2025 IEEE International Conference on Intelligence and Security Informatics (ISI)
July 12-14, 2025. Hong Kong, China

The Cost of Uncertainty in Self-play Reinforcement Learning and Search

Marcus Oscarsson*, Joel Brynielsson*†, Mika Cohen*†, Farzad Kamrani†, Christoffer Limér†
*KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
†FOI Swedish Defence Research Agency, SE-164 90 Stockholm, Sweden
Email: moscars@kth.se, joel@kth.se, mikac@kth.se, farzad.kamrani@foi.se, christoffer.limer@foi.se

2025 IEEE International Conference on Intelligence and Security Informatics (ISI) | 979-8-3315-1276-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISI65680.2025.11201174

***Abstract*—The combination of reinforcement learning and look-ahead search introduced in AlphaGo, has revolutionized our understanding of tactics and strategy in classical strategy games such as Go and chess. Until recently, this pioneering approach has been limited to perfect information games, where players have full information about the current state of the game. This paper investigates the recent generalization of reinforcement learning with search to imperfect information games, such as poker, where parts of the game state, e.g., the opponent's hand, is hidden from the player. The paper explores how well this approach scales as the amount of hidden information increases. To this end, the current state of the art in reinforcement learning with search, the student of games general learning algorithm, is reproduced and evaluated across three variants of a custom poker game, each differing by the number of hidden cards dealt to players. It is found that games with less hidden information are learned more effectively, and that computational demands scale sublinearly with increasing hidden information.**

***Index Terms*—Reinforcement learning; tree search; imperfect information games; computer poker; counterfactual regret minimization; student of games algorithm.**

I. Introduction

Strategy games have a rich history of being used as benchmarks to measure progress in artificial intelligence (AI). Notable advances include Deep Blue reaching superhuman performance in chess in 1997 [1], and AlphaGo reaching superhuman performance in Go in 2016 [2]. A common characteristic of most strategy games mastered by AI has been that they are perfect information games, where all aspects of the game state are completely visible to both players.

Adversarial interactions in the real world often involve uncertainty about the adversary, making real-world decision-making more closely match imperfect information games, where players can have asymmetric information about the game state [3]. Indeed, in serious gaming, as practiced in, e.g., defense [4] and cybersecurity [5], [6], hidden information (the "fog of war") often plays an essential role.

In this more general class of games, poker has long served as the main benchmark for AI algorithms. Until recently, AI algorithms could only play small poker variants developed for research purposes. However, with recent advances in tree search algorithms, superhuman AIs in conventional poker variants have been developed [7], [8], [9], [10].

While these milestones in poker were based on look-ahead search alone, the strongest AIs for perfect information games use a combination of reinforcement learning and look-ahead search introduced in AlphaGo and later generalized in AlphaZero [11]. A first step towards generalizing this paradigm further to the broader class of imperfect information games was achieved in recursive belief-based learning (ReBeL) [12], a combination of reinforcement learning and look-ahead search. ReBeL reached superhuman performance in poker and poker-like games without relying on the game-specific abstraction techniques used in earlier, purely search-based AIs, while requiring substantially less computational effort. In November 2023, DeepMind published a refinement and extension of ReBeL that incorporated even more aspects of AlphaZero: the student of games (SoG) algorithm [13]. SoG achieves strong performance in large perfect and imperfect information games alike, being the first algorithm to do so and an important step toward truly general algorithms for arbitrary environments.

Poker games have often been seen as a good proxy for real-world decision-making [8], [9], as many real-world applications, e.g., negotiations, can be modeled as poker-like games. However, a key difference between poker games and more general decision-making is that the degree of hidden information in poker is solely determined by the private cards, whereas the degree of uncertainty can be larger and less tangible in other applications. If we want to apply general algorithms such as SoG to more complex scenarios, or apply similar algorithms as reasoning modules in larger systems, we may need to handle significantly more hidden information.

This paper investigates how reinforcement learning with look-ahead search scales with an increased degree of hidden information. We evaluate this in the realm of poker games, the standard benchmark in the field, and with the current state of the art, SoG. The research question is formulated as follows:

- How does the degree of imperfect information in poker games affect the performance of the student of games general learning algorithm?

To the best of our knowledge, the paper provides the first public implementation of the SoG algorithm and the first replication outside of DeepMind.

II. Background

This section introduces the relevant concepts and terminology from poker variants and reviews previous research

related to imperfect information games and general learning algorithms.

A. Poker

This paper focuses on two-player *heads-up* poker variants. A typical poker match consists of multiple games, and each game consists of multiple rounds. Each round ends with a betting phase where players wager chips; if a player is unwilling to match a wager, they *fold*, and the other player wins the *pot*. The most commonly played poker variants consist of four rounds. During the first round, the players are dealt a fixed number of private cards, called their *hand*. All subsequent rounds, called the *flop*, *turn*, and *river*, begin with cards being dealt face-up as public cards. If neither player folds by the end of the *river* betting phase, they combine their private and public cards to create the best five-card hand; the strongest hand wins the pot.

Different poker variants have different betting rules. *Limit* games restrict players to a single bet size per betting phase. *Pot-limit* games bound bets by the current pot size. *No-limit* games allow players to bet any amount up to all their chips, called going *all-in*. Two popular poker families are Texas hold'em and Omaha. The primary difference is that Omaha hands contain four private cards, while hold'em hands have two. The most popular poker variant is *no-limit hold'em*, followed by *pot-limit Omaha*.[1]

In most variants, players must wager a small amount of chips before the first round to stimulate play. The largest forced wager is the big blind (bb), typically about 1% of the players' starting stack. The common win-rate metric in poker AI literature is milli big blinds per game (mbb/g), which measures how many thousandths of a big blind a player wins on average per game [8], [9].

B. Related Work

Significant effort has been devoted to developing algorithms for poker games. The introduction of counterfactual regret minimization (CFR) in 2008 was a major breakthrough, allowing small poker games to be solved [14]. In 2014, an enhancement of CFR, called CFR$^+$, was developed, improving its convergence rate by up to an order of magnitude [15]. Using CFR$^+$, researchers tackled a large poker game played by humans, and in 2015, using 900 core-years of computing and 11 TB of storage, they solved heads-up limit hold'em, achieving near-zero exploitability [7].

The focus quickly shifted to the more popular no-limit hold'em. While limit hold'em could be solved with a modern computer cluster by traversing its game tree of 10^{14} information states, no-limit hold'em's game tree of 10^{164} states made this approach intractable. Researchers borrowed ideas from perfect information games, dividing the game tree into subgames and using neural networks to summarize game states. However, the imperfect information in poker significantly increased the complexity of these methods.

[1]https://upswingpoker.com/poker-rules/pot-limit-omaha-rules/.

In 2016, two research groups made significant progress, creating superhuman AI agents for no-limit hold'em. Moravčík et al. published DeepStack [8], while Brown et al. introduced Libratus [9]. Both teams developed new methods for dividing the game tree into subgames and refining strategies during play. In 2018, the team behind Libratus released Pluribus, extending their techniques from heads-up poker to six-player variants of the game [10].

Progress also advanced in developing general algorithms for imperfect information games. In 2020, Brown et al. developed ReBeL, achieving superhuman performance in poker and the game Scotland Yard [12]. ReBeL combined search, learning, and game-theoretic reasoning through self-play.

In November 2023, the team behind DeepStack released SoG [13]. This unified algorithm combines search, learning, and game-theoretic reasoning, integrating techniques from AlphaZero and DeepStack. SoG demonstrated strong performance in poker, Scotland Yard, chess, and Go, marking a significant step toward truly general game-playing algorithms.

III. Theory

This section introduces the formalisms used in the rest of the paper, and describes the prerequisite concepts and algorithms, which need to be understood to understand the SoG algorithm.

A. Game Theory

This section introduces game-theoretic concepts and notation using the framework of *extensive-form games*. A game between two players starts in a specific world state w^{init}. As players choose actions $a \in \mathcal{A}$, the game proceeds to successor world states $w \in \mathcal{W}$ until reaching a terminal state. A world state can be a decision node, a terminal node, or a chance node, representing a stochastic event with a fixed distribution, such as new public cards being revealed. Sequences of actions taken during the game are called histories and denoted $h \in \mathcal{H}$. At terminal histories, $z \subset \mathcal{H}$, each player i receives utility $u_i(z)$.

An *information state* is a representation of one player's information. Specifically, the histories $s_i \in \mathcal{S}_i$ for player i form a set of histories indistinguishable to player i due to missing information. For example, after players have been dealt their cards in poker, every situation where player i has been dealt a specific hand is indistinguishable to that player since they do not know the opponent's private cards; the histories in the information state differ only in the chance event determining the opponent's private cards. A player i plays a policy $\pi_i : \mathcal{S}_i \to \Delta(\mathcal{A})$, where $\Delta(\mathcal{A})$ denotes a probability distribution over actions $\mathcal{A}$. Given a policy profile for both players, $\pi = (\pi_i, \pi_{-i})$, where $-i$ is the opponent of player i, the utility of this profile to player i is denoted as $u_i(\pi_i, \pi_{-i})$.

Definition 1. A **best response** to an opponent policy π_{-i} is some policy π_i^{best} that maximizes the utility to player i against π_{-i}. Thus, $\pi_i^{best} = \text{argmax}_{\hat{\pi}_i}(u_i(\hat{\pi}_i, \pi_{-i}))$.

Definition 2. A policy profile $\pi = (\pi_a, \pi_b)$ is a **Nash equilibrium** if and only if π_a is a best response to π_b and π_b is a best response to π_a.

Definition 3. A policy $\pi_i : \mathcal{S}_i \to \Delta(\mathcal{A})$ is called **pure** if for every information state $\mathcal{S}_i$ there exists an action a such that the probability of $a \in \Delta(A)$ is 1; otherwise the policy is called **mixed**.

In perfect information games, the utility of an action depends only on the current world state $w \in \mathcal{W}$. For example, in chess, the value of an action in a specific board configuration is independent of the history of play (except for moves leading to a draw by repetition). In imperfect information games, the utility depends on additional factors like the opponent's beliefs about the player's private information and strategy, and vice versa. In poker, for instance, the value of bluffing depends on its frequency in the player's strategy; if the opponent believes the player never bluffs, bluffing is highly effective, and vice versa. If the opponent knows the player's private information, they can counter any strategy perfectly.

Thus, when considering actions from an information state S_i, the player must consider both their opponent's possible information states, as well as their own from the opponent's perspective, called their *range*, to conceal private information effectively. Formally, the range of player i is a distribution over information states $\Delta[S_i(S_{pub})]$ consistent with the public information S_{pub} for player i.

Game-theoretic reasoning in imperfect information games requires managing uncertainty about the current world state by considering the ranges of both the opponent and oneself. To handle this efficiently, we define a *public belief state* (PBS) $\beta = (S_{pub}, r)$, $r \in \Delta[S_i(S_{pub})] \times \Delta[S_{-i}(S_{pub})]$, where r contains the ranges of both players over the public state S_{pub}. This modeling of game states is called *belief representation*, while the extensive-form representation is termed *discrete representation* [12]. Both representations are used in SoG and are further discussed in Section III-B.

B. Re-solving

Re-solving is a method for finding an optimal policy within a limited part of a game, called a subgame; a subgame is a game rooted in the public state of a larger game. In poker, a subgame is uniquely defined by the betting history and the public cards at the root of the subgame.

CFR and CFR$^+$ can solve large games by traversing the entire game tree at each iteration using the *discrete representation*. In 2014, CFR$^+$ was used to solve limit hold'em, precomputing a complete strategy for its $\sim 10^{14}$ information states, requiring 11 TB of storage.[2] The policy for each state could then be retrieved during play from this lookup table. However, this approach is infeasible for larger games like no-limit hold'em, with 10^{164} information states [8], or pot-limit Omaha, which is even larger. Thus, a method similar to the search used in perfect information games is needed, where we search to a limited depth $d > 0$ from the current public state. If an oracle could provide the expected utilities for each state below d, a Nash equilibrium could be found without traversing the entire game tree.

For perfect information games, we can use a search algorithm like Monte Carlo tree search, starting from the current game state, and train a deep neural network to act as an oracle, taking a game state as input and outputting the value for both players. To achieve a similar decomposition in imperfect information games, we use the *belief representation*, which includes all relevant information about the history and the players' ranges. This allows constructing a subgame from the current public belief state, β.

Safe re-solving is a method for solving subgames [9]. Assuming a player uses a strategy covering all possible information states, called the *blueprint* strategy π^b, re-solving improves play in a subgame by searching for a better solution than the blueprint. One approach is to assume that both players follow the blueprint until the subgame, and then solve the subgame assuming that their ranges match what π^b implies. This method is called *unsafe subgame solving* because it assumes that the opponent plays according to the blueprint, leading to potentially exploitable strategies if the opponent deviates [9].

A stronger approach is *safe re-solving*, which creates a larger augmented subgame. For each opponent information state $S_{-i}(S_{pub})$, the opponent has the option to opt-out of the subgame and receive the payoff for playing optimally against our blueprint strategy in the subgame. This payoff is the counterfactual value for the opponent at the start of the subgame, $v_{-i}(\pi^b, h_{pub})$. Re-solving with these augmented actions regularizes the subgame strategy to be at least as strong as the blueprint strategy, ensuring that the subgame solution's exploitability is no higher than that of the blueprint strategy. This likely results in a less exploitable strategy due to the more detailed search conducted in the subgame.

Continual re-solving uses safe re-solving without needing a precomputed blueprint strategy. Instead, it employs solutions from subgames higher up in the game tree as the blueprint for new subgames. These blueprint solutions from previous searches can be passed down the search in a compressed form, requiring only two vectors: the acting player's range and the counterfactual values for the opponent. These counterfactual values provide an upper bound to the expected value for the opponent in all information states, given the history. The acting player's current range can be computed by applying Bayes' rule to the previous range, given the policy and action taken. The opponent's counterfactual values are produced when solving a subgame using CFR. This method is advantageous as it is not directly affected by the opponent's actions: the opponent's range is not part of the re-solving, and the upper bound on their counterfactual values holds as they act.

C. Student of Games

We have now covered the necessary theory to understand how the SoG algorithm works. SoG searches the game tree using continual re-solving. The subgames generated during re-solving are solved with a combination of CFR and CFR$^+$.

[2]Without compression, 262 TB would be needed using single-precision floating-point numbers.

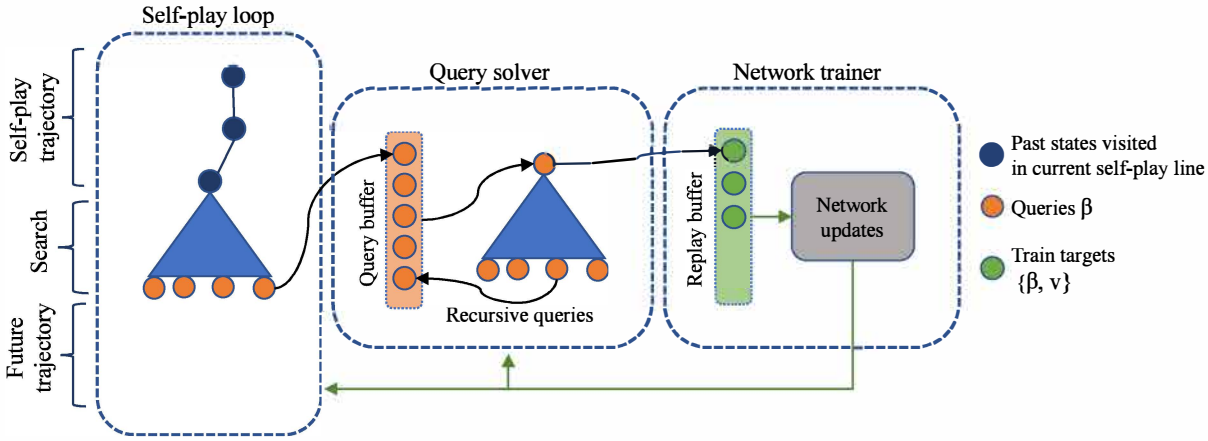

Fig. 1. Schematic of the self-play training process. Self-play is conducted with two identical SoG agents playing against each other. Neural net queries made during self-play are logged in a query buffer, and these queries are solved by solving a subgame rooted at these states. The feature-target pair found by the query solver is added to a replay buffer and is then used for network training. The figure is an adaptation of Fig. 8 by Schmid et al. [13].

Specifically, it employs linear averaging and regret matching$^+$ from CFR$^+$ [13], [15], while using simultaneous updates similar to standard CFR.

Continual re-solving and CFR are tailored to achieve strong performance in imperfect information games. To ensure that SoG also performs well in perfect information games, the authors combined continual re-solving with an expanding search tree. Instead of limiting a subgame to a fixed depth d, the subgame is dynamically grown. The growth rate of the subgame is controlled by two parameters: s and c. The parameter s denotes the total number of expansion simulations, and c is the number of expansions of the subgame tree per CFR iteration. The total number of CFR iterations in the algorithm is thus $\frac{s}{|c|}$.

In perfect information games, more resources are dedicated to expansions and fewer to CFR updates, as there is no uncertainty about players' ranges and no need to find a mixed strategy. Conversely, in imperfect information games, the addition of new public belief states is slower, and more computation is focused on refining and balancing the strategy.

D. Sound Self-play

Neural networks summarize public belief states at the leaf nodes of subgames. Schmid et al. developed sound self-play for training these neural networks in SoG through self-play [13].

Two instances of the SoG algorithm play against each other, evaluating each situation using continual re-solving with subgame decomposition, as described in Section III-B. The values of all non-terminal leaf nodes of the subgames are estimated by the neural network. The queries to the network are public belief states, β, that the network evaluates. A fixed proportion, defined by the hyperparameter q_{search}, of the queries produced during self-play are added to a query buffer and later revisited by a solver that studies the situation more closely by re-solving subgames rooted at these public belief states. New recursive queries may be created if the subgame rooted at β does not reach the end of the game in all paths. A fraction, defined by the hyperparameter $q_{recursive}$, of these recursive queries are added back to the query buffer. For each query studied by the solver, a feature-target pair, $\{\beta, v\}$, is created containing the PBS and the corresponding counterfactual values. These pairs are added to a replay buffer used to train the neural network. Fig. 1 provides a schematic overview of the modules used in self-play training.

This self-referential approach, where the network produces its own training targets, leads to a bottom-up learning process. Initially, the network learns to estimate the values of states near the end of the game, as those targets do not rely on the network. Gradually, the network improves at estimating values higher in the game tree, as the reliability of targets produced by the solver increases.

Algorithm 1 shows the primary self-play loop, using RESOLVE(L), which corresponds to re-solving a subgame through continual re-solving. To ensure generalization, the

Algorithm 1 Sound Self-play

procedure SELFPLAY
  Get world state corresponding to the start of the game $w \leftarrow w^{\text{INIT}}$
  while w is not terminal **do**
    if chance acts in w **then**
      $a \leftarrow$ uniform random action
    else
      ▷ Act for all non-chance players
      $\pi_{\text{controller}} \leftarrow$ SELFPLAYCONTROLLER(w)
      ▷ Mix policy with uniform prior to encourage exploration
      $\pi_w^{\text{selfplay}} \leftarrow (1-\epsilon) \cdot \pi_{\text{controller}} + \epsilon \cdot \pi_{\text{uniform}}$
      $a \leftarrow$ sample action from π_w^{selfplay}
    end if
    $w \leftarrow$ apply action a on state w
  end while
end procedure

procedure SELFPLAYCONTROLLER(w)
  $\beta \leftarrow$ public state at w
  $L \leftarrow$ tree for the subgame rooted at β
  $v, \pi, nn\_queries \leftarrow$ RESOLVE(L)
  $queries \leftarrow$ pick on average q_{search} neural net queries β from $nn\_queries$
  Append $queries$ to the query buffer
  return $\pi(\beta)$
end procedure

sound self-play encourages exploration off the self-play line in two ways. First, it mixes the strategy produced by SoG with a uniform strategy using a strength factor ϵ, called the uniform policy mix. Second, recursive queries generated during query solving help generalize to situations off the self-play line.

The training procedure has strong theoretical guarantees, converging to a Nash equilibrium strategy asymptotically as the number of CFR iterations for re-solving subgames approaches infinity, assuming a sufficiently expressive neural network architecture. For the proof, see Theorem 4 and the accompanying proof in the supplementary materials of Schmid et al. [13].

IV. Methodology

This section describes the methods used to evaluate the research question, detailing the evaluation method, the performed experiments, and the metrics used. It also provides the context necessary for replicating the results, by describing the training process and the neural networks used.

A. Turn Pot-limit Omaha

To analyze the scaling properties of SoG, we devised a custom poker game with desirable attributes for our evaluation. This game allows control of the degree of hidden information while minimizing emergent differences in the game rules that may be influenced by varying degrees of hidden information. We call the game turn pot-limit Omaha (TPLO). The first public card revealed is the turn. As a result, our game has three betting rounds instead of four. We limit the deck size to 12 cards; jack through ace in three suits. As alluded to by its name, the game is pot-limit.

We devised three variants of TPLO, named TPLO1, TPLO2, and TPLO3, where the number in each name indicates the number of private cards dealt to each player. We removed flushes, straights, and full houses from TPLO3, since these combinations are not possible in TPLO1 and TPLO2.

B. Student of Games Implementation

We implemented the SoG algorithm in C++ and the network architecture in Torch. Our implementation makes one modification to the original SoG algorithm: instead of dynamically growing the game tree for subgames, we use fixed-size trees. This restriction should not significantly affect the performance in poker, since the dynamically constructed game trees are shallow enough in poker to be prebuilt before solving (given the tree expansion rate $s = 10$ used by the SoG authors for no-limit hold'em).

C. Evaluation Framework

Our SoG implementation initially targeted the small academic poker variant pot-limit Leduc hold'em [16]. In this variant, players are dealt one private card from a six-card deck, and one public card is revealed. This experiment primarily validated the implementation, as the small game tree allows direct calculation of exploitability. We measured exploitability for varying CFR iterations to compare scaling performance with prior work. More iterations refine the strategy and approach equilibrium, reducing noise in training data as the query solver becomes more precise.

The primary set of experiments was designed to answer the research question. This involved testing our SoG implementation across the three variants of TPLO. Since the only variable factor for these three games is the level of hidden information, the performance impact of this aspect on SoG could be evaluated.

The algorithm was trained separately for each of the three TPLO games. During training, snapshots of partially trained agents competed against the fully trained model in their respective games. The relative strength of these snapshots was compared for each game to determine how much, and at what rate, the model improved during training. The fully trained agent also played against a uniform random opponent. Since the TPLO games have the same action space, the uniform random opponent was identical across games, which allows for a direct comparison between agents. All agents used an action abstraction allowing one pot-size bet.

Another metric used was the validation loss during training. Training was conducted in 30 cycles, each consisting of a sequence of self-play, query-solving, and network training. The validation loss presented is the validation loss on the most recent cycle processed up to that point. In other words, after n cycles, the validation loss measured is the validation loss on the data produced during the nth cycle. We also measure convergence by how much the strategy changes between successive cycles. Since the strategies in prior states affect the ranges in later states, we consider the strategy at the root of the game for a fair comparison, as the ranges at the root are always the same.

D. Variance Reduction

Poker games have a high degree of variance, and luck plays a large part in which player wins a game. Consequently, the evaluation of poker agents can require a large number of games to get statistically significant results. Such an evaluation can be computationally costly and motivates the use of variance reduction techniques. For our evaluation, we use imaginary observations [17] in all situations where the players reveal their private cards. This technique utilizes the fact that we know the strategies of both players, and therefore we know the ranges of both players in all states. So when the players need to reveal their private cards, we do not have to consider the specific hand they held in that game—we can instead utilize their range as a distribution over all possible hands, given how the game played out. So with this method, we get the expected value of $\mathcal{O}(|\text{possible private hands}|^2)$ different games, weighted by their probability of occurring, thereby significantly reducing the variance.

E. Counterfactual Value Network

The counterfactual value network (CVN) is used to summarize game states, which is used to limit the search depth and

act as an oracle for states on the frontier of subgames being re-solved.

The input to the CVN is a public belief state β, which includes a representation of the public state S_{pub} and beliefs r regarding the information states of both players. For poker games, the public information S_{pub} is summarized by three pieces of information: the public cards, the player next to act, and the total bets by both players. The public cards are represented as a one-hot encoding of all cards in the deck. The player next to act (including the chance player) is represented by one-hot encoding, and the sum of bets made by each player is normalized by their stack size. The beliefs about private information, $r \in \Delta[S_i(S_{pub})] \times \Delta[S_{-i}(S_{pub})]$, are also provided as input to the network, where for poker games this is a distribution over all possible private hands. The output of the CVN is the values to both players of β for each information state, and the architecture used is a multilayer perceptron.

The hyperparameters for the architecture were chosen based on prior work as well as what showed promising results during early testing. Moravčík et al. showed with DeepStack that they reached significant diminishing returns after three hidden layers for no-limit hold'em [8]. Schmid et al. used a hidden layer size of around 1.5 times the number of possible private hands [13]. Based on these works, we chose three hidden layers and a hidden layer size of 300 for the TPLO variants, where the number of private hands for TPLO3 is 220.

Early testing showed that adding normalization to the network, similar to the use of layer normalization [18] for turn endgame hold'em by Brown et al. [12], was beneficial. Our use of the GELU activation function [19] was also inspired by ReBeL [12]. Subsequent to the final linear layer, we force the output of the network to obey the zero-sum property of two-player zero-sum games [8]. The network was trained with the Adam optimizer [20] using Huber loss [21], as used widely in previous poker AI research [8], [12], [13].

F. Hyperparameters

Table I shows the hyperparameters used in the experiments. The hyperparameters for the architecture are described in Section IV-E, and the hyperparameters for self-play are inspired by those used by Schmid et al. [13] for no-limit hold'em with SoG, but are scaled for the smaller buffer sizes in our experiments. As a reference, previous top poker agents for large games typically use between 100 and 1000 CFR iterations [7], [8], [12]. Table II compares the games used for evaluation, including the sizes of the networks used.

V. Results

This section presents the exploitability of our SoG implementation in Leduc hold'em followed by the results of experiments on scaling with hidden information.

A. Exploitability in Pot-limit Leduc Hold'em

Exploitability was calculated for strategies produced by our SoG implementation and is presented in Fig. 2 as a function of the number of CFR iterations used when solving subgames.

TABLE I
HYPERPARAMETERS USED FOR TRAINING THE CVN WITH SELF-PLAY.

Hyperparameter	Symbol	Part of	Leduc	TPLO
CFR iterations	N	Re-solving	varies	200
CFR warm starting iterations	d	Re-solving	varies	100
Batch size		CVN	100	100
Number of hidden layers		CVN	4	3
Size of hidden layers		CVN	128	300
Initial learning rate (LR)	α_{init}	CVN	10^{-3}	10^{-4}
LR decay rate	T_{decay}	CVN	0.5	0.5
LR decay steps		CVN	3k	5k
Replay buffer size		Self-play	15k	30k
Queries per search	q_{search}	Self-play	30	4.5
Recursive queries per search	$q_{recursive}$	Self-play	3	0.5
Uniform policy mix	ϵ	Self-play	0.1	0.1

TABLE II
COMPARISON OF GAMES USED TO EVALUATE THE ALGORITHM. NETWORK SIZE REFERS TO THE TOTAL NUMBER OF LEARNABLE PARAMETERS. THE HIDDEN LAYERS ARE FIXED FOR THE THREE TPLO GAMES, WHILE THE INPUT AND OUTPUT LAYERS ARE ADJUSTED FOR THE NUMBER OF HANDS.

Game	Possible hands	Network size	Blinds	Stacks
Pot-limit Leduc	6	50K	300	1200
TPLO1	12	200K	100	5000
TPLO2	66	270K	100	5000
TPLO3	220	450K	100	5000

The same number of CFR iterations was also used when re-solving queries to obtain targets for network training.

We compared the self-play value net with unlimited search depth. The self-play value net uses our SoG implementation trained with sound self-play, while unlimited search depth employs continual re-solving, but always searches to the end of the game, bypassing the need for network estimation of public belief states. For reference, two additional strategies are included in the comparison: a uniform random strategy and a random value net. The latter uses continual re-solving with a randomly initialized value network, producing random strategies at the beginning of the game until the terminal states become reachable within subgames.

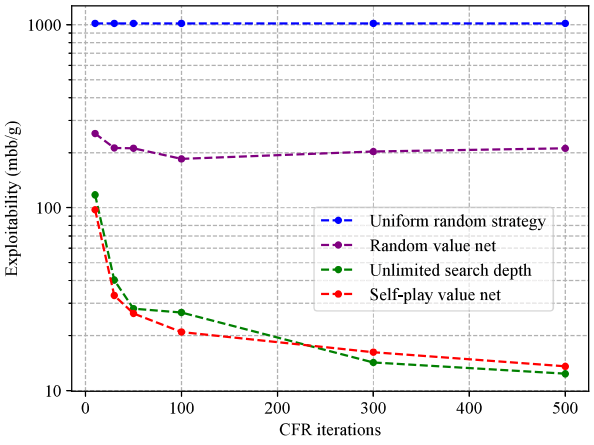

Fig. 2. Convergence of different techniques in pot-limit Leduc hold'em with respect to the number of CFR iterations used.

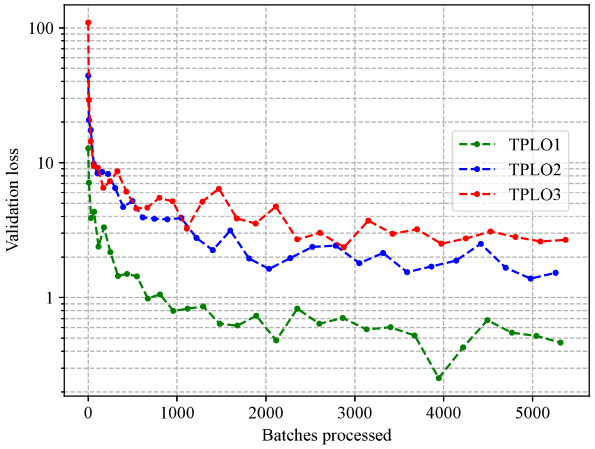

Fig. 3. Validation loss when training SoG for the three TPLO games.

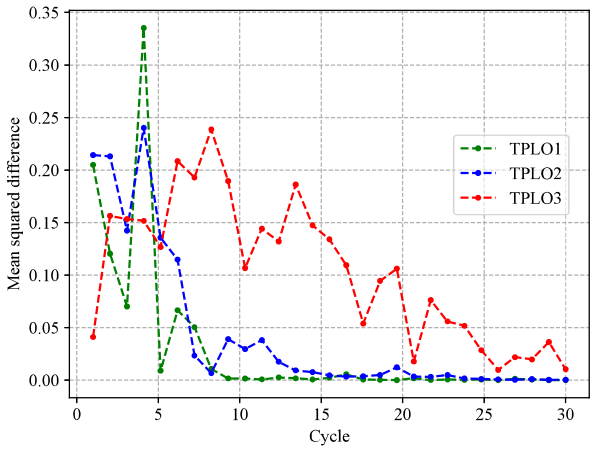

Fig. 4. The mean squared difference when comparing the policy produced at the root node between successive cycles.

B. Scaling with Hidden Information

The model was trained for the same number of cycles for each of the three games, and compared to the time required for training the algorithm for TPLO1; the training duration for TPLO2 was 1.23 times longer, and the training duration for TPLO3 was 1.77 times longer. Fig. 3 presents a comparison of the validation loss observed during training. Additionally, Fig. 4 shows the evolution of the policy at the root node of the game as a comparison over successive training cycles. The policy is represented as a matrix, where each row corresponds to a probability distribution over possible actions for a possible private hand.

Table III shows the results of the evaluation when comparing snapshots of models for the TPLO games. The evaluation uses the variance reduction techniques described in Section IV-D.

VI. Discussion

This section provides an analysis of the results presented in the previous section.

TABLE III
LUCK-ADJUSTED WIN RATES (MBB/G) FOR THE THREE TPLO GAMES. EACH VALUE SHOWS THE WIN RATE OF THE FULLY TRAINED MODEL AGAINST A SPECIFIC OPPONENT. OPPONENTS INCLUDE A UNIFORM RANDOM STRATEGY AND SNAPSHOTS OF THE MODEL AFTER 10% AND 30% OF TRAINING. WIN RATES ARE REPORTED WITH A 95% CONFIDENCE INTERVAL.

Opponent	TPLO1 (mbb/g)	TPLO2 (mbb/g)	TPLO3 (mbb/g)
Uniform random	2464 ± 149	1448 ± 137	469 ± 94
10%	104 ± 55	147 ± 54	237 ± 41
30%	38 ± 56	67 ± 55	63 ± 44

A. Exploitability in Pot-limit Leduc Hold'em

The exploitability results from our evaluation of pot-limit Leduc hold'em were consistent with those in the original SoG paper [13], and with similar algorithms like DeepStack [8] in no-limit Leduc hold'em.[3] Although the SoG paper lacks detailed training setup information for Leduc, our findings suggest that its exploitability results are replicable.

Interestingly, the self-play value net outperformed unlimited search depth with fewer CFR iterations. This result might seem counterintuitive, as unlimited search depth always searches to the end of the game, not needing the network to estimate state values. A possible explanation is that, despite searching to the end, unlimited search depth produces noisier early iteration value estimates. Consider the case where we use 20 CFR iterations—the value estimates produced by the network will always be based on what 20 CFR iterations would yield. Therefore, even after a small number of iterations, the self-play value net can produce decent value estimates for its leaf nodes (corresponding to 20 CFR iterations), while unlimited search depth only achieves value estimates as accurate as the current iteration. For this reason, it is reasonable that using a network trained with self-play initially converges faster than unlimited search.

During initial testing, we used random situation generation instead of self-play, similar to DeepStack's training [8]. This resulted in significantly worse exploitability, likely due to the poker-specific abstractions used in DeepStack. Neither SoG nor ReBeL employs such abstractions, and Brown et al. showed that training with randomly generated situations yields poor results for ReBeL [12]. Our experiments confirm this result for SoG as well.

B. Scaling with Hidden Information

In Table III, a higher degree of hidden information correlates with a significant reduction in the win rate against a uniform random strategy. This suggests that in a game with more hidden information, the SoG algorithm fails to learn the intricacies of the game to the same degree, given equal training time. The table also suggests that the algorithm converges more slowly with more hidden information. This finding is strengthened by Fig. 4, which shows that the time it takes for the strategy in the root node to stabilize increases with a higher degree of hidden information.

[3]https://github.com/lifrordi/DeepStack-Leduc.

When comparing the fully trained agent versus the agent that has completed 10% of the training cycles, the win rate is higher for games with more hidden information, with a similar although less prominent trend for the 30% partially trained snapshots. These results suggest that more imperfect information makes the training process less efficient and that the algorithm requires more training data to generalize and understand the game well.

A clear pattern also emerges in the training loss: a greater degree of hidden information corresponds to higher validation loss. An important consideration is that the network must handle a more complex problem for the larger games with respect to the size of the network output, as it needs to produce output values for all possible private hands for both players.

An interesting aspect of the results is the relative training time. TPLO3 has 18 times as many possible private hands compared with TPLO1, which means that the CVN needs about 18 times more input and output, and the CFR calculations made during re-solving have to handle 18 times as many states. Given this context it is a surprising result that the training time was only 1.77 times longer for TPLO3 relative to TPLO1. The slight difference can be explained by that our SoG implementation uses tensor objects from the Torch library for most operations on player ranges, and the Torch library is well optimized for tensor operations. Nonetheless, given that all parts of our implementation were written in C++, this low relative slowdown with respect to the number of possible private hands is a surprising result. The results suggest that SoG scales well with the degree of hidden information in terms of computational demands. It should, however, be noted that the specifics of the implementation can significantly affect the computational demands, and in this project the primary focus has been on clarity rather than on computational performance.

VII. Conclusions

In this paper, we presented the first public reproduction of the current state of the art in reinforcement learning with search: the student of games (SoG) algorithm. We evaluated how well the algorithm scales in a custom pot-limit Omaha variant with varying numbers of private cards. Our results show that the degree of hidden information significantly impacts the performance: the algorithm converges faster and learns the game to a greater extent with less hidden information. Somewhat surprisingly, the computational demands per training cycle appear to increase sublinearly with the degree of hidden information.

Looking ahead, future work could scale our experiments to larger poker games to assess whether the observed scaling properties hold in larger games and with more computational resources. It would also be valuable to test whether these scaling properties replicate in other imperfect information games with different forms of hidden information dynamics.

Code Availability

The full implementation of our SoG reproduction is available at https://github.com/moscars/student-of-games.

References

[1] M. Campbell, A. J. Hoane Jr., and F.-h. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, 2002, doi: 10.1016/S0004-3702(01)00129-1.

[2] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016, doi: 10.1038/nature16961.

[3] T. Sandholm, "The state of solving large incomplete-information games, and application to poker," *AI Magazine*, vol. 31, no. 4, pp. 13–32, 2010, doi: 10.1609/aimag.v31i4.2311.

[4] UK MoD, "Wargaming handbook," Development, Concepts and Doctrine Centre, Ministry of Defence Shrivenham, United Kingdom, 2017. [Online]. Available: https://www.gov.uk/government/publications/defence-wargaming-handbook

[5] J. Brynielsson, U. Franke, and S. Varga, "Cyber situational awareness testing," in *Combatting Cybercrime and Cyberterrorism: Challenges, Trends and Priorities*, B. Akhgar and B. Brewster, Eds. Cham, Switzerland: Springer, 2016, ch. 12, pp. 209–233, doi: 10.1007/978-3-319-38930-1_12.

[6] J. Brynielsson, M. Cohen, P. Hansen, S. Lavebrink, M. Lindström, and E. Tjörnhammar, "Comparison of strategies for honeypot deployment," in *Proceedings of the 2023 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2023)*. New York, NY: ACM, 2023, pp. 611–618, doi: 10.1145/3625007.3631602.

[7] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, "Heads-up limit hold'em poker is solved," *Science*, vol. 347, no. 6218, pp. 145–149, 2015, doi: 10.1126/science.1259433.

[8] M. Moravčík et al., "DeepStack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017, doi: 10.1126/science.aam6960.

[9] N. Brown and T. Sandholm, "Safe and nested subgame solving for imperfect-information games," in *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*. San Diego, CA: NeurIPS, 2017, pp. 689–699.

[10] N. Brown and T. Sandholm, "Superhuman AI for multiplayer poker," *Science*, vol. 365, no. 6456, pp. 885–890, 2019, doi: 10.1126/science.aay2400.

[11] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018, doi: 10.1126/science.aar6404.

[12] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, "Combining deep reinforcement learning and search for imperfect-information games," in *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS 2020)*. San Diego, CA: NeurIPS, 2020, pp. 17 057–17 069.

[13] M. Schmid et al., "Student of Games: A unified learning algorithm for both perfect and imperfect information games," *Science Advances*, vol. 9, no. 46, pp. 1–14, 2023, Art. no. eadg3256, doi: 10.1126/sciadv.adg3256.

[14] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, "Regret minimization in games with incomplete information," in *Proceedings of the 20th Conference on Neural Information Processing Systems (NIPS 2007)*. San Diego, CA: NeurIPS, 2007, pp. 1729–1736.

[15] O. Tammelin, "Solving large imperfect information games using CFR+," 2014, arXiv: 1407.5042.

[16] F. Southey et al., "Bayes' bluff: Opponent modelling in poker," in *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI 2005)*. Arlington, VA: AUAI Press, 2005, pp. 550–558, arXiv: 1207.1411.

[17] M. Bowling, M. Johanson, N. Burch, and D. Szafron, "Strategy evaluation in extensive games with importance sampling," in *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*. New York, NY: ACM, 2008, pp. 72–79, doi: 10.1145/1390156.1390166.

[18] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016, arXiv: 1607.06450.

[19] D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," 2016, arXiv: 1606.08415.

[20] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 2015 Third International Conference on Learning Representations (ICLR 2015)*, 2015, arXiv: 1412.6980.

[21] P. J. Huber, "Robust estimation of a location parameter," *Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964, doi: 10.1214/aoms/1177703732.