Targeted Search Control in AlphaZero for Effective Policy Improvement

Alexandre Trudeau
University of Alberta
Edmonton, Canada
trudeau1@ualberta.ca

Michael Bowling
University of Alberta
Edmonton, Canada
mbowling@ualberta.ca

ABSTRACT

AlphaZero is a self-play reinforcement learning algorithm that achieves superhuman play in chess, shogi, and Go via policy iteration. To be an effective policy improvement operator, AlphaZero's search requires accurate value estimates for the states appearing in its search tree. AlphaZero trains upon self-play matches beginning from the initial state of a game and only samples actions over the first few moves, limiting its exploration of states deeper in the game tree. We introduce Go-Exploit, a novel search control strategy for AlphaZero. Go-Exploit samples the start state of its self-play trajectories from an archive of *states of interest*. Beginning self-play trajectories from varied starting states enables Go-Exploit to more effectively explore the game tree and to learn a value function that generalizes better. Producing shorter self-play trajectories allows Go-Exploit to train upon more independent value targets, improving value training. Finally, the exploration inherent in Go-Exploit reduces its need for exploratory actions, enabling it to train under more exploitative policies. In the games of Connect Four and 9x9 Go, we show that Go-Exploit learns with a greater sample efficiency than standard AlphaZero, resulting in stronger performance against reference opponents and in head-to-head play. We also compare Go-Exploit to KataGo, a more sample efficient reimplementation of AlphaZero, and demonstrate that Go-Exploit has a more effective search control strategy. Furthermore, Go-Exploit's sample efficiency improves when KataGo's other innovations are incorporated.

KEYWORDS

AlphaZero; Search Control; Reinforcement Learning; Planning

ACM Reference Format:
Alexandre Trudeau and Michael Bowling. 2023. Targeted Search Control in AlphaZero for Effective Policy Improvement. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 9 pages.

1 INTRODUCTION

AlphaZero [11–13] is a model-based reinforcement learning (RL) algorithm that has achieved impressive results in two-player, zero-sum games, reaching superhuman play in chess, shogi, and Go. AlphaZero simulates self-play matches with a perfect model of its environment (the rules of the game) to train a neural network that learns a value function and action selection priors over states. Each turn, the value function and priors guide a lookahead search that returns an improved policy. AlphaZero trains its neural network on the self-play matches produced under the improved policies, enabling it to improve its play via policy iteration.

Despite its success, AlphaZero's training suffers from sample inefficiency. In 19x19 Go, AlphaZero requires hundreds of millions of training samples to attain superhuman play ([12], Figure 1c). AlphaZero's sample efficiency depends upon the distribution of states visited and trained upon. Although AlphaZero has a perfect model of its environment, it cannot feasibly visit and learn the optimal value for each state. Instead, AlphaZero trains upon the states that it visits on-policy in simulated self-play matches beginning from the initial state of the game. As in other RL algorithms [14], AlphaZero takes exploratory actions during its self-play matches so that it can train upon a variety of states, enabling it to make more informed action selections in the future. AlphaZero employs simplistic exploration mechanisms during self-play training: randomly perturbing the learned priors guiding search and stochastically selecting actions near the start of self-play matches. As a result, AlphaZero's training procedure exhibits the following limitations:

(1) Since AlphaZero begins its self-play matches from the initial state of a game, it often transitions into a terminal state before reaching and exploring states deeper in the game tree. In addition, AlphaZero only samples actions over the first few moves of a self-play match, further limiting exploration deeper in the game tree.
(2) AlphaZero's exploration mechanisms cause it to train under weaker, exploratory policies, slowing policy iteration.
(3) AlphaZero only produces a single, noisy value target from a full self-play match, slowing value training.

We hypothesized that AlphaZero could address these limitations, and learn with greater sample efficiency, with a more effective search control strategy. Sutton and Barto define search control as "the process that selects the starting states and actions for the simulated experiences generated by the model" [14]. In AlphaZero, this would amount to strategically choosing the starting state of its simulated trajectories. We propose one such strategy that adheres to four guiding principles. The algorithm should:

(a) Continually visit new states throughout the state space to learn their values and a good policy.
(b) Keep track of *states of interest* and have the ability to reliably revisit them for further exploration.
(c) Limit exploration's bias in the learning targets.
(d) Produce more independent value targets to train upon.

In this paper, we introduce Go-Exploit, a novel search control strategy for AlphaZero. Go-Exploit takes inspiration from Go-Explore [3] and Exploring Restart Distributions [15], which begin simulated episodes from previously visited states sampled from a memory. Similarly, Go-Exploit maintains an archive of *states of interest*. At

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom.

the beginning of a self-play trajectory, the start state is either uniformly sampled from the archive or is set to the initial state of the game. Two factors influencing Go-Exploit's performance are the definition of *states of interest* and the structure of the archive. In this paper, we experiment with two definitions of *states of interest* and three archive structures.

In the games of Connect Four and 9x9 Go, we show that Go-Exploit exhibits a greater sample efficiency than standard AlphaZero, measured in their average win rates against reference opponents over the course of training and in the results of their head-to-head play. We also compare and contrast Go-Exploit and KataGo [20], a more sample efficient reimplementation of AlphaZero. Go-Exploit's search control strategy results in faster learning than KataGo's. Furthermore, Go-Exploit's sample efficiency improves when KataGo's other innovations are incorporated. We conclude by showing how Go-Exploit's adherence to the guiding principles enables it to learn more effectively than AlphaZero.

2 ALPHAZERO

AlphaZero [12] represents its learned game knowledge with a neural network $f_\theta(s) = (\mathbf{p}, v)$ parameterized by weights θ. f_θ takes a state s as input and outputs a value estimate v estimating the expected game outcome from state s under AlphaZero's current policy. f_θ also outputs a vector of action selection priors $\mathbf{p}$ estimating AlphaZero's current policy from state s.

AlphaZero improves its play by training upon simulated matches played against itself. These self-play matches begin from the initial state of a game s_0. On each turn t, AlphaZero performs a variant of Monte Carlo Tree Search [1, 2, 4], inspired by PUCB [8], to determine the action a_t that is played. The search tree is initialized with a root node corresponding to s_t and a set of edges representing the legal actions. Each edge stores a set of statistics $\{N(s,a), Q(s,a), P(s,a)\}$. $N(s,a)$ is the number of times (s,a) has been traversed during the given search. $Q(s,a)$ is the backed up action-value estimate of (s,a). $P(s,a)$ is the prior probability of selecting action a from state s. In each search iteration, the search tree is traversed from the root node using PUCT action selection:

$$a = \arg\max_a Q(s,a) + c_{\text{puct}} P(s,a) \frac{\sqrt{N(s)}}{1 + N(s,a)}$$

$N(s)$ is the number of times state s has been visited during the given search and $c_{\text{puct}} > 0$ is an exploration constant. This action selection rule encourages the search to traverse state-action pairs with large action-value estimates $Q(s,a)$, large priors $P(s,a)$, and few search visits $N(s,a)$. Once the search traverses a state-action pair (s_L, a) with $N(s_L, a) = 0$, the successor state s' is added as a child of s_L and f_θ runs inference on the new state: $f_\theta(s') = (\mathbf{p}, v)$. The edge statistics of the legal actions that can be taken from s' are initialized as follows: $\{N(s',a) = 0, Q(s',a) = 0, P(s',a) = p_a\}$, where p_a is the component of $\mathbf{p}$ corresponding to action a. Next, the value estimate v is backed up to the state-action pairs that were traversed in the given iteration to update their action-values $Q(s,a)$. $Q(s,a)$ averages the value estimates v of the states in the subtree of (s,a). It estimates the expected outcome from (s,a) based on the value estimates of the likeliest successor states.

Once the final search iteration is complete, the search returns a policy π_t. The components of π_t depend upon the distribution of search visits over the root state's actions: $\pi_t(a|s_t) = \frac{N(s_t,a)^{1/\tau}}{\sum_b N(s_t,b)^{1/\tau}}$, where $\tau > 0$ is a Softmax temperature. In the first k moves of a self-play match, action a_t is sampled from π_t. After the first k moves, AlphaZero aims to be more exploitative and plays the action that was most visited during search. When a self-play match reaches a terminal state s_T with outcome z, AlphaZero produces training samples (s_t, π_t, z) that are added to an experience replay buffer B [6, 7] with fixed size $|B|$. Once b_{step} new training samples have been added to the replay buffer, b_{batch} training tuples are uniformly sampled to update f_θ. The neural network's parameters θ are updated via stochastic gradient descent on the loss function

$$\text{loss} = (z - v)^2 - \pi_t^T \log(\mathbf{p}) + c||\theta||^2$$

where c is a regularization constant. Once f_θ is updated, the next learning step begins.

Training f_θ's policy head on the policies π_t and the value head on the self-play match outcomes z brings about policy iteration, enabling AlphaZero to learn stronger policies. AlphaZero's search is a policy improvement operator because it concentrates the search visits on the root actions with the largest action-values $Q(s,a)$. This brings about policy improvement as long as the value estimates used in search are sufficiently accurate under the current policy. Then, AlphaZero selects an action a_t with respect to the improved policy π_t. Training f_θ's value head on outcomes produced under the improved policies enables policy evaluation to be with respect to the improved policy. These alternating processes of policy improvement and policy evaluation enable AlphaZero to learn stronger policies over time. However, the scarcity of independent value targets z relative to the policy targets π_t can slow AlphaZero's value training and its subsequent ability to produce improved policies.

2.1 Exploration in AlphaZero

The accuracy of f_θ's value estimates depends upon the distribution of states visited and trained upon. To have accurate value estimates for the diverse set of states that appear during search, AlphaZero must explore the state space during training. AlphaZero ensures exploration by introducing stochasticity into its action selection.

In its search, AlphaZero perturbs the priors over the root node's actions with noise. When the root node s_r is evaluated by the neural network $f_\theta(s_r) = (\mathbf{p}, v)$, the vector of action probabilities $\mathbf{p}$ is perturbed by a noise vector $\mathbf{d} \sim \text{Dir}(\alpha)$ sampled from a Dirichlet distribution. The perturbed priors $P(s_r, a)$ are computed using the equation $P(s_r, a) = (1-\epsilon)p_a + \epsilon d_a$, where p_a and d_a are components of $\mathbf{p}$ and $\mathbf{d}$, respectively, and $0 < \epsilon < 1$. Randomly perturbing the priors over the root node's actions causes the policy π_t returned by search to also be perturbed, introducing randomness in AlphaZero's action selection during self-play.

AlphaZero also achieves exploration through action sampling. Upon the completion of a search, the search visits over the root node's actions are converted into a policy $\pi_t(a|s_t) = \frac{N(s_t,a)^{1/\tau}}{\sum_b N(s_t,b)^{1/\tau}}$. The Softmax temperature τ helps control the level of exploration vs. exploitation in the produced policies. When $\tau = 1.0$, the components of the policy π_t are directly proportional to the search visits over the root state's actions. When $\tau < 1.0$, the policies produced concentrate a greater portion of the probability on the most visited root actions and are, therefore, more exploitative. When $\tau > 1.0$,

the policies produced are more uniform, and thus, more exploratory. For the first k moves of a self-play game, the action that is played is sampled: $a_t \sim \pi_t$. Sampling actions proportionally to the search visit counts ensures that a variety of actions are tried from a given state, while still favouring the selection of actions that had large action-values $Q(s, a)$ and large priors $P(s, a)$.

The stochasticity in AlphaZero's action selection presents an exploration-exploitation trade-off. On the one hand, the stochasticity allows AlphaZero to perform policy evaluation at a diverse set of states, improving the accuracy of the value estimates used during search. This enables AlphaZero's search to be a more effective policy improvement operator. On the other hand, the stochasticity causes AlphaZero to generate self-play matches under weaker exploratory policies. This causes policy evaluation to be with respect to the weaker policies and for policy improvement to be with respect to policies π_t perturbed with Dirichlet noise, slowing down policy iteration. AlphaZero manages this exploration-exploitation trade-off with the temperature τ, the number of action sampling moves k, and with ϵ, which controls the magnitude of the Dirichlet noise. These hyperparameters must be set large enough to ensure that AlphaZero sufficiently explores the state space, however, they cannot be so large that AlphaZero learns weak policies. This leads action sampling to only take place at the beginning of self-play matches, limiting the exploration of states later in games.

3 GO-EXPLOIT

Given the limitations we identified in AlphaZero's training procedure, we adopted the guiding principles in section 1 in designing a new search control strategy for AlphaZero. We took inspiration from Go-Explore [3] and Exploring Restart Distributions [15] by incorporating an archive of *states of interest* in AlphaZero. Our algorithm, called Go-Exploit, modifies AlphaZero by beginning self-play trajectories from *states of interest* sampled from this archive. This enables Go-Exploit to reliably revisit *states of interest* throughout the game tree (guiding principle (b)) and to complete more self-play trajectories per learning step (guiding principle (d)). Then, the remainder of the self-play trajectory is produced identically to AlphaZero. However, Go-Exploit applies AlphaZero's exploration mechanisms of action sampling and Dirichlet noise from trajectories beginning throughout the game tree, enabling Go-Exploit to effectively explore the state space (guiding principle (a)). Since exploration is built into the "Go" step of sampling the start state of a self-play trajectory, we anticipated that Go-Exploit would require less stochasticity in its action selection than AlphaZero, enabling it to learn under more exploitative policies (guiding principle (c)). In this paper, we explore this approach while experimenting with two definitions of *states of interest* and three archive structures to see how they respectively impact the sample efficiency of Go-Exploit.

The way *states of interest* is defined affects the performance of Go-Exploit because it changes the distribution of states that f_θ is trained upon. *Go-Exploit Visited States* considers nonterminal states visited during self-play games as states of interest because we want action selection to improve from the states visited under AlphaZero's current policy. *Go-Exploit Search States* considers nonterminal search states appearing in trajectories beginning from s_0 as states of interest because their value estimates influence the policies π_t returned by search. Each variant of Go-Exploit samples start states from the archive uniformly at random. Since the archive can contain multiple copies of a state, it favours the selection of states that are frequently visited or observed during search.

3.1 Go-Exploit Visited States

Go-Exploit Visited States makes simple modifications to AlphaZero. First, it initializes an archive A with the initial state of a game s_0. This archive is shared amongst *training actors* that generate self-play trajectories. At the beginning of each self-play trajectory, Go-Exploit uniformly samples a random number $r \in [0, 1]$. If $r < \lambda$, Go-Exploit begins its self-play trajectory from s_0. If $r \geq \lambda$, Go-Exploit begins its self-play trajectory from a *state of interest* uniformly sampled from the archive. Second, Go-Exploit samples actions from π_t for the first k moves of a self-play trajectory regardless of whether the trajectory begins at s_0. Finally, once a self-play trajectory completes, Go-Exploit Visited States adds the nonterminal states that were visited to the archive A. We experimented with two variants of Go-Exploit Visited States using two different archives. Go-Exploit Visited States Expanding Archive (GEVE) uses an expanding archive consisting of every visited state. Go-Exploit Visited States Circular Archive (GEVC) employs a fixed-size circular archive consisting of the most recently visited states.

3.2 Go-Exploit Search States

Go-Exploit Search States makes similar modifications to AlphaZero. It also employs *training actors* that sample start states from A and produce training data for f_θ, however, they do not add visited states nor search states to the archive. Go-Exploit Search States, instead, concurrently runs *archive actors* responsible for populating the archive. The archive actors always play out complete self-play matches beginning from s_0. Once an archive actor's self-play match is complete, it adds all of the nonterminal states that appeared during search into archive A. We experimented with two variants of Go-Exploit Search States. Go-Exploit Search States Reservoir Archive (GESR) uses a fixed-size archive and Reservoir Sampling [18] to determine which states are added/removed from the archive. Reservoir Sampling approximates the distribution of states that would be included in the Expanding Archive. The Expanding Archive is not always feasible due to memory constraints. Go-Exploit Search States Circular Archive (GESC) employs a fixed-size circular archive consisting of the most recently observed search states. Pseudocode for each variant of Go-Exploit can be found in the appendix [17].

3.3 Related Work

Although Go-Exploit is inspired by Go-Explore [3], the two algorithms work very differently. In Go-Explore, the "Go" step is exploitative because it loads a start state associated with a high scoring trajectory. Exploratory actions are taken from this state to discover higher scoring trajectories. In Go-Exploit, on the other hand, the "Go" step is exploratory because it begins self-play trajectories from states throughout the game tree. Due to the exploration inherent in the sampling of the start states, Go-Exploit can then produce the remainder of its self-play trajectories under more exploitative policies. Hence the name Go-Exploit.

Go-Exploit extends Exploring Restart Distributions (ERDs) [15] to the new setting of AlphaZero. ERDs maintains a *restart memory* of visited states and combines it with the environment's initial state distribution to form the starting state distribution in a simulated environment. Go-Exploit Visited States is analogous to *Uniform Restart*, which uniformly samples the initial state of an episode from the circular restart memory. However, beginning self-play trajectories from previously visited states may not result in the most efficient learning. Go-Exploit Search States extends ERDs beyond visited states. Go-Exploit deliberately uses the notion of *states of interest* when defining which states to include in its archive to allow the inclusion of states that have never been explicitly visited. This enables Go-Exploit Search States to focus its planning updates on successor states appearing in search whose value estimates influence the returned policies.

In MuZero [10], the successor to AlphaZero that plans with a learned model, greater sample efficiency is also achieved via search control. The authors introduce a variant of MuZero, called *MuZero Reanalyze*, that revisits previously visited states and performs a new search with the latest model. The model is then trained upon the new policy and value targets returned by the search. MuZero Reanalyze and Go-Exploit Visited States are similar in that they both plan from previously visited states. However, MuZero Reanalyze does not simulate new self-play trajectories from these previously visited states, limiting its exploration of the state space.

KataGo [20] also incorporates search control into the AlphaZero framework. We will describe KataGo's search control procedure in Section 5 and then evaluate its sample efficiency relative to Go-Exploit.

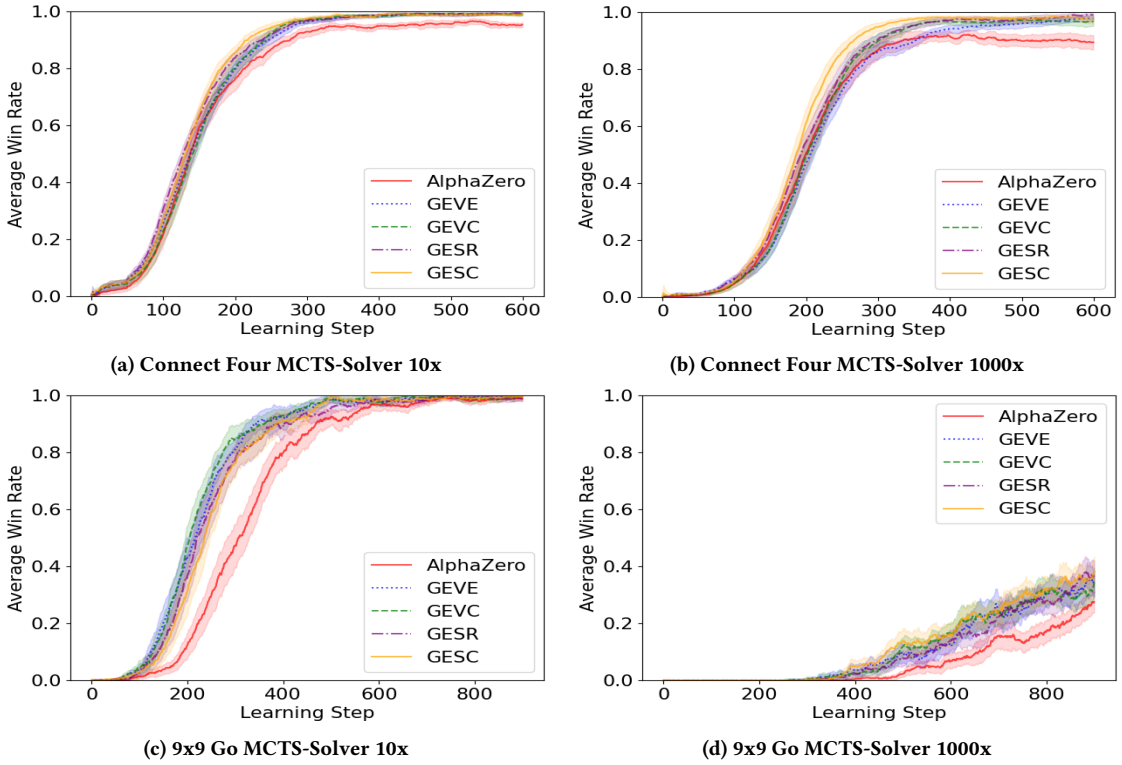
4 EVALUATING GO-EXPLOIT

Experiments were conducted in Connect Four and 9x9 Go to evaluate the sample efficiencies of the four variants of Go-Exploit relative to AlphaZero and KataGo and to understand how the changes introduced in Go-Exploit impact policy iteration. By performing experiments in Connect Four and 9x9 Go, we evaluated Go-Exploit in two domains with different characteristics and sizes. Connect Four has a smaller state space than 9x9 Go but has a greater percentage of terminal states in its game tree. Go-Exploit and elements of KataGo were coded on top of DeepMind's OpenSpiel [5] implementation of AlphaZero (our code is publicly available [16]). Experiments were run using OpenSpiel's versions of Connect Four and Go. Each experiment was run on a node with 48 cores (2 x AMD Milan 7413), 4 NVIDIA A100 gpus, and 498G of memory.

To compare the sample efficiencies of AlphaZero, Go-Exploit, and KataGo, we had to choose a metric measuring sample efficiency. Sample efficiency can be measured by the average performance achieved over a computational budget. We chose this metric because it is less sensitive to training horizon, it accounts for how effectively an algorithm learns throughout training, and it differentiates between algorithms achieving similar asymptotic performance. The average performance over a computational budget can be represented by the "area under the curve" (AUC) in a performance vs. learning step graph. In our experiments, training runs lasted a fixed computational budget of 600 learning steps in Connect Four and 900 learning steps in 9x9 Go. An algorithm's performance was measured by its win rate against a fixed reference opponent called MCTS-Solver [19]. Over the course of training, 50 evaluator threads played evaluation matches against different difficulty levels of MCTS-Solver with 1x, 10x, 100x, and 1000x as many search iterations as AlphaZero, Go-Exploit, and KataGo. Equal numbers of evaluation matches were played as player 1 and player 2. Wins, draws, and losses were scored 1, 0.5, and 0, respectively. After each learning step, the win rate against each difficulty level of MCTS-Solver was computed by averaging the evaluation match results over the previous 50 learning steps. In our hyperparameter sweeps, 10 independent training runs were executed for each hyperparameter setting with randomly chosen seeds. Upon their completion, the win rates against each difficulty level of MCTS-Solver were averaged at each learning step. The average win rates were summed over all learning steps to compute the AUC over the computational budget. Ultimately, the AUC achieved against MCTS-Solver 10x was used to select hyperparameter values. To compare the sample efficiencies of AlphaZero, Go-Exploit, and KataGo, an additional 30 validation runs were conducted using the best hyperparameter settings. The AUCs against MCTS-Solver 1x, 10x, 100x, and 1000x were computed to see how the algorithms performed against different fixed reference opponents.

For each algorithm and variant, we swept over the following hyperparameters: the learning rate lr of f_θ, the regularization constant c of the loss function, the Dirichlet distribution parameter α, the constant ϵ affecting the magnitude of the Dirichlet noise, the exploration constant c_{puct} in search, the number of action sampling moves k, the probability λ of beginning self-play trajectories from s_0, the archive type A_{type}, the archive size $|A|$, and the Softmax temperature τ. One hyperparameter was swept over at a time. The exact values swept over and the best performing hyperparameter values appear in the appendix [17]. There are additional hyperparameters involved in AlphaZero that do not directly affect the distribution of states visited and trained upon, and thus, are not pertinent to our main investigation. We chose sensible values for our experimental setup that also showed visible learning progress in our domains and held them fixed for all algorithms and variants. This includes the architecture of f_θ, the number of threads, and batch sizes. These values also appear in the appendix [17].

Once the 30 validation runs were executed, we produced the learning curves appearing in Figure 1 with shaded 95% confidence intervals. In Figures 1a and 1b, we observe that in Connect Four, the four variants of Go-Exploit achieve greater AUCs than AlphaZero against MCTS-Solver. Early on in training, AlphaZero and the four variants of Go-Exploit exhibit similar learning rates, but as training progresses, AlphaZero's learning curve levels off to a lower asymptotic win rate. It should be noted that during the hyperparameter sweeps, we observed that AlphaZero could match Go-Exploit's asymptotic win rate with different hyperparameter values but at the cost of a lower AUC (i.e., AlphaZero attains this asymptotic win rate too slowly). These results suggest that Go-Exploit is able to learn more efficiently than AlphaZero in Connect Four. Figures 1a and 1b also illustrate that Go-Exploit achieves even greater AUCs in Connect Four when including search states in its archive rather than visited states. Furthermore, greater sample efficiency is realized when Go-Exploit utilizes a Circular Archive focusing training on the states observed under the most recent policies. In Figures 1c and



(a) Connect Four MCTS-Solver 10x

(b) Connect Four MCTS-Solver 1000x

(c) 9x9 Go MCTS-Solver 10x

(d) 9x9 Go MCTS-Solver 1000x

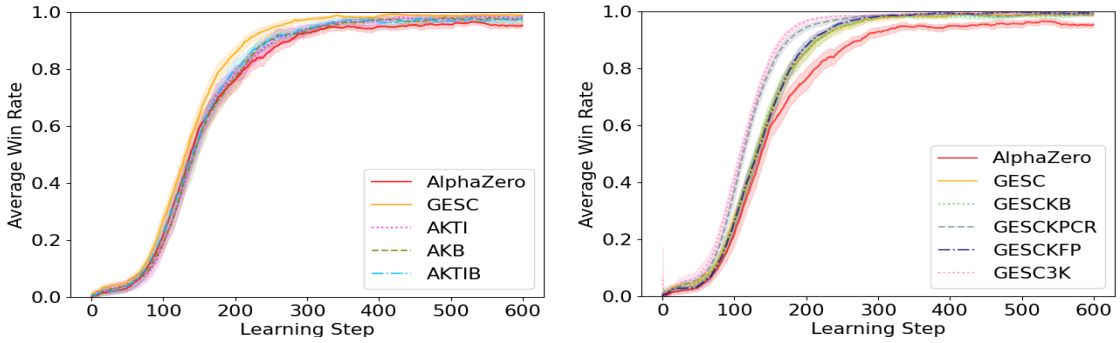
Figure 1: AlphaZero and Go-Exploit's win rates against MCTS-Solver 10x and 1000x in Connect Four and 9x9 Go. The win rates were averaged over the 30 validation runs and the shaded regions represent 95% confidence intervals.

1d, we observe that the four variants of Go-Exploit achieve much greater AUCs than AlphaZero in 9x9 Go. Go-Exploit exhibits its superior learning efficiency early on in training with much steeper learning curves than AlphaZero. Ultimately, AlphaZero and the four variants of Go-Exploit reach similar asymptotic win rates. Figure 1c suggests that Go-Exploit may learn marginally faster with visited states rather than search states in the archive. Furthermore, Go-Exploit Visited States obtains a slightly greater AUC with a Circular Archive rather than an Expanding Archive. Comparing the plots in Figure 1 also reveals that Go-Exploit's gain in sample efficiency is much greater in 9x9 Go than in Connect Four. This suggests that Go-Exploit's gains in sample efficiency may be greater in larger games. This may be due to the fact that when the search space increases in size, AlphaZero wastes more samples to reach new states deeper in the game tree.

To further measure Go-Exploit's learning efficiency relative to AlphaZero, we conducted head-to-head matches between AlphaZero and each variant of Go-Exploit. Head-to-head matches were played using the validation runs' saved neural network checkpoints from learning steps 300 and 600 in Connect Four and learning steps 300 and 900 in 9x9 Go. Each algorithm's 30 neural network checkpoints played the other algorithms' 30 neural network checkpoints in one game as player 1 and one game as player 2. The win rates from the Connect Four and 9x9 Go tournaments appear in Table 1. Each row consists of two algorithms – Algorithm 1 and Algorithm 2. The remaining entries in the row show Algorithm 1's win rate against Algorithm 2 at the respective checkpoints in Connect Four and 9x9 Go. These results reaffirm what was observed in Figure 1. In the Connect Four tournament, GESC outperformed AlphaZero and the other variants of Go-Exploit at checkpoint 300, reflecting its superior win rate at learning step 300 in Figures 1a and 1b. At checkpoint 600, each variant of Go-Exploit outperformed AlphaZero but

Table 1: Algorithm 1's win rates in head-to-head matches

Algorithm		Connect Four Checkpoint		9x9 Go Checkpoint	
1	2	300	600	300	900
GEVE	AlphaZero	0.538	0.643	0.790	0.641
GEVC	AlphaZero	0.483	0.593	0.795	0.655
GESR	AlphaZero	0.513	0.603	0.790	0.670
GESC	AlphaZero	0.582	0.632	0.753	0.652
GESC	GEVE	0.565	0.515	0.471	0.506
GESC	GEVC	0.601	0.530	0.400	0.469
GESC	GESR	0.605	0.519	0.536	0.532
GESR	GEVE	0.505	0.493	0.516	0.502
GESR	GEVC	0.496	0.502	0.436	0.488
GEVC	GEVE	0.483	0.495	0.509	0.488



(a) AlphaZero with KataGo's search control strategy **(b) Go-Exploit with three of KataGo's modifications**

Figure 2: (a) Comparing the learning speeds of AKTI, AKB, and AKTIB to standard AlphaZero and GESC. (b) Comparing the learning speeds of GESCKB, GESCKPCR, GESCKFP, and GESC3K to standard AlphaZero and GESC. Both plots show the win rates against MCTS-Solver 10x in Connect Four. The shaded regions represent 95% confidence intervals.

none stood out against each other. This mirrors the fact that the four variants of Go-Exploit achieved similar asymptotic win rates that were higher than AlphaZero's in Figures 1a and 1b. In the 9x9 Go tournament, each variant of Go-Exploit dominated AlphaZero at checkpoints 300 and 900. This is consistent with Go-Exploit's superior win rate against MCTS-Solver in Figures 1c and 1d. At checkpoint 300, GEVC outperformed both variants of Go-Exploit Search States and marginally beat GEVE, reflecting its superior win rate early on in training in Figure 1c.

5 GO-EXPLOIT VS. KATAGO

KataGo [20] is an open-source reimplementation of AlphaZero introducing multiple modifications to the original algorithm improving its sample efficiency. In this section, we compare Go-Exploit to the search control procedures introduced in KataGo. Then, we argue that KataGo's other modifications are compatible with Go-Exploit and help it achieve even greater sample efficiency.

5.1 KataGo's Search Control Strategy

In the original KataGo paper and its subsequent release notes, key modifications to AlphaZero are highlighted. An additional change, which is not emphasized, is a search control strategy. KataGo's search control procedure involves self-play trajectory initialization and position branching. KataGo initializes self-play trajectories by sampling the first few moves from the policies $\mathbf{p}$ output by f_θ. KataGo occasionally branches trajectories by selecting a different action from the one that was originally played. KataGo also periodically branches from an early position by selecting the action with the greatest value from a set of randomly sampled actions.

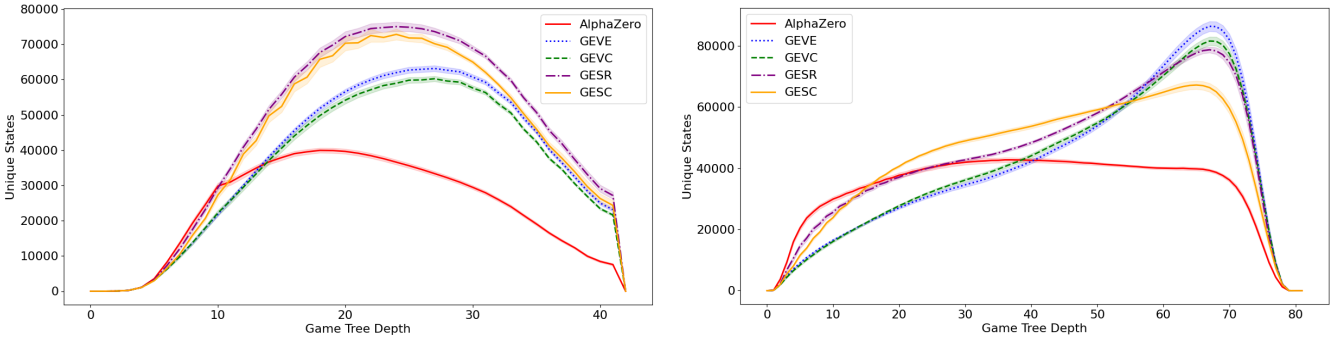
To compare the effectiveness of Go-Exploit's search control procedure to KataGo's, we ran OpenSpiel's AlphaZero implementation with KataGo's search control procedure in Connect Four. We ran AlphaZero with KataGo's trajectory initialization (AKTI), with KataGo's branching schemes (AKB), and then with both together (AKTIB). A hyperparameter sweep was conducted for each and then 30 additional validation runs were executed. The learning curves for AlphaZero, AlphaZero with KataGo's search control procedures, and GESC appear in Figure 2a. This figure illustrates that KataGo's search control strategy achieves a greater AUC than standard AlphaZero, however, it is inferior to the AUC of GESC. KataGo's search control strategy marginally improves AlphaZero's sample efficiency in Connect Four but not nearly as much as Go-Exploit.

5.2 Go-Exploit's Compatibility With KataGo

Excluding its trajectory initialization, KataGo's other modifications are compatible with Go-Exploit. KataGo's key modifications alter f_θ's architecture, the model updates, the feature representation, and search, which are orthogonal to Go-Exploit. To provide evidence that KataGo's other modifications are complementary with Go-Exploit, we incorporated three of KataGo's modifications into GESC to see if they could help it achieve even greater sample efficiency. The first modification was KataGo's branching scheme (GESCKB). While Go-Exploit's trajectory initialization resembles branching, it is not equivalent to KataGo's branching procedure. The second modification was "Playout Cap Randomization" (GESCKPCR), which randomly varies the number of search iterations performed. The third modification was "Forced Playouts and Policy Target Pruning" (GESCKFP), which forces visits to certain root actions during search. We ran GESC with each modification individually and then with all three (GESC3K) to see if an even greater sample efficiency could be achieved. For each variant, we performed a hyperparameter sweep and 30 additional validation runs. Their respective learning curves appear in Figure 2b. This figure illustrates that GESC maintains a similar AUC when combined with branching. However, GESC achieves an even greater AUC with Playout Cap Randomization and Forced Playouts + Policy Target Pruning. Furthermore, GESC achieves an even greater AUC when combined with all three. While not definitive, this supports our argument that KataGo's modifications to AlphaZero, other than its trajectory initialization, are complementary with Go-Exploit.

6 UNDERSTANDING GO-EXPLOIT

To understand why Go-Exploit learns more efficiently than standard AlphaZero, we collected statistics on the distribution of states



(a) Connect Four **(b) 9x9 Go**

Figure 3: The average number of unique nonterminal states visited by AlphaZero and Go-Exploit as a function of game tree depth. The shaded regions represent 95% confidence intervals.

visited during self-play and observed during search in the validation runs. In the following subsections, we appeal to these collected statistics and our guiding principles to try to establish why Go-Exploit outperforms AlphaZero in both Connect Four and 9x9 Go.

6.1 Greater Exploration of the State Space

We have argued that one of AlphaZero's limitations is that it does not effectively explore states deeper in the game tree. Since AlphaZero always begins its self-play trajectories from s_0, it often transitions into a terminal state before reaching and exploring states deep in the game tree. In addition, AlphaZero only samples actions over the first k moves of a self-play game, further limiting exploration deeper in the game tree. These suspicions are confirmed when comparing AlphaZero's distribution of unique visited states to Go-Exploit's. Figures 3a and 3b depict each algorithm's distribution of unique nonterminal states visited by game tree depth in Connect Four and 9x9 Go, respectively. In these plots, we observe that each variant of Go-Exploit visits a greater total number of unique nonterminal states than AlphaZero, particularly deeper in the game tree. At earlier game tree depths, AlphaZero visits more unique states than Go-Exploit. This is expected since AlphaZero begins each self-play trajectory from the initial state of the game whereas Go-Exploit begins its self-play trajectories from states throughout the game tree. In Figure 3a, the AlphaZero line gradually decreases at deeper game tree depths because in Connect Four, AlphaZero often transitions into a terminal state and then resets to s_0. In Figure 3b, the AlphaZero line remains fairly level over all game tree depths because there are fewer terminal states in 9x9 Go's game tree. Despite AlphaZero's increased exploration in 9x9 Go, Go-Exploit still outperforms AlphaZero by a greater margin in the larger game of 9x9 Go. It should also be noted that in Figure 3a, the slope of the AlphaZero line abruptly decreases at depth 10, which is the number of action sampling moves AlphaZero employs in Connect Four. This confirms the suspicion that only sampling actions over the first k moves of a self-play match hinders AlphaZero's exploration. These plots demonstrate that Go-Exploit is able to more effectively visit and train upon states throughout the state space than AlphaZero (guiding principle (a)) since it begins its self-play trajectories from states throughout the game tree and then subsequently takes exploratory actions from these varied starting points.

Among the variants of Go-Exploit, Go-Exploit Search States visits a greater total number of unique states than Go-Exploit Visited States. In fact, in Connect Four, this is the case over all game tree depths. This may partly explain why Go-Exploit Search States exhibits a greater sample efficiency than Go-Exploit Visited States in Connect Four. On the other hand, in 9x9 Go, Go-Exploit Search States visits a greater number of unique states at earlier game tree depths whereas Go-Exploit Visited States visits a greater number of unique states deeper in the game tree. Furthermore, the percentage difference in visited unique states between Go-Exploit Search States and Go-Exploit Visited States is greater in Connect Four than 9x9 Go. This may partly explain why the differences in sample efficiency between the variants of Go-Exploit are much smaller in 9x9 Go than in Connect Four.

To understand how the differences between AlphaZero and Go-Exploit's state visit distributions impact policy iteration, we compared their value losses over visited states and search states. To establish a fair comparison, we generated 500 self-play matches beginning from the initial state of the game using the final neural network checkpoints from the validation runs. For each visited state, we computed the squared error between the state's value estimate v_i and the outcome of the game z_i. For each state observed during search, a trajectory was played to completion without Dirichlet noise and action sampling so that the value loss could also be computed over search states. Table 2 shows each algorithm's average value loss over visited states and search states at checkpoint 600 in Connect Four and checkpoint 900 in 9x9 Go. We observe that each variant of Go-Exploit has a smaller value loss over visited states and search states than AlphaZero.[1] The fact that Go-Exploit has a smaller value loss over visited states in trajectories beginning from s_0 is particularly striking considering that AlphaZero only trains on trajectories beginning from s_0 and Go-Exploit does not. Go-Exploit's superior value loss over visited states and search states

[1]It may be surprising that smaller value losses were achieved over search states, however, this is due to there being no added stochasticity in these trajectories.

Table 2: Value losses over visited states and search states at checkpoint 600 in Connect Four, checkpoint 900 in 9x9 Go

	Connect Four		9x9 Go	
Algorithm	Visited	Search	Visited	Search
AlphaZero	0.196	0.161	0.293	0.244
GEVE	0.161	0.136	0.227	0.179
GEVC	0.164	0.116	0.223	0.166
GESR	0.170	0.127	0.214	0.165
GESC	0.151	0.108	0.241	0.172

Table 3: Hyperparameter values affecting the exploration-exploitation trade-off

	Connect Four					9x9 Go				
	c_{puct}	τ	k	α	ϵ	c_{puct}	τ	k	α	ϵ
AZ	1	1	10	1	0.25	1	1	2	0.03	0.1
GEVE	1	1	5	1	0.25	2	1	1	0.03	0.1
GEVC	1	1	10	1	0.1	2	1	1	0.03	0.1
GESR	1	1	2	1	0.25	1	1	2	0.03	0.1
GESC	1	1	10	1	0.25	2	1	1	0.03	0.1

illustrates that its value function can better predict match outcomes under its current policy and at a greater set of states than AlphaZero. We believe this can be attributed, in part, to Go-Exploit's more effective exploration of the game tree than AlphaZero. The fact that Go-Exploit trains a more accurate and more generalizable value function might be what enables its search to be a more effective policy improvement operator.

6.2 More Independent Value Targets

Another potential reason for Go-Exploit's smaller value losses over visited states and search states can be the fact that it produces and trains upon more independent value targets than AlphaZero. In AlphaZero, a new policy target is produced for each visited state whereas only a single independent value target is produced for each self-play trajectory (the outcome of the game). In addition to their scarcity, the value targets trained upon are noisy. The self-play match outcomes are affected by action sampling and Dirichlet noise, and therefore, may not reflect the true values of the visited states. Since Go-Exploit begins its self-play trajectories from states throughout the game tree, its self-play trajectories are shorter, on average, than AlphaZero's. In fact, in Connect Four, AlphaZero completes an average of 147.01 trajectories per learning step whereas each variant of Go-Exploit completes over 323, on average. Similarly, in 9x9 Go, AlphaZero completes an average of 74.83 trajectories per learning step whereas each variant of Go-Exploit completes over 147, on average. Since Go-Exploit completes more self-play trajectories per learning step than AlphaZero, its experience replay buffer contains more independent value targets, on average, than AlphaZero's. Consistently training on a greater number of independent value targets (guiding principle (d)) may enable Go-Exploit to train a more accurate value function, allowing search to be a more effective policy improvement operator.

6.3 Training Under More Exploitative Policies

In Figures 3a and 3b, we observed that Go-Exploit's search control strategy enables it to more effectively explore the game tree than AlphaZero. Since there is exploration inherent in the sampling of a start state, we hypothesized that Go-Exploit would require less stochasticity in its action selection than AlphaZero (guiding principle (c)). This would allow Go-Exploit to train under more exploitative policies, accelerating its policy iteration. Our hypothesis was mostly confirmed in our hyperparameter sweeps. Table 3 shows the best performing values for the hyperparameters affecting the exploration-exploitation trade-off for AlphaZero (AZ) and each variant of Go-Exploit in Connect Four and 9x9 Go. In Connect Four, GEVE, GEVC, and GESR are tuned more exploitatively than AlphaZero since they use fewer action sampling moves k or a smaller Dirichlet noise magnitude ϵ. GESC, however, was tuned identically to AlphaZero. In 9x9 Go, GEVE, GEVC, and GESC appear to be tuned more exploitatively than AlphaZero since they use fewer action sampling moves k, however, this conclusion is less certain since they employ larger c_{puct} constants, which affect the policies π_t returned by search. Since Go-Exploit is either tuned identically or more exploitatively than AlphaZero in both Connect Four and 9x9 Go, we can conclude that Go-Exploit relies less upon stochastic action selection than AlphaZero to explore the state space. This enables Go-Exploit to produce and train under policies that are inherently more exploitative, accelerating policy iteration.

7 CONCLUSION

In this paper, we have identified limitations in AlphaZero's training procedure and introduced a search control strategy, called Go-Exploit, that mitigates them. In sampling the start states of self-play trajectories from an archive of *states of interest*, Go-Exploit more effectively visits and revisits states throughout the state-space than standard AlphaZero. Furthermore, Go-Exploit produces and trains upon more independent value targets than AlphaZero. These factors enable Go-Exploit to learn a more accurate value function than AlphaZero, allowing search to be a more effective policy improvement operator. In addition, the exploration built into the "Go" step reduces Go-Exploit's need for exploratory actions, yielding self-play trajectories produced under stronger, more exploitative policies. These three factors accelerate Go-Exploit's policy iteration, which results in greater sample efficiency. We demonstrated Go-Exploit's ability to learn faster than AlphaZero in Connect Four and 9x9 Go, both measured in its performance against a common reference opponent and in head-to-head matches. We also showed that Go-Exploit utilizes a more effective search control strategy than KataGo and can benefit from KataGo's other improvements.

We have investigated two definitions of *states of interest* and three archive structures but have only sampled from the archive uniformly at random. Future work could investigate new ways of defining *states of interest*, new archive structures, and additional schemes for weighting and/or sampling states in the archive. Additional avenues for future work could include investigating how Go-Exploit can be used with a learned model [10] and in non-deterministic or imperfect information games [9].

REFERENCES

[1] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43.

[2] Rémi Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*. Springer, 72–83.

[3] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2021. First return, then explore. *Nature* 590, 7847 (2021), 580–586.

[4] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*. Springer, 282–293.

[5] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. 2019. OpenSpiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453* (2019).

[6] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8, 3 (1992), 293–321.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[8] Christopher D Rosin. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61, 3 (2011), 203–230.

[9] Martin Schmid, Matej Moravcik, Neil Burch, Rudolf Kadlec, Josh Davidson, Kevin Waugh, Nolan Bard, Finbarr Timbers, Marc Lanctot, Zach Holland, et al. 2021. Player of games. *arXiv preprint arXiv:2112.03178* (2021).

[10] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[11] 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.

[12] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[13] 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.

[14] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[15] Arash Tavakoli, Vitaly Levdik, Riashat Islam, Christopher M Smith, and Petar Kormushev. 2018. Exploring restart distributions. *arXiv preprint arXiv:1811.11298* (2018).

[16] Alexandre Trudeau and Michael Bowling. 2023. *Go-Exploit code*. https://github.com/alextrudeau/go_exploit

[17] Alexandre Trudeau and Michael Bowling. 2023. Targeted search control in AlphaZero for effective policy improvement. *arXiv preprint arXiv:2302.12359* (2023).

[18] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[19] Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. 2008. Monte-Carlo tree search solver. In *International Conference on Computers and Games*. Springer, 25–36.

[20] David J Wu. 2019. Accelerating self-play learning in Go. *arXiv preprint arXiv:1902.10565* (2019).