_{I\in\mathcal{I}_2^S} \mathbf{BV}_I(\sigma)\right] + v_2 + \epsilon_O$$

∎

Lemma 4 *Given an ϵ_O-exploitable σ in the original game, if we replace the strategy in a sub-game with a strategy σ^S that is ϵ_S-exploitable in the gadget game $G\langle S, \sigma, w\rangle$, then the new combined strategy has an exploitability no more than $\epsilon_O + EXP_{w,\sigma}^S + U_{w,\sigma}^S + \epsilon_S$.*

Proof. We use that $\max(a,b) = a + b - \min(a,b)$. From applying Lemma 3 with $w_I = \mathbf{BV}_I(\sigma \to \sigma^S)$ and expanding $\mathrm{EXP}_{\mathbf{BV}(\sigma\to\sigma^S),\sigma}^S$ we get exploitability no more than $\epsilon_O - \sum_{I\in\mathcal{I}_2^S}\mathbf{BV}_I(\sigma)$ plus

$$\sum_{I\in\mathcal{I}_2^S} \max(\mathbf{BV}_I(\sigma \to \sigma^S), \mathbf{BV}_I(\sigma))$$

$$\leq \sum_{I\in\mathcal{I}_2^S} \max(\mathbf{BV}_I(\sigma \to \sigma^S), \max(w_I, \mathbf{BV}_I(\sigma)))$$

$$= \sum_{I\in\mathcal{I}_2^S} \Big(\mathbf{BV}_I(\sigma \to \sigma^S) + \max(w_I, \mathbf{BV}_I(\sigma))$$
$$\qquad - \min(\mathbf{BV}_I(\sigma \to \sigma^S), \max(w_I, \mathbf{BV}_I(\sigma)))\Big)$$

$$\leq \sum_{I\in\mathcal{I}_2^S} \Big(\mathbf{BV}_I(\sigma \to \sigma^S) + \max(w_I, \mathbf{BV}_I(\sigma))$$
$$\qquad - \min(\mathbf{BV}_I(\sigma \to \sigma^S), w_I)\Big)$$

$$= \sum_{I\in\mathcal{I}_2^S} \Big(\max(w_I, \mathbf{BV}_I(\sigma)) + \max(w_I, \mathbf{BV}_I(\sigma \to \sigma^S)) - w_I\Big)$$

$$= \sum_{I\in\mathcal{I}_2^S} \max(w_I, \mathbf{BV}_I(\sigma)) + \sum_{I\in\mathcal{I}_2^S} \max(w_I, \mathbf{BV}_I(\sigma \to \sigma^S)) - \sum_{I\in\mathcal{I}_2^S} w_I$$

From Lemma 2 we get

$$\leq \sum_{I\in\mathcal{I}_2^S} \max(w_I, \mathbf{BV}_I(\sigma)) + \mathrm{U}_{w,\sigma}^S + \epsilon_S$$

Adding $\epsilon_O - \sum_I \mathbf{BV}_I(\sigma)$ we get the upper bound $\epsilon_O + \mathrm{EXP}_{w,\sigma}^S + \mathrm{U}_{w,\sigma}^S + \epsilon_S$. ∎

Lemma 5 *Assume we are performing one step of re-solving on subtree S, with constraint values w approximating opponent best-response values to the previous strategy σ, with an approximation error bound $\sum_I |w_I - BV_I(\sigma)| \leq \epsilon_E$. Then we have $EXP^S_{w,\sigma} + U^S_{w,\sigma} \leq \epsilon_E$.*

Proof. $\text{EXP}^S_{w,\sigma}$ measures the amount that the w_I exceed $\text{BV}_I(\sigma)$, while $\text{U}^S_{w,\sigma}$ bounds the amount that the w_I underestimate $\text{BV}_I(\sigma \to \sigma^S)$ for any σ^S, including the original σ. Thus, together they are bounded by $|w_I - \text{BV}_I(\sigma)|$:

$$
\begin{aligned}
\text{EXP}^S_{w,\sigma} + \text{U}^S_{w,\sigma} &= \sum_{I \in \mathcal{I}^S_2} \max(\text{BV}_I(\sigma), w_I) - \sum_{I \in \mathcal{I}^S_2} \text{BV}_I(\sigma) \\
&\quad + \min_{\sigma^S} \sum_{I \in \mathcal{I}^S_2} \max(w_I, \text{BV}_I(\sigma \to \sigma^S)) - \sum_{I \in \mathcal{I}^S_2} w_I \\
&\leq \sum_{I \in \mathcal{I}^S_2} \max(\text{BV}_I(\sigma), w_I) - \sum_{I \in \mathcal{I}^S_2} \text{BV}_I(\sigma) \\
&\quad + \sum_{I \in \mathcal{I}^S_2} \max(w_I, \text{BV}_I(\sigma)) - \sum_{I \in \mathcal{I}^S_2} w_I \\
&= \sum_{I \in \mathcal{I}^S_2} [\max(w_I - \text{BV}_I(\sigma), 0) + \max(\text{BV}_I(\sigma) - w_I, 0)] \\
&= \sum_{I \in \mathcal{I}^S_2} |w_I - \text{BV}_I(\sigma)| \leq \epsilon_E
\end{aligned}
$$

∎

Lemma 6 *Assume we are solving a game G with T iterations of CFR-D where for both players p, subtrees S, and times t, we use subtree values v_I for all information sets I at the root of S from some suboptimal black box estimator. If the estimation error is bounded, so that $\min_{\sigma^*_S \in NE_S} \sum_{I \in \mathcal{I}^S_2} |v^{\sigma^*_S}(I) - v_I| \leq \epsilon_E$, then the trunk exploitability is bounded by $k_G/\sqrt{T} + j_G \epsilon_E$ for some game specific constant $k_G, j_G \geq 1$ which depend on how the game is split into a trunk and subgames.*

Proof. This follows from a modified version the proof of Theorem 2 of Burch et al. (*17*), which uses a fixed error ϵ and argues by induction on information sets. Instead, we argue by induction on entire public states.

For every public state s, let N_s be the number of subgames reachable from s, including any subgame rooted at s. Let $Succ(s)$ be the set of our public states which are reachable from s without going through another of our public states on the way. Note that if s is in the trunk, then every $s' \in Succ(s)$ is in the trunk or is the root of a subgame. Let $D_{TR}(s)$ be the set of our trunk public states reachable from s, including s if s is in the trunk. We argue that for any public state s where we act in the trunk or at the root of a subgame

$$
\sum_{I \in s} R^{T,+}_{full}(I) \leq \sum_{s' \in D_{TR}(s)} \sum_{I \in s'} R^{T,+}(I) + T N_s \epsilon_E \tag{1}
$$

First note that if no subgame is reachable from s, then $N_s = 0$ and the statement follows from Lemma 7 of (*14*). For public states from which a subgame is reachable, we argue by induction on $|D_{TR}(s)|$.

For the base case, if $|D_{TR}(s)| = 0$ then s is the root of a subgame S, and by assumption there is a Nash Equilibrium subgame strategy σ_S^* that has regret no more than ϵ_E. If we implicitly play σ_S^* on each iteration of CFR-D, we thus accrue $\sum_{I \in s} R_{full}^{T,+}(I) \leq T\epsilon_E$.

For the inductive hypothesis, we assume that (1) holds for all s such that $|D_{TR}(s)| < k$.

Consider a public state s where $|D_{TR}(s)| = k$. By Lemma 5 of (*14*) we have

$$\sum_{I \in s} R_{full}^{T,+}(I) \leq \sum_{I \in s} \left[R^T(I) + \sum_{I' \in Succ(I)} R_{full}^{T,+}(I) \right]$$

$$= \sum_{I \in s} R^T(I) + \sum_{s' \in Succ(s)} \sum_{I' \in s'} R_{full}^{T,+}(I')$$

For each $s' \in Succ(s)$, $D(s') \subset D(s)$ and $s \notin D(s')$, so $|D(s')| < |D(s)|$ and we can apply the inductive hypothesis to show

$$\sum_{I \in s} R_{full}^{T,+}(I) \leq \sum_{I \in s} R^T(I) + \sum_{s' \in Succ(s)} \left[\sum_{s'' \in D(s')} \sum_{I \in s''} R^{T,+}(I) + TN_{s'}\epsilon_E \right]$$

$$\leq \sum_{s' \in D(s)} \sum_{I \in s'} R^{T,+}(I) + T\epsilon_E \sum_{s' \in Succ(s)} N_{s'}$$

$$= \sum_{s' \in D(s)} \sum_{I \in s'} R^{T,+}(I) + T\epsilon_E N_s$$

This completes the inductive argument. By using regret matching in the trunk, we ensure $R^T(I) \leq \Delta\sqrt{AT}$, proving the lemma for $k_G = \Delta|\mathcal{I}_{TR}|\sqrt{A}$ and $j_G = N_{root}$. ∎

Lemma 7 *Given our strategy σ, if the opponent is acting at the root of a public subtree S from a set of actions A, with opponent best-response values $BV_{I \cdot a}(\sigma)$ after each action $a \in A$, then replacing our subtree strategy with any strategy that satisfies the opponent constraints $w_I = \max_{a \in A} BV_{I \cdot a}(\sigma)$ does not increase our exploitability.*

Proof. If the opponent is playing a best response, every counterfactual value w_I before the action must either satisfy $w_I = BV_I(\sigma) = \max_{a \in A} BV_{I \cdot a}(\sigma)$, or not reach state s with private information I. If we replace our strategy in S with a strategy σ_S' such that $BV_{I \cdot a}(\sigma_S') \leq BV_I(\sigma)$ we preserve the property that $BV_I(\sigma') = BV_I(\sigma)$. ∎

Theorem 2 *Assume we have some initial opponent constraint values w from a solution generated using at least T iterations of CFR-D, we use at least T iterations of CFR-D to solve each re-solving game, and we use a subtree value estimator such that $\min_{\sigma_S^* \in NE_S} \sum_{I \in \mathcal{I}_2^S} |v^{\sigma_S^*}(I) - v_I| \leq$*

33

ϵ_E, *then after d re-solving steps the exploitability of the resulting strategy is no more than* $(d+1)k/\sqrt{T} + (2d+1)j\epsilon_E$ *for some constants k, j specific to both the game and how it is split into subgames.*

Proof. Continual re-solving begins by solving from the root of the entire game, which we label as subtree S_0. We use CFR-D with the value estimator in place of subgame solving in order to generate an initial strategy σ_0 for playing in S_0. By Lemma 6, the exploitability of σ_0 is no more than $k_0/\sqrt{T} + j_0\epsilon_E$.

For each step of continual re-solving $i = 1, ..., d$, we are re-solving some subtree S_i. From the previous step of re-solving, we have approximate opponent best-response counterfactual values $\widetilde{\mathrm{BV}}_I(\sigma_{i-1})$ for each $I \in \mathcal{I}_2^{S_{i-1}}$, which by the estimator bound satisfy $|\sum_{I \in \mathcal{I}_2^{S_{i-1}}} BV_I(\sigma_{i-1}) - \widetilde{\mathrm{BV}}_I(\sigma_{i-1})| \leq \epsilon_E$. Updating these values at each public state between S_{i-1} and S_i as described in the paper yields approximate values $\widetilde{\mathrm{BV}}_I(\sigma_{i-1})$ for each $I \in \mathcal{I}_2^{S_i}$ which by Lemma 7 can be used as constraints $w_{I,i}$ in re-solving. Lemma 5 with these constraints gives us the bound $\mathrm{EXP}_{w_i,\sigma_{i-1}}^{S_i} + \mathrm{U}_{w_i,\sigma_{i-1}}^{S_i} \leq \epsilon_E$. Thus by Lemma 4 and Lemma 6 we can say that the increase in exploitability from σ_{i-1} to σ_i is no more than $\epsilon_E + \epsilon_{S_i} \leq \epsilon_E + k_i/\sqrt{T} + j_i\epsilon_E \leq k_i/\sqrt{T} + 2j_i\epsilon_E$.

Let $k = \max_i k_i$ and $j = \max_i j_i$. Then after d re-solving steps, the exploitability is bounded by $(d+1)k/\sqrt{T} + (2d+1)j\epsilon_E$. ∎

Best-response Values Versus Self-play Values

DeepStack uses self-play values within the continual re-solving computation, rather than the best-response values described in Theorem 2. Preliminary tests using CFR-D to solve smaller games suggested that strategies generated using self-play values were generally less exploitable and had better one-on-one performance against test agents, compared to strategies generated using best-response values. Figure 6 shows an example of DeepStack's exploitability in a particular river subgame with different numbers of re-solving iterations. Despite lacking a theoretical justification for its soundness, using self-play values appears to converge to low exploitability strategies just as with using best-response values.

One possible explanation for why self-play values work well with continual re-solving is that at every re-solving step, we give away a little more value to our best-response opponent because we are not solving the subtrees exactly. If we use the self-play values for the opponent, the opponent's strategy is slightly worse than a best response, making the opponent values smaller and counteracting the inflationary effect of an inexact solution. While this optimism could hurt us by setting unachievable goals for the next re-solving step (an increased $\mathrm{U}_{w,\sigma}^S$ term), in poker-like games we find that the more positive expectation is generally correct (a decreased $\mathrm{EXP}_{w,\sigma}^S$ term.)
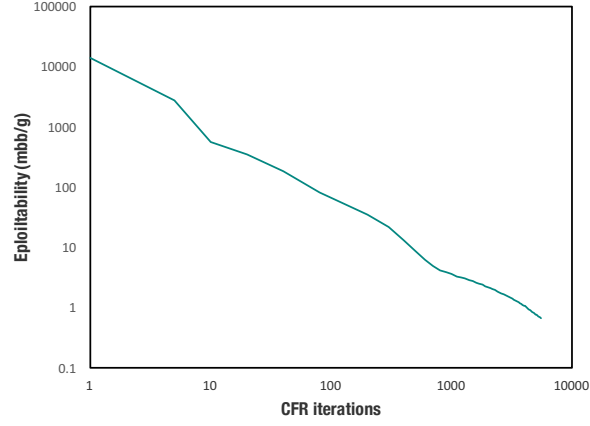
Figure 6: DeepStack's exploitability within a particular public state at the start of the river as a function of the number of re-solving iterations.

Pseudocode

Complete pseudocode for DeepStack's depth-limited continual re-resolving algorithm is in Algorithm 1. Conceptually, DeepStack can be decomposed into four functions: RE-SOLVE, VALUES, UPDATESUBTREESTRATEGIES, and RANGEGADGET. The main function is RE-SOLVE, which is called every time DeepStack needs to take an action. It iteratively calls each of the other functions to refine the lookahead tree solution. After T iterations, an action is sampled from the approximate equilibrium strategy at the root of the subtree to be played. According to this action, DeepStack's range, $\vec{r}_1$, and its opponent's counterfactual values, $\vec{v}_2$, are updated in preparation for its next decision point.

Algorithm 1 Depth-limited continual re-solving

INPUT: Public state S, player range $\mathbf{r}_1$ over our information sets in S, opponent counterfactual values $\mathbf{v}_2$ over their information sets in S, and player information set $I \in S$

OUTPUT: Chosen action a, and updated representation after the action $(S(a), \mathbf{r}_1(a), \mathbf{v}_2(a))$

1: **function** RE-SOLVE$(S, \mathbf{r}_1, \mathbf{v}_2, I)$
2:     $\sigma^0 \leftarrow$ arbitrary initial strategy profile
3:     $\mathbf{r}_2^0 \leftarrow$ arbitrary initial opponent range
4:     $R_G^0, R^0 \leftarrow \mathbf{0}$ ▷ Initial regrets for gadget game and subtree
5:     **for** $t = 1$ to T **do**
6:        $\mathbf{v}_1^t, \mathbf{v}_2^t \leftarrow$ VALUES$(S, \sigma^{t-1}, \mathbf{r}_1, \mathbf{r}_2^{t-1}, 0)$
7:        $\sigma^t, R^t \leftarrow$ UPDATESUBTREESTRATEGIES$(S, \mathbf{v}_1^t, \mathbf{v}_2^t, R^{t-1})$
8:        $\mathbf{r}_2^t, R_G^t \leftarrow$ RANGEGADGET$(\mathbf{v}_2, \mathbf{v}_2^t(S), R_G^{t-1})$
9:     **end for**
10:    $\overline{\sigma}^T \leftarrow \frac{1}{T}\sum_{t=1}^{T}\sigma^t$ ▷ Average the strategies
11:    $a \sim \overline{\sigma}^T(\cdot|I)$ ▷ Sample an action
12:    $\mathbf{r}_1(a) \leftarrow \langle \mathbf{r}_1, \sigma(a|\cdot)\rangle$ ▷ Update the range based on the chosen action
13:    $\mathbf{r}_1(a) \leftarrow \mathbf{r}_1(a)/\|\mathbf{r}_1(a)\|_1$ ▷ Normalize the range
14:    $\mathbf{v}_2(a) \leftarrow \frac{1}{T}\sum_{t=1}^{T}\mathbf{v}_2^t(a)$ ▷ Average of counterfactual values after action a
15:    **return** $a, S(a), \mathbf{r}_1(a), \mathbf{v}_2(a)$
16: **end function**

17: **function** VALUES$(S, \sigma, \mathbf{r}_1, \mathbf{r}_2, d)$ ▷ Gives the counterfactual values of the subtree S under σ,
                                                     computed with a depth-limited lookahead.
18:     **if** S is terminal **then**
19:        $\mathbf{v}_1(S) \leftarrow U_S \mathbf{r}_2$ ▷ Where U_S is the matrix of the bilinear utility function at S,
20:        $\mathbf{v}_2(S) \leftarrow \mathbf{r}_1^\mathsf{T} U_S$ $U(S) = \mathbf{r}_1^\mathsf{T} U_S \mathbf{r_2}$, thus giving vectors of counterfactual values
21:        **return** $\mathbf{v}_1(S), \mathbf{v}_2(S)$
22:     **else if** $d = $ MAX-DEPTH **then**
23:        **return** NEURALNETEVALUATE$(S, \mathbf{r}_1, \mathbf{r}_2)$
24:     **end if**
25:    $\mathbf{v}_1(S), \mathbf{v}_2(S) \leftarrow \mathbf{0}$
26:    **for** action $a \in S$ **do**
27:        $\mathbf{r}_{\text{Player}(S)}(a) \leftarrow \langle \mathbf{r}_{\text{Player}(S)}, \sigma(a|\cdot)\rangle$ ▷ Update range of acting player based on strategy
28:        $\mathbf{r}_{\text{Opponent}(S)}(a) \leftarrow \mathbf{r}_{\text{Opponent}(S)}$
29:        $\mathbf{v}_1(S(a)), \mathbf{v}_2(S(a)) \leftarrow$ VALUE$(S(a), \sigma, \mathbf{r}_1(a), \mathbf{r}_2(a), d+1)$
30:        $\mathbf{v}_{\text{Player}(S)}(S) \leftarrow \mathbf{v}_{\text{Player}(S)}(S) + \sigma(a|\cdot)\mathbf{v}_{\text{Player(S)}}(S(a))$ ▷ Weighted average
31:        $\mathbf{v}_{\text{Opponent}(S)}(S) \leftarrow \mathbf{v}_{\text{Player}(S)}(S) + \mathbf{v}_{\text{Opponent(S)}}(S(a))$
                                          ▷ Unweighted sum, as our strategy is already in-
                                          cluded in opponent counterfactual values
32:     **end for**
33:     **return** $\mathbf{v}_1, \mathbf{v}_2$
34: **end function**

36

35: **function** UPDATESUBTREESTRATEGIES($S, \mathbf{v}_1, \mathbf{v}_2, R^{t-1}$)
36:     **for** $S' \in \{S\} \cup$ SubtreeDescendants(S) with Depth(S') < MAX-DEPTH **do**
37:         **for** action $a \in S'$ **do**
38:             $R^t(a|\cdot) \leftarrow R^{t-1}(a|\cdot) + \mathbf{v}_{\text{Player}(S')}(S'(a)) - \mathbf{v}_{\text{Player}(S')}(S')$
                                                                        ▷ Update acting player's regrets
39:         **end for**
40:         **for** information set $I \in S'$ **do**
41:             $\sigma^t(\cdot|I) \leftarrow \frac{R^t(\cdot|I)^+}{\sum_a R^t(a|I)^+}$ ▷ Update strategy with regret matching
42:         **end for**
43:     **end for**
44:     **return** σ^t, R^t
45: **end function**

46: **function** RANGEGADGET($\mathbf{v}_2, \mathbf{v}_2^t, R_G^{t-1}$)   ▷ Let opponent choose to play in the subtree or receive the input value with each hand (see Burch et al. (*17*))

47:     $\sigma_G(\text{F}|\cdot) \leftarrow \frac{R_G^{t-1}(\text{F}|\cdot)^+}{R_G^{t-1}(\text{F}|\cdot)^+ + R_G^{t-1}(\text{T}|\cdot)^+}$ ▷ F is Follow action, T is Terminate
48:     $\mathbf{r}_2^t \leftarrow \sigma_G(\text{F}|\cdot)$
49:     $\mathbf{v}_G^t \leftarrow \sigma_G(\text{F}|\cdot)\mathbf{v}_2^{t-1} + (1 - \sigma_G(\text{F}|\cdot))\mathbf{v}_2$ ▷ Expected value of gadget strategy
50:     $R_G^t(\text{T}|\cdot) \leftarrow R_G^{t-1}(\text{T}|\cdot) + \mathbf{v}_2 - v_G^{t-1}$ ▷ Update regrets
51:     $R_G^t(\text{F}|\cdot) \leftarrow R_G^{t-1}(\text{F}|\cdot) + \mathbf{v}_2^t - v_G^t$
52:     **return** $\mathbf{r}_2^t, R_G^t$
53: **end function**