Self-Play Multi-Valued States for Imperfect Information Games

Kevin A. Wang
Meta AI (FAIR)
kevinwang@kevinwang.us

Anton Bakhtin
Work done while at Meta AI (FAIR)

David Wu
Work done while at Meta AI (FAIR)

Noam Brown
Work done while at Meta AI (FAIR)

ABSTRACT

Depth-limited search has been a vital ingredient in superhuman agents for games such as chess, go, and poker. However, while evaluating states at the depth-limit is simple in perfect-information games like chess and go, it's complex in imperfect-information games like poker. A solution called "multi-valued states" was used in the first ever superhuman multiplayer no-limit hold'em agent, Pluribus. This paper presents a self-play method for multi-valued states, and we show that it outperforms the handcrafted method used in Pluribus in a medium-sized poker game. We also conduct experiments in small games, showing that our method empirically decreases in exploitability as training progresses. We also show that our method can be seen as an imperfect-information generalization of AlphaZero, and that in a perfect-information setting, our method converges to an approximate Nash equilibrium with just one continuation policy.

KEYWORDS

Machine Learning, game theory, multi-valued states

1 INTRODUCTION

Depth-limited search has been a vital method for superhuman agents in perfect-information games like chess [11] and Go [27]. However, naively using perfect-information depth-limited search algorithms in imperfect-information games is unsound. This is because in imperfect-information games, histories do not have well-defined values like they do in perfect-information games. A previous method, multi-valued states [9], solves this problem by letting each player choose a continuation policy at each subgame leaf. The depth-limited subgames are modified so that upon reaching a leaf, the game does not instantly terminate. Instead, each player makes one final action: they simultaneously choose, from a set of policies, a policy that they would like to play for the rest of the game. Then, the terminal value is the expected value of each player playing their chosen policy for the rest of the game, starting at that leaf. We call these policies "continuation policies", and we call the sets of continuation policies "populations".

With a sufficient set of continuation policies for each player, the Nash equilibria of the depth-limited subgame are exactly the Nash equilibria of the original game. Even with a small number of continuation policies per player, the approach works well in practice: multi-valued states were successfully used as a component of the first superhuman AI for multiplayer poker, Pluribus [8].

Previously, two methods were described for producing the set of continuation policies. The first is called the bias method, in which the population consists of multiple copies of a blueprint, each modified to increase the probability of a different action at every infostate. The bias method was used in Pluribus. The second is the best-response method, which involves a process alternating between adding a best-response to current depth-limited subgame solution, and calculating a new depth-limited subgame solution.

This paper describes a new procedure for generating continuation policies from scratch, by training policy neural networks via self play. We call this self-play method. At a high level, the algorithm involves repeating the following procedure: adding a new, arbitrarily initialized continuation policy to the population, and then, over a number of self-play games, training the continuation policy to imitate the solved depth-limited subgame search policies, before finally freezing that continuation policy and repeating the procedure by adding a new one. This method reduces to an AlphaZero-like algorithm in the case of perfect information games.

We conduct experiments in a medium-sized poker game, where we show that our method produces agents which outperform agents that use the bias method for making continuation policies. We evaluate these agents on their head-to-head performance against an NFSP policy.

We also conduct experiments in games small enough that it is practical to compute a policy's exact exploitability. These experiments show that over the course of self-play, the search agent's exploitability tends to decrease. In the case of Leduc poker, our algorithm reaches a lower exploitability than a baseline algorithm, NFSP, with less wall-clock time for training.

Finally, we conduct experiments in a toy perfect-information game, to demonstrate that the self-play algorithm converges to a Nash equilibrium by only training a single continuation policy per player.

2 BACKGROUND

In this paper, we will use the factored observation game model from Kovařík et al. [16].

In this paper, we only consider finite, 2-player, zero-sum games.

A **world state** $w \in \mathcal{W}$ is a state in the game.

In an N-player game, the joint actions for the players are $\mathcal{A} = \mathcal{A}_1 x ... x \mathcal{A}_N$. The set of legal actions for player i at w is denoted by $\mathcal{A}_i(w) \subseteq \mathcal{A}_i$.

When transitioning from world state w to w' via joint action a, player i receives a **private observation** from a function $O_{priv(i)}(w, a, w')$, and all players receive the same **public observation** from the function $O_{pub}(w, a, w')$. After a joint action a is taken, each player i receives a reward $\mathcal{R}_i(w, a)$.

A **history** $h = (w^0, a^0, w^1, a^1, ..., w^t)$ is a finite sequence of actions and world states. An **information state** of a player (**infostate** for short, also called an **action-observation history (AOH))** is

a sequence of the player's observations and actions, denoted by $I = (O_i^0, O_i^1, O_i^2, ..., O_i^t)$.

A history with no legal actions is a **terminal** history.

A **public state** is a sequence of public observations, denoted by $s_{pub} = (O_{pub}^0, O_{pub}^1, ..., O_{pub}^t)$.

A **public belief state (PBS)** [4] β is a public state and a joint probability distribution over the players' possible infostates in the public state. A public belief state is also known as a *joint range* of a public state [17].

A **policy** (also known as a **strategy**) for a player i, denoted σ_i, is a function that maps each infostate to a probability distribution over legal actions at that infostate. A **policy profile** is a joint policy, one for each player: (σ_1, σ_2). The policy of the player other than i is denoted as σ_{-i}.

The **expected value (EV)** of a policy profile σ for a player i at a history h is the expected sum of future rewards for i given that all players play σ. It is denoted as $v_i^\sigma(h)$. The EV of the root history is v_i^σ.

A **best response** to σ_{-i} is a policy $BR(\sigma_{-i})$ such that $v_i^{(BR(\sigma_{-i}),\sigma_i)}$ is maximized. A policy profile is a **Nash equilibrium** if neither player can achieve a higher EV by switching to a different policy (given that the other players do not change their policies). In a two-player zero-sum game, the **exploitability** of a policy profile measures the amount that a policy profile can be exploited by best-responses. Formally, the exploitability of σ is $\sum_{p\in\{1,2\}} max_{\sigma_p} v_p(\sigma'_p, \sigma_{-p})/2$

In the context of perfect-information games, a **subgame** of a game *rooted* at a history h is identical to the original game, but starts at h. It is the set of states that can be reached by starting at h and applying any number of actions. A **depth-limited subgame** is a subgame that only extends for a limited number of actions past the root. A **leaf node**, or **leaf**, is a history at the bottom of a depth-limited subgame. If a history h is a terminal history in the depth-limited subgame, but not in the original game, it is a leaf node.

In this paper, we define the **imperfect-information subgame** of an imperfect-information game *rooted* at a public belief state β as identical to the original game, but at the start of the subgame, a history is sampled from the joint probability distribution of β. Play then continues from that history as usual.

3 RELATED WORK

3.1 Multi-Valued States

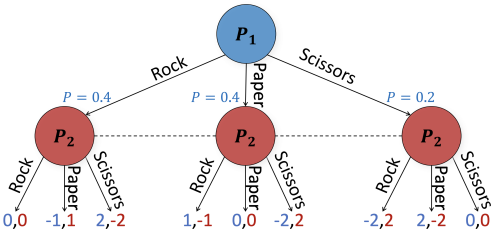
This section describes **multi-valued states** [9], a method for depth-limited solving in imperfect information games.

In perfect information games such as chess or Go, a subgame may be turned into a depth-limited subgame by turning states at some depth-limit into leaf nodes. That is, each state which we would like to turn into a leaf is replaced with a terminal history. The value of each terminal history is assigned to be the equilibrium value (or, realistically, an approximation thereof) of that state. If the terminal values are exactly the equilibrium values, then the Nash equilibria of the depth-limited subgame correspond to the Nash equilibria of the original game, and an agent can play the game perfectly by solving such depth-limited subgames for each move [26].

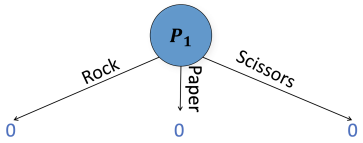
As explained in Brown et al. [9] and Kovařík et al. [17], this method does not work in imperfect information games. In these settings, turning histories into leaf nodes by assigning them an equilibrium value may result in depth-limited subgames whose equilibria do not correspond to equilibria of the full game. Thus, if one plays the full game by solving such subgames and playing according to them, the resulting full-game strategy could be exploitable. See Figure 1 for an example of such a scenario.

Multi-valued states fixes this problem by modifying the depth-limited subgame, allowing each player to privately choose from a set of policies (called **continuation policies**) after a leaf has been reached. In the modified subgame, players make this choice simultaneously, and they only know their information state for that leaf, not necessarily the precise history. The terminal value after a leaf is reached and both players choose is the expected value of each player playing their continuation policy from that history onwards. As with any game, players can play mixed strategies in the modified subgame, and can thus mix over their continuation policies at leaf information states. In practice, the expected value of a leaf node can be calculated exactly in small games, or approximated by performing a number of rollouts and averaging the rewards.

In this paper, we refer to the set of continuation policies as a **population**. (It has also been referred to as a "portfolio" [17].)



(a) Variant of Rock-Paper-Scissors in which the only optimal player 1 policy is (R=0.4, P=0.4, S=0.2)
Terminal values are color-coded.
Dashed lines indicate that Player 2 doesn't know which state they are in.



(b) The depth-limited subgame rooted at player 1's history with a depth-limit of 1, where leaves have been replaced by their expected value, assuming both players play optimally. Player 1 cannot find the optimal policy from solving this subgame.

Figure 1: Example showing the difficulty of evaluating states in imperfect-information games*

*Diagrams from Brown et al. [4] (used with permission)

When the population consists of a sufficient number of different continuation policies, the Nash equilibria of the modified subgame match the Nash equilibria of the original game. As intuition, imagine that each player's population consists of all their possible pure policies. Then, the resulting modified subgame is equivalent to the same subgame with no depth limit.

This method works well empirically. With only k=4 continuation policies per player, this method was successfully used as part of Pluribus, the first superhuman multiplayer no-limit hold'em AI [8].

In Brown et al. [9], two methods are given for generating the population of continuation policies. Both make use of a *blueprint policy* – a policy that is an approximate equilibrium of the full game. In large poker games such as no-limit Texas hold'em, this is usually computed by solving a smaller, abstracted version of the full game.

In the first method, the **bias** method, the population for each player is composed of the blueprint and several copies of the blueprint which are manually biased towards some actions. For example, in poker, one may create a biased blueprint by copying the blueprint but multiplying the probability of folding by 10 at all states (and then renormalizing the probabilities). One can do that same for checking/calling and for betting/raising, resulting in 4 versions of the blueprint. This is the method used in Pluribus.

The second method is the **best-response** method. (This method was previously named the **self-generative** method, but we will call it the best-response method in this paper to disambiguate it from our self-play method.) The population begins with just one strategy per player, the blueprint. One can define a policy for a given player which uses depth-limited search at the root with just that one "choice" of continuation policy, and plays according to the blueprint after the depth-limit. An opponent best-response is trained to exploit such a policy. That best-response is added to the opponent's population. We can again define a policy for the given player where they use depth-limited search at the root (the opponent now has two continuation policies to choose from), and play the blueprint beyond the depth-limit. Again, an opponent best-response is trained to exploit that policy. This process is repeated for each player for as many continuation policies as one wishes to generate. This process is similar to double-oracle methods in multi-agent reinforcement learning [2, 20, 22, 24].

3.2 Other Related Work

Our work builds off of the extensive literature of deep learning and search in games, which started in perfect information games [3, 27, 28, 33].

As we describe in Section 5, our method generates and maintains populations of policies. In this way, it is related to previous work on population-based multi-agent reinforcement learning, such as double oracle methods and PSRO [20, 24, 35]. Specifically, our method learns populations of policies, one at a time, through self-play, similar to Self-Play PSRO [21].

Our work uses neural nets and depth-limited subgame search in imperfect-information games. The ReBeL algorithm [4] also contains those elements, but while ReBeL learns a PBS value network, our method uses multi-valued states and learns policy networks. Like ReBeL, the algorithm in this work reduces to an AlphaZero-like algorithm in perfect information games. This work can be thought of as a parallel to ReBeL: one using self-play to learn PBS value functions, and one using self-play to learn multi-valued states.

Other methods using deep learning for approximating Nash equilibria policies in two-player zero-sum games are based on function-approximation versions of tabular algorithms like counterfactual regret minimization [32, 37], and do not use search at test-time [5, 13, 23, 31]. Likewise, algorithms closely related to standard reinforcement learning algorithms, like MMD [29] and NeuRD [15], use deep learning but do not use search at test-time.

4 NESTED DEPTH-LIMITED SUBGAME SOLVING

As in previous works on depth-limited solving [9], the method described in this paper uses nested solving. When using techniques to make subgame solving "safe" [4, 6, 10], nested solving can produce policies that have theoretical guarantees. For example, the resulting policy can be proven to be no less exploitable than a blueprint policy used to make it.

However, in the experiments in this paper, we do nested solving with "unsafe" subgame solving. Although this lacks any guarantees on exploitability, it is simpler than safe solving techniques. Empirically, it performs well (surprisingly, often better than safe subgame solving techniques [6, 8]). It was used solely [8] or in conjunction with safe subgame solving [9] in previous multi-valued states experiments.

In nested unsafe subgame solving [12], an agent initially roots a depth-limited subgame at the root PBS of the game. The agent solves the subgame and plays according to it, until it reaches a leaf of the subgame. The subgame can be solved with any equilibrium-finding technique, such as CFR (counterfactual regret minimization) [37]. Then, it computes a new PBS by using Bayes' rule, assuming both players play according to the solution of the subgame. It then roots a new subgame at that PBS, and continues play. Each time it reaches another leaf, it repeats this process, until it reaches a terminal history.

5 SELF-PLAY MULTI-VALUED STATES

In this section we describe a new method for generating population strategies, which we call **self-play multi-value states**. Like in the best-response method, we start with one continuation policy per player. However, unlike in the best-response method, we train these continuation policies through self-play. This is done as follows:

Given a set of continuation policies $\{\sigma_1, \sigma_2, ...\sigma_k\}$ for each player, (initially containing only one continuation policy per player), we have both players iteratively perform nested depth-limited subgame solving with multi-valued states using those policies, and both players play according to the depth-limited equilibrium solutions until reaching the the true end of the game. We then train σ_k for each player to imitate the solutions to all subgames solved during this self-play, and repeat across many games, improving the policy σ_k. We say that one iteration of self-play and training is an **inner-loop** iteration.

Once σ_k for each player has reasonably converged, we freeze it and never train it again, and we add a new arbitrarily initialized σ_{k+1} for each player to the set of continuation policies. We then repeat the entire procedure again for $k + 1$, and so on. We call one

iteration of adding, training, and freezing a new continuation policy per player an **outer-loop** iteration.

On the kth outer-loop iteration, there are always $k-1$ frozen continuation policies per player and 1 actively training continuation policy per player. The sizes of the gadget subgames solved grow by a factor of $O(k^2)$ as k grows, so k ought to stay reasonably small to keep subgame solving tractable. In our experiments, k does not exceed 10.

Algorithm 1 contains pseudocode for the algorithm.

Algorithm 1 Self-Play Multi-Valued States

```
function SELF-PLAY MULTI-VALUED STATES
    ℙ1 ← ∅        ▷ Player 1's population of continuation policies
    ℙ2 ← ∅        ▷ Player 2's population of continuation policies
    for k ← 1...K iterations do
        for player p ∈ {1, 2} do
            σ_p^k ← arbitrarily initialized policy
            ℙ_p ← ℙ_p ∪ σ_p^k
        end for
        for N iterations do
            training_data ← SELF-PLAY(ℙ1, ℙ1)
            for player p ∈ {1, 2} do
                Train σ_p^k to imitate training_data
            end for
        end for
    end for
end function

function SELF-PLAY(ℙ1, ℙ2)
    // this implementation uses unsafe nested solving
    training_data ← ∅
    h ← new initial game state
    PBS_h ← {h : 100%}
    while h is not a terminal state do
        G′ ← modified depth-limited subgame rooted at PBS_h
        σ′* ← Nash equilibrium of G′        ▷ e.g. solve with CFR
        add σ′* to training_data
        h ← leaf sampled from G′ according to σ′*
        PBS_h ← PBS of h according to σ′*
    end while
    return training_data
end function
```

This method has a number of benefits over the previous methods for selecting a population of continuation policies.

First, the self-play method doesn't require a precomputed blueprint policy. Blueprints are commonly used in algorithms for poker, since they can be created through abstractions [8]. However, in other games, it may not be clear how to generate a blueprint. Regardless, generating a blueprint adds an additional step to the algorithm, which the self-play method avoids.

Intuitively, the self-play method will organically generate useful continuation policies each iteration, as opposed to the bias method, where crude biasing of actions is not guaranteed to produce useful policies. In addition, the bias method requires one to inspect and manually define the biased policies, which may be non-trivial in games with complex action spaces. The self-play method, on the other hand, runs automatically with no domain knowledge required.

In perfect-information games, the self-play method converges to an equilibrium in the very first outer-loop iteration. This is not a feature of the best-response method. Like in ReBeL [4], self-play multi-valued states simply reduces to an AlphaZero-like algorithm when used in perfect information games.

Compared to the best-response method, the self-play method is more tractable to train. In large games, the approximate best-response of the best-response method can be trained through a deep reinforcement learning algorithm. However, each episode of the game will only produce 1 datapoint for each state actually traversed during the course of the episode. In contrast, the self-play method uses as training data all infostates in each depth-limited subgame it solves, making it tractable to train in larger games.

6 EXPERIMENTS

We performed experiments using games in the OpenSpiel library [18]. All continuation policy neural nets were implemented using PyTorch [25].

6.1 Experiments in Medium-Sized Poker

We performed experiments using a medium-sized game: "Universal Poker" with mostly default parameters , but with a smaller deck size of 10 cards (5 ranks and 2 suits) and larger stack sizes per player. The game is similar to Leduc Poker [30], but instead of a maximum of two bets per round, there is a maximum of four bets total. Instead of a fixed bet size, there are two bet sizes: pot and all-in. The game tree has 74,261 histories.

Each player antes 100 chips in the beginning of the game, and each player has a total stack of 1,800 chips (including the ante).

We ran self-play multi-valued states for 8 outer-loop iterations, each consisting of 1000 self-play episodes. The value at each leaf node, after each player chooses a continuation policy, is estimated by performing 200 rollouts to the end of the game using the chosen continuation policies. The value is then cached for the rest of the subgame solve.

To evaluate performance, we ran head-to-head evaluations against a trained neural fictitious self-play (NFSP) [14] policy. We used NFSP hyperparameters from a previously published hyperparameter search on Leduc Poker [36]. We used the OpenSpiel implementation of NFSP, which uses neural nets implemented with TensorFlow [1]. We verified that the performance of the NFSP implementation matched the results in Walton and Lisy [36] for Leduc Poker. We used the same neural network architecture for NFSP and for the continuation policy nets: a fully-connected neural network with 2 hidden layers of size 512. Both NFSP and self-play multi-valued states were evaluated after being trained for 39 hours of wall-clock time.

To solve the depth-limited subgames, we used external sampling Monte Carlo CFR [19] with 5,000 iterations. We ran 30,000 episodes for each head-to-head (15,000 for each policy as player 1).

The results of the experiments are in Figure 2. We compare three methods: Self-Play Multi-Valued States (ours), Naive, and Bias. The Naive method is an agent that does search with only a single continuation policy per player: the NFSP policy. The Bias

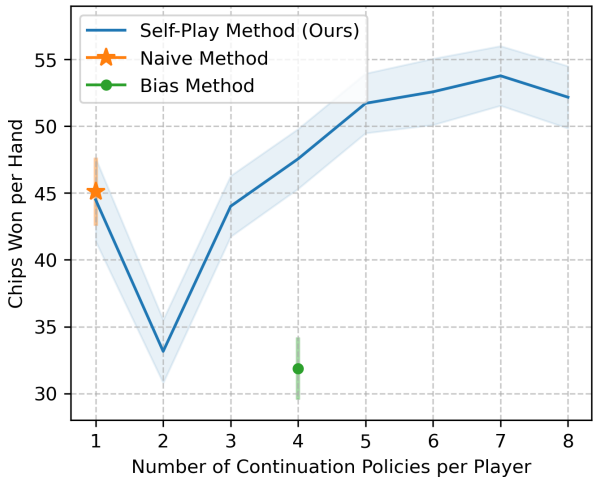


Figure 2: Head to Head Performance Against NFSP in Medium-Sized Poker Game
(Each player antes 100 chips at the start of the game)
(Shaded areas represent 95% confidence intervals)

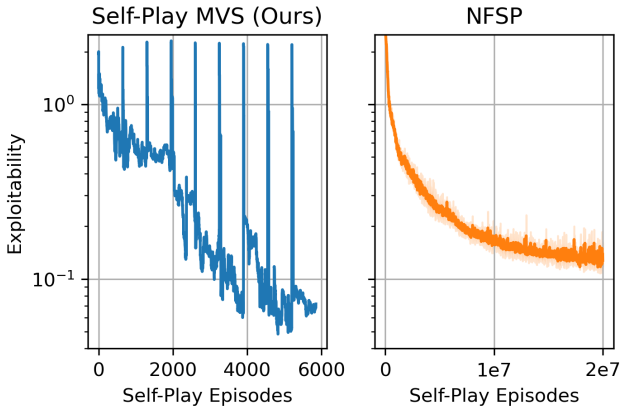


Figure 3: Exact Exploitability in Leduc Poker

method is an agent that does search with 4 continuation policies per player: the NFSP policy, the NFSP policy biased towards folding, the NFSP policy biased towards checking/calling, and the NFSP policy biased towards betting/raising.

All three search methods have a significantly positive winrate against the raw NFSP policy. However, the highest winrate is achieved by the self-play method, winning between 50 and 55 chips per hand with 5 or more continuation policies. Even when using the same amount of continuation policies as the bias method (4), our method performs significantly better.

Surprisingly, in this experiment, search with the naive method performs better than with the bias method, despite the fact that bias method's continuation policy population is a superset of the naive method's. Perhaps this is because the faulty assumption of doing search with the naive method isn't so faulty in this head-to-head setup: the equilibrium value of a leaf assumes that both players will continue playing the blueprint (the NFSP policy) until the end of the game. This is generally unsound because a player acting as if the assumption is true may become exploitable to non-blueprint policies of the opponent. However, in this specific case, the assumption is partially true: the opponent will in fact play the NFSP policy until the end of the game. Thus, assigning a value to leaf nodes based on the assumption that the opponent will only play the blueprint should actually be quite useful in head-to-head performance against an opponent that only plays the blueprint.

In view of this hypothesis, the self-play method's similar performance with $k = 1$ continuation policy is impressive, since the continuation policy has no relation to the NFSP policy it is being evaluated against.

6.2 Experiments in Small Imperfect Information Games

We also performed experiments using small games, where the exact exploitability of a policy can be easily computed: Leduc Poker and a small Liar's Dice game.

In these experiments, we perform self-play multi-valued states, and keep track of the time-averaged search policy in a tabular policy. Every time we solve a depth-limited subgame, in addition to adding the solution policy to the training data, we also update the tabular policy at every infostate in the depth-limited subgame. For a given infostate, we update the new tabular policy at that infostate to be a weighted average of the existing tabular policy and the new solution. For more details see Section 7

6.2.1 Leduc Poker. We keep track of the empirical policy played by the search agent during self-play, and calculate its exploitability. For comparison, we also ran NFSP with the same neural net architecture, and computed its exploitability. The results are plotted in Figure 3. The total wall-time for the NFSP run was 66 hours, while the total wall-time for the self-play multi-valued states run was 21 hours. The self-play multi-valued states experiment was single-threaded. (Neither algorithm's implementation is thoroughly optimized, so wall-times are presented here only to give a rough idea.) The NFSP run used hyperparameters from Walton and Lisy [36], including fully-connected neural networks with 4 hidden layers of size 128 each. The self-play multi-valued states policy neural networks used 2 hidden layers of size 512 each.

The spikes in the left subplot of Figure 3 indicate when a new outer-loop iteration began. The experiment consisted of 9 outer-loop iterations, with 650 self-play episodes in each outer-loop iteration.

These experimental results show that self-play multi-valued states produces search agents with empirically low exploitability, and provide evidence that the algorithm's search agents tend to become less and less exploitable as self-play training goes on.

6.2.2 Liar's Dice. We performed the same experiment in a small liar's dice game in OpenSpiel, with 1 die of 5 faces. The experiment results can be seen in Figure 4.

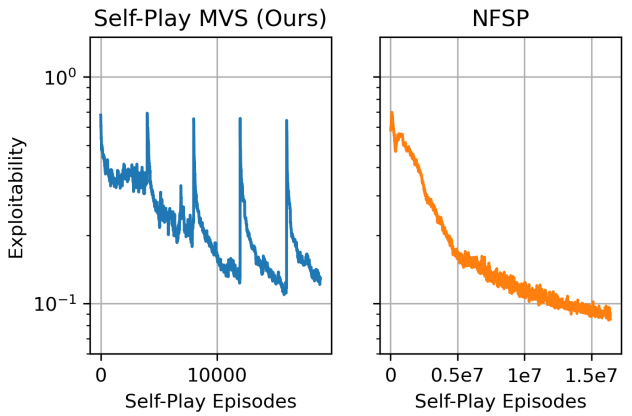


Figure 4: Exact Exploitability in Liar's Dice

In this experiment, like for Leduc, the self-play multi-valued states policy's exploitability also decreases as training progresses. Unlike in Leduc poker, the NFSP comparison ultimately achieved lower exploitability. We observed that the self-play policy seemed to plateau in exploitability near the end of training. We hypothesize that perhaps the unsoundness introduced by using nested unsafe subgame solving matters more in this game, and causes the resulting policy to be exploitable.

The NFSP agent was trained with the same hyperparameters as in Leduc poker. The self-play multi-valued states agent was trained with 6 outer-loop iterations of 4000 episodes each.

6.3 Experiments in Perfect Information Game

To demonstrate that self-play multi-valued states converges to a Nash equilibrium policy in the first outer-loop iteration, we ran the algorithm on a tiny perfect-information game: Hex with 3 rows and 2 columns. The game has 1,333 histories.

The experiment results can be seen in Figure 5. Within 500 self-play episodes, the empirical time-averaged policy of each of the runs converged to an epsilon Nash equilibrium (with an exploitability of less than 10^{-3}).

7 EXPERIMENT DETAILS

The medium-sized poker game we used for the experiments in Section 6.1 was parameterized in OpenSpiel with:

```
{"numSuits": 2, "stack": "1800 1800", "numRanks": 5}
```

The self-play multi-valued states training was done using 1 GPU. For the experiments in medium-sized poker, self-play was parallelized across 6 CPU cores, and training was single-threaded.

During self-play, we used an exploration term $\alpha = 0.1$. For any player's move, there was a probability of α that a move would be selected at uniform random, instead of according to the subgame solution.

In our experiments, self-play produced training data in the form of pairs of infostate tensors (the inputs) and distributions over legal actions (the inputs). Infostate tensors are representations of infostates which are suitable as neural network inputs, and in our experiments we used the default OpenSpiel implementations to get them. Our neural networks were trained to minimize the KL-divergence loss function over the logarithmic softmax of the neural network's outputs (after masking out illegal actions), using PyTorch's `KLDivLoss`. The loss was weighted by multiplying the loss by the reach probability of the acting player.

In the exact exploitability experiments, every time we solve a depth-limited subgame, we update the tabular policy at every infostate in the depth-limited subgame. For a given infostate, we update the new tabular policy at that infostate to be a weighted average of the existing tabular policy and the new solution. The weight of the existing tabular policy is initially 0 at every infostate. The weight of the new solution is equal to the player's probability of reaching that infostate. The weight of the existing tabular policy is then incremented by the weight of the new solution. The weight of the existing tabular policy is decayed by multiplying by $n/(n+1)$ where n is the inner-loop iteration (similar to linear CFR [7]) and by multiplying by a constant of 0.95.

The code used for these experiments will be open-sourced.

8 CONCLUSIONS

We present self-play multi-valued states, a new method using self-play to train continuation policies for multi-valued states from scratch. We show that it performs better in our experiments than the previous "bias method" for multi-valued states. We also show in experiments for small games that its exploitability decreases as self-play training progresses. Finally, we demonstrate that in perfect information games, the algorithm reduces to an AlphaZero-like algorithm which converges to the equilibrium policy by training just a single continuation policy per player.

8.1 Future Work

In this work, we used head-to-head playthroughs to evaluate performance of search policies in medium-sized games. Although we

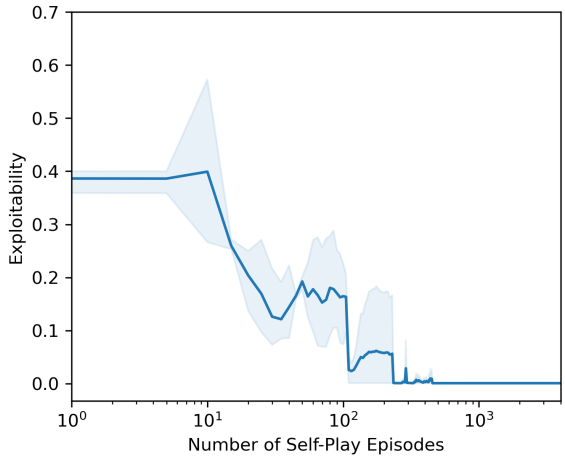


Figure 5: Exploitability of Time-Averaged Self-Play Policy in 3x2 Hex, Single Outer-Loop Iteration
(Shaded areas represent min/max of 3 runs with different random seeds)

would ideally like to measure the exploitability instead of head-to-head performance, it is intractable to calculate exact exploitability of search-based policies in anything other than small games.

Though expensive, it is possible to approximate the exploitability of a policy using reinforcement learning methods [34]. Doing so on the algorithm from this work will give a more complete picture of its exploitability than just looking at head-to-head performance.

In this work, we only ran experiments in games with no more than 100,000 states. The algorithm can be scaled up just by modifying the policy net architecture to run on games with more public states (such as large poker and liar's dice games).

This paper presents a novel method of generating populations of continuation policies for depth-limited search with multi-valued states. This method empirically improves on previous methods of generating these populations. However, it remains an open question whether or not this method is provably superior. And as with other algorithms that involve populations of policies (like double oracle), this work invites the question: can we do better? For example, is it possible to create an algorithm where the newest policy we add to the population is the policy that maximally decreases the exploitable of the search agent?

ACKNOWLEDGMENTS

If you wish to include any acknowledgments in your paper (e.g., to people or funding agencies), please do so using the 'acks' environment. Note that the text of your acknowledgments will be omitted if you compile your document with the 'anonymous' option.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Lukáš Adam, Rostislav Horčík, Tomáš Kasl, and Tomáš Kroupa. 2020. Double Oracle Algorithm for Computing Equilibria in Continuous Games. (09 2020). https://arxiv.org/abs/2009.12185

[3] Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems* 30 (2017).

[4] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. 2020. Combining deep reinforcement learning and search for imperfect-information games. *Advances in Neural Information Processing Systems* 33 (2020), 17057–17069.

[5] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. 2019. Deep counterfactual regret minimization. In *International conference on machine learning*. PMLR, 793–802.

[6] Noam Brown and Tuomas Sandholm. 2017. Safe and nested subgame solving for imperfect-information games. *Advances in neural information processing systems* 30 (2017).

[7] Noam Brown and Tuomas Sandholm. 2019. Solving imperfect-information games via discounted regret minimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1829–1836.

[8] Noam Brown and Tuomas Sandholm. 2019. Superhuman AI for multiplayer poker. *Science* 365, 6456 (2019), 885–890.

[9] Noam Brown, Tuomas Sandholm, and Brandon Amos. 2018. Depth-limited solving for imperfect-information games. *Advances in neural information processing systems* 31 (2018).

[10] Neil Burch, Michael Johanson, and Michael Bowling. 2014. Solving imperfect information games using decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 28.

[11] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. 2002. Deep blue. *Artificial intelligence* 134, 1-2 (2002), 57–83.

[12] Sam Ganzfried and Tuomas Sandholm. 2015. Endgame solving in large imperfect-information games. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. 37–45.

[13] Audrūnas Gruslys, Marc Lanctot, Rémi Munos, Finbarr Timbers, Martin Schmid, Julien Perolat, Dustin Morrill, Vinicius Zambaldi, Jean-Baptiste Lespiau, John Schultz, et al. 2020. The advantage regret-matching actor-critic. *arXiv preprint arXiv:2008.12234* (2020).

[14] Johannes Heinrich and David Silver. 2016. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121* (2016).

[15] Daniel Hennes, Dustin Morrill, Shayegan Omidshafiei, Rémi Munos, Julien Perolat, Marc Lanctot, Audrunas Gruslys, Jean-Baptiste Lespiau, Paavo Parmas, Edgar Duéñez-Guzmán, et al. 2020. Neural replicator dynamics: Multiagent learning via hedging policy gradients. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. 492–501.

[16] Vojtěch Kovařík, Martin Schmid, Neil Burch, Michael Bowling, and Viliam Lisỳ. 2022. Rethinking formal models of partially observable multiagent decision making. *Artificial Intelligence* 303 (2022), 103645.

[17] Vojtěch Kovařík, Dominik Seitz, Viliam Lisỳ, Jan Rudolf, Shuo Sun, and Karel Ha. 2023. Value functions for depth-limited solving in zero-sum imperfect-information games. *Artificial Intelligence* 314 (2023), 103805.

[18] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. 2019. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR* abs/1908.09453 (2019). arXiv:1908.09453 [cs.LG] http://arxiv.org/abs/1908.09453

[19] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. 2009. Monte Carlo sampling for regret minimization in extensive games. *Advances in neural information processing systems* 22 (2009).

[20] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. 2017. A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems* 30 (2017).

[21] Stephen McAleer, JB Lanier, Kevin Wang, Pierre Baldi, Roy Fox, and Tuomas Sandholm. 2022. Self-play psro: Toward optimal populations in two-player zero-sum games. *arXiv preprint arXiv:2207.06541* (2022).

[22] Stephen McAleer, John B Lanier, Kevin A Wang, Pierre Baldi, and Roy Fox. 2021. XDO: A double oracle algorithm for extensive-form games. *Advances in Neural Information Processing Systems* 34 (2021), 23128–23139.

[23] Stephen Marcus McAleer, Gabriele Farina, Marc Lanctot, and Tuomas Sandholm. [n.d.]. ESCHER: Eschewing Importance Sampling in Games by Computing a History Value Function to Estimate Regret. In *Deep Reinforcement Learning Workshop NeurIPS 2022*.

[24] Brendan McMahan, Geoffrey Gordon, and Avrim Blum. 2003. Planning in the Presence of Cost Functions Controlled by an Adversary. In *International conference on machine learning*. 536—-543.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[26] Claude E Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275.

[27] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[28] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[29] Samuel Sokota, Ryan D'Orazio, J Zico Kolter, Nicolas Loizou, Marc Lanctot, Ioannis Mitliagkas, Noam Brown, and Christian Kroer. 2022. A unified approach to reinforcement learning, quantal response equilibria, and two-player zero-sum games. *arXiv preprint arXiv:2206.05825* (2022).

[30] Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. 2005. Bayes' Bluff: Opponent Modelling in Poker. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence* (Edinburgh, Scotland) *(UAI'05)*. AUAI Press, Arlington, Virginia, USA, 550–558.

[31] Eric Steinberger, Adam Lerer, and Noam Brown. 2020. DREAM: Deep regret minimization with advantage baselines and model-free learning. *arXiv preprint arXiv:2006.10410* (2020).

[32] Oskari Tammelin. 2014. Solving large imperfect information games using CFR+. *arXiv preprint arXiv:1407.5042* (2014).

[33] Gerald Tesauro et al. 1995. Temporal difference learning and TD-Gammon. *Commun. ACM* 38, 3 (1995), 58–68.

[34] Finbarr Timbers, Nolan Bard, Edward Lockhart, Marc Lanctot, Martin Schmid, Neil Burch, Julian Schrittwieser, Thomas Hubert, and Michael Bowling. 2020. Approximate exploitability: Learning a best response in large games. *arXiv preprint arXiv:2004.09677* (2020).

[35] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019), 350–354.

[36] Michael Walton and Viliam Lisy. 2021. Multi-agent Reinforcement Learning in OpenSpiel: A Reproduction Report. *arXiv preprint arXiv:2103.00187* (2021).

[37] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. 2007. Regret minimization in games with incomplete information. *Advances in neural information processing systems* 20 (2007).