# Regret-Guided Search Control for Efficient Learning in AlphaZero

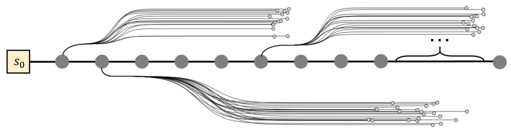
**Anonymous authors**
Paper under double-blind review

## Abstract

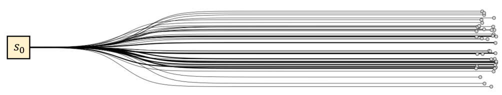
Reinforcement learning (RL) agents achieve remarkable performance but remain far less learning-efficient than humans. While RL agents require extensive self-play games to extract useful signals, humans often need only a few games, improving rapidly by repeatedly revisiting states where mistakes occurred. This idea, known as *search control*, aims to restart from valuable states rather than always from the initial state. In AlphaZero, prior work Go-Exploit applies this idea by sampling past states from self-play or search trees, but it treats all states equally, regardless of their learning potential. We propose *Regret-Guided Search Control* (RGSC), which extends AlphaZero with a regret network that learns to identify high-regret states, where the agent's evaluation diverges most from the actual outcome. These states are collected from both self-play trajectories and MCTS nodes, stored in a prioritized regret buffer, and reused as new starting positions. Across 9×9 Go, 10×10 Othello, and 11×11 Hex, RGSC outperforms AlphaZero and Go-Exploit by an average of 77 and 89 Elo, respectively. When training on a well-trained 9×9 Go model, RGSC further improves the win rate against KataGo from 69.3% to 78.2%, while both baselines show no improvement. These results demonstrate that RGSC provides an effective mechanism for search control, improving both efficiency and robustness of AlphaZero training.

## 1 Introduction

Reinforcement learning (RL) is the process of training an agent through interaction with the environment and optimizing its behavior based on rewards. The foundations of RL were originally inspired by human learning, where humans acquire new knowledge through trial-and-error experiences. However, despite this conceptual similarity, current RL approaches remain far less efficient than human learning (Tsividis et al., 2021; Iii & Sadigh, 2023). Consider the case of mastering the game of Go. An RL agent such as AlphaZero (Silver et al., 2017; 2018) requires millions of self-play games to reach superhuman performance. In contrast, professional human players can achieve comparable strength after far fewer games.



(a) Human learning.



(b) RL Agent (AlphaZero) learning.

Figure 1: Humans focus on correcting mistakes, whereas RL always starts from the initial state.

One key difference lies in how learning progresses. As illustrated in Figure 1, humans do not rely on playing massive numbers of games from the beginning. Instead, they repeatedly review the critical positions where mistakes occurred and refine their understanding until those weaknesses are corrected. AlphaZero, in contrast, always restarts from the empty board and updates all positions uniformly based on the obtained outcome, which substantially increases the number of episodes required to master a game.

To bridge this gap, recent studies have investigated restarting strategies to improve the efficiency of RL. This idea, originating from Sutton & Barto (2018), was formalized as the concept of *search control*, which refers to selecting critical starting states for the simulated experiences. Building on this principle, several works have been proposed for constructing restart distributions in RL, such as sampling from the past trajectories (Tavakoli et al., 2020), starting from states closer to the goal (Florensa et al., 2017), or leveraging expert demonstrations (Uchendu et al., 2023). Go-Exploit (Trudeau & Bowling, 2023) further extends this idea to the AlphaZero framework by maintaining a buffer of states from self-play trajectories or search nodes and uniformly sampling them as new starting positions. Collectively, these approaches demonstrate that choosing starting states, rather than always restarting from the initial state, can significantly accelerate RL learning. However, a key limitation of Go-Exploit is that it considers all states equally. In practice, not all states contribute equally to learning progress. Many states are already mastered, while only a small subset of states are actually critical for improvement. This phenomenon becomes exacerbated in the later stages of training, as the agent's understanding of the game improves and mistakes become increasingly rare. This motivates the need to identify and prioritize the most informative states for search control.

To address this challenge, we propose *Regret-Guided Search Control* (RGSC), a framework that extends AlphaZero by identifying and revisiting high-regret states. Specifically, RGSC leverages a regret network to detect states where the agent's evaluation diverges most from the game outcome. Since most states have near-zero regret, making direct learning of regret values challenging, we design a ranking-based objective that guides the network to distinguish the most informative states. These states are then stored in a prioritized regret buffer. By repeatedly restarting from these states, the agent can focus on correcting its most critical mistakes, thereby mimicking human learning and achieving more efficient training. Experimental results show that RGSC outperforms both AlphaZero and Go-Exploit across three board games, including 9x9 Go, 10x10 Othello, and 11x11 Hex, achieving an average improvement of 77 Elo over AlphaZero and 89 Elo over Go-Exploit. Furthermore, when continuing training from a strong, nearly converged model in 9x9 Go for 40 iterations, RGSC still improves the win rate from 69.3% to 78.2%, whereas both AlphaZero and Go-Exploit show no improvement. Moreover, additional analysis demonstrates that RGSC successfully identifies high-regret states and systematically reduces their regret during training. In summary, RGSC provides an effective mechanism for search control in AlphaZero. Our results highlight regret-guided search control as a promising direction for improving the efficiency and robustness of reinforcement learning.

## 2 Background

### 2.1 Search Control in Reinforcement Learning

The concept of *search control* (Sutton & Barto, 2018) was first introduced in the Dyna general framework (Sutton, 1991), which integrates real experience with model-generated simulated experience. In this setting, simulated experience is generated through search control, which determines the starting states and actions for rollouts, rather than always beginning from a fixed initial state. This allows planning to focus computation on states that provide more information to accelerate learning.

Several subsequent works have adopted the principle of search control by choosing different starting states during training. For example, Go-Explore (Ecoffet et al., 2021) addresses hard-exploration problems by maintaining a database of promising states, and periodically selecting from these states to discover high-reward trajectories. This approach allows systematic exploration of rarely visited regions and achieved state-of-the-art results in an extremely difficult environment, *Montezuma's Revenge*. (Florensa et al., 2017) propose another approach by selecting starting states near the goal and gradually moving them backward, thereby constructing a curriculum in reverse to facilitate learning in sparse reward environments. Jump-Start Reinforcement Learning (JSRL) (Uchendu et al., 2023) samples initial states from expert demonstration trajectories, allowing the agent to focus on meaningful states early in training, thereby improving sample efficiency. Tavakoli et al. (2020) provides a formal definition for exploring restart distributions by introducing a restart distribution $\rho(s)$ over states. By altering the distribution of the restart states, the learning objective is modified to

$$L(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \rho(s) \sum_{a \in \mathcal{A}} \pi(a \mid s) \left(q_\pi(s,a) - \hat{q}_\pi(s,a)\right)^2, \tag{1}$$

where $\mathcal{S}$ and $\mathcal{A}$ denote the state and action spaces, $\pi(a \mid s)$ is the policy, $q_\pi(s, a)$ is the true action-value function under $\pi$, and $\hat{q}_\pi(s, a)$ is its learned approximation. The restart distribution $\rho(s)$ specifies the probability of selecting state $s \in \mathcal{S}$ as a restart point. Two restart strategies are proposed: (a) uniform restart, which samples from recent experiences, and (b) prioritized restart, which ranks states according to their state-value temporal-difference (TD) error.

Moreover, search control is also widely applied at the level of task selection under curriculum-based environments. For example, several studies (Jiang et al., 2021; Dennis et al., 2020; Parker-Holder et al., 2023) adaptively sample training levels based on estimated regret, encouraging the agent to focus on levels with higher learning potential. While these methods operate at the level of task selection, our goal is to identify the most challenging states within the same game level.

### 2.2 Search Control in AlphaZero

AlphaZero (Silver et al., 2018) is a reinforcement learning algorithm that can master board games such as Chess, Shogi, and Go without requiring human knowledge. The training process alternates between two phases: a *self-play* phase and an *optimization* phase. During the self-play phase, the agent generates games against itself by combining Monte Carlo tree search (MCTS) Browne et al. (2012); Coulom (2007) with a two-head neural network, including a policy network that outputs a probability distribution over all possible actions and a value head that predicts the win rate of a given state. In the optimization phase, trajectories collected from self-play are stored in a replay buffer and sampled to update the neural network, training the policy head to predict the MCTS search distribution and the value head to predict the final game outcome. Although AlphaZero has demonstrated superhuman performance in board games, it requires extensive computation, especially in games with long trajectories, because every self-play game must start from the empty board. This issue is exacerbated in 19x19 Go, where a single game often exceeds 250 moves, making it necessary to spend enormous computational resources to generate self-play games (e.g., roughly 1.5 million TPU-hours as reported in (Silver et al., 2018)).

To alleviate this issue, several studies have incorporated search control into AlphaZero by adjusting self-play games to begin from particular intermediate states. For example, KataGo (Wu, 2020), one of the current strongest open-source Go programs, proposes selecting starting states either by randomly playing several moves with the policy network or by sampling and slightly modifying states from past self-play trajectories. Similar to Florensa et al. (2017), Björnsson (2023) proposes starting self-play from later stages of the game and gradually shifting the starting state toward the initial position. This approach accelerates the training process, particularly in the early phases. Recently, Trudeau & Bowling (2023) proposes Go-Exploit, which systematically investigates restart state methods within the AlphaZero algorithm. Go-Exploit maintains a buffer of states collected either from self-play trajectories (Go-Exploit Visited states Circular archive; GEVC) or from nodes within the MCTS (Go-Exploit Search states Circular archive; GESC). For each self-play game, the agent starts from the initial state with probability $\lambda$; otherwise, it uniformly samples a state from the buffer as the starting state. Go-Exploit achieves higher sample efficiency and stronger performance than AlphaZero in both Connect Four and 9x9 Go, with GEVC and GESC showing similar results. However, a key limitation of Go-Exploit is its uniform sampling. By treating all states equally, the method fails to align with the principle of restart distribution mentioned in Equation 1, which emphasizes prioritizing important states that provide better learning.

## 3 Regret-Guided Search Control

### 3.1 Regret Definition in Board Games

We propose *Regret-Guided Search Control* (RGSC), a framework that extends AlphaZero by identifying and prioritizing high-regret states as search control openings for self-play in board games, as shown in Figure 2. Unlike the original AlphaZero, as shown in Figure 1b, where self-play always starts from the empty board, RGSC guides self-play to begin from states with higher *regret*, where regret reflects positions that the current agent has not yet mastered. These states can appear either along the self-play trajectory or within the MCTS search tree. This allows the agent to focus on learning and exploring unfamiliar states with greater potential for improvement. Note that

Go-Exploit adopts a similar idea of restarting from previously collected states, but it samples them uniformly, which fails to capture the most informative states.

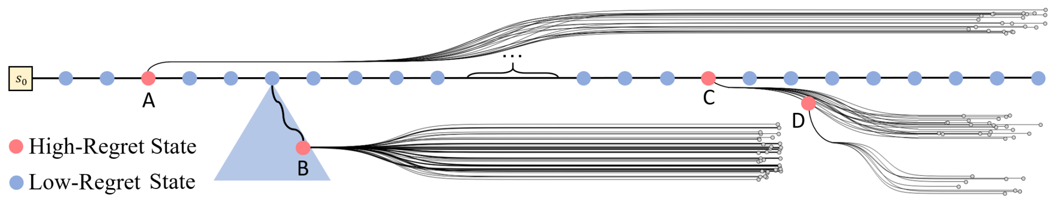


Figure 2: Overview of RGSC. The regret network selects high-regret states (red circles) from both self-play (`A`) or MTCS search node (`B`), which serve as restart points for further self-play. The newly generated trajectories can further branch out, e.g., state `D` originates from a restart at state `C`.

To formalize this idea, we define regret as a measure of the discrepancy between the agent's evaluation and the game outcome. Given a self-play trajectory with state $s_0$, $s_1$, . . ., $s_T$ and a game outcome $z$, the regret of state $s_t$ is

$$\mathcal{R}(s_t) = \frac{1}{T-t}\sum_{i=t}^{T}\left(V_{selected}(s_i) - z\right)^2, \tag{2}$$

where $V_{selected}(s_i)$ represents the MCTS value of the selected action at state $s_i$. Intuitively, $\mathcal{R}(s_t)$ measures the average discrepancy accumulated from $s_t$ to the terminal state $s_T$, capturing states whose mis-evaluation has long-term impact on the outcome.

Note that $\mathcal{R}(s_t)$ is calculated only after the game is finished. Moreover, the same state $s$ may have different regret values across trajectories, since the subsequent moves and outcome can vary. As training progresses and the agent's evaluations become more accurate, the regret of previously misjudged states gradually decreases. Conceptually, this resembles how human players repeatedly review their mistakes to improve.

## 3.2 Regret Network

Although the regret $\mathcal{R}(s_t)$ of states on a finished self-play trajectory can be directly calculated, many internal states in the MCTS search tree are not part of the actual trajectory and thus have no regret values. Nevertheless, these states may still include critical states that the agent has not yet mastered. Leveraging such states allows the agent to obtain more diverse restart states beyond the limited set of self-play trajectories.

A naive approach is to train a *regret value network* that, given a state, directly predicts its regret value, similar to settings in learning-to-learn problems (Wang et al., 2017; Chu et al., 2024; Gupta et al., 2020). However, predicting regret value for arbitrary states in AlphaZero training is highly challenging. First, the distribution is extremely imbalanced: most states have near-zero regret, while high-regret states occur only rarely. Second, the learning target is non-stationary: high-regret states are selected for restarts, and once revisited, their regret typically decreases quickly as the agent corrects its mistakes. As a result, predicting regret becomes extremely difficult for this naive approach.

To tackle this challenge, we propose to learn regret with a ranking-based objective. The key idea is that instead of predicting precise regret values, which are imbalanced and non-stationary targets, we only need to identify which states have higher regret among all collected states. This relaxation guides the model to focus on the most informative states, ensuring that they are included in the prioritized buffer for restarting self-play.

Specifically, we incorporate a *regret ranking network* into the AlphaZero network. Given a state $s$, the regret ranking network outputs an unnormalized score, $\gamma_s$, where $\gamma_s$ represents the ranking score of state $s$. Note that $\gamma$ is a relative ranking score rather than the true regret value, with higher scores corresponding to states with higher regrets. For a set of candidate states $\mathcal{S}$, the restart distribution $\rho(s \mid \mathcal{S})$ is derived as follows:

$$\rho(s \mid \mathcal{S}) = \frac{\exp(\gamma_s)}{\sum_{s' \in \mathcal{S}} \exp(\gamma_{s'})}. \tag{3}$$

Following Equation 1, the ranking objective is to maximize

$$\mathcal{J}_{\text{rank}} = \sum_{s \in \mathcal{S}} \rho(s \mid \mathcal{S}) \mathcal{R}(s), \tag{4}$$

which encourages the model to assign high probability to the highest-regret states, as these are the most critical for maximizing $\mathcal{J}_{\text{rank}}$ and for restarting self-play.

To better optimize the regret ranking network, we apply an exponential transformation to the regret values, which preserves the ranking order. Then, the network is optimized by using a surrogate objective

$$\tilde{\mathcal{J}}_{\text{rank}} = \sum_{s \in \mathcal{S}} \rho(s \mid \mathcal{S}) \exp\big(\mathcal{R}(s)\big), \tag{5}$$

and the corresponding loss function is defined as

$$\mathcal{L}_{\text{rank}} = -\log \tilde{\mathcal{J}}_{\text{rank}} = -\log \sum_{s \in \mathcal{S}} \rho(s \mid \mathcal{S}) \exp\big(\mathcal{R}(s)\big) \tag{6}$$

$$= -\log \sum_{s \in \mathcal{S}} \Big( \exp\big(\log \text{softmax}(\gamma_s) + \mathcal{R}(s)\big)\Big). \tag{7}$$

The derived loss can be interpreted as adding regret as an additive bias to the log-softmax scores, providing a smooth approximation to selecting the highest-regret states. We provide a detailed derivation in the Appendix C.

Although the regret ranking network can differentiate states with higher regrets, its ranking score is not bounded within the true regret value range. Therefore, to provide a quantitative measurement of regret for the selected states, our regret network consists of both a regret value network and a regret ranking network. The regret ranking network identifies high-regret states, while the regret value network estimates their actual regret value.

## 3.3 Prioritized Regret Buffer for Search Control

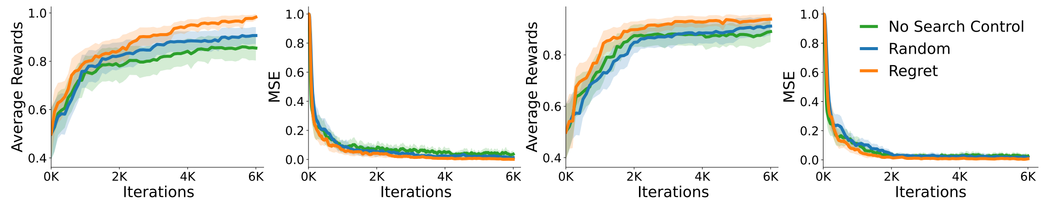
We describe the *prioritized regret buffer* (PRB), which utilizes the regret network to allow search control during AlphaZero training. For each self-play game, we first apply the regret ranking network to evaluate all states that appear both in the self-play trajectory and in the MCTS search trees. The state with the highest ranking score is then selected. If the selected state $s$ appears in the self-play trajectory, we calculate its regret value $\mathcal{R}(s)$ using Equation 2; if it appears only in the search tree, its regret value is estimated by the regret value network. The PRB maintains only a fixed capacity of $K$ states. If the PRB is not yet full, the selected state $s$ is added directly. Otherwise, it is added only if its regret is higher than that of the lowest-regret state currently in the PRB. This ensures that the PRB consistently stores a set of high-regret states for restarting.

For each self-play game, search control guides the choice of restarting state, starting from the empty board with probability $1 - \lambda$, and from a state sampled from PRB with probability $\lambda$. We adopt a softmax distribution over all states in PRB when sampling to ensure high-regret states are prioritized. The probability of selecting a state $s_i$ in PRB is defined as $P(s_i) = \mathcal{R}(s_i)^{1/\tau} / \sum_j \mathcal{R}(s_j)^{1/\tau}$, where $\tau$ is the sampling temperature.

For restarting games from states in PRB, we update their regret values $\mathcal{R}^{\text{new}}(s_i)$ after replaying each game using an exponential moving average (EMA):

$$\mathcal{R}^{\text{new}}(s_i) \leftarrow (1 - \alpha) \times \mathcal{R}^{\text{old}}(s_i) + \alpha \times \mathcal{R}(s_i), \tag{8}$$

where $\mathcal{R}^{\text{old}}(s_i)$ is the previous regret value stored in the buffer, $\mathcal{R}(s_i)$ is the regret calculated from the newly finished self-play game, and $\alpha$ is the EMA coefficient. This prevents regret values from decreasing abruptly and ensures that once the agent has consistently mastered this state, its regret will gradually decay, thereby reducing the probability of the state being sampled from the buffer. In summary, this design mirrors how humans repeatedly review mistakes until they are fully understood. We have also provided a detailed algorithm for RGSC in the Appendix B.



(a) $n = 5$, Rewards (b) $n = 5$, MSE (c) $n = 6$, Rewards (d) $n = 6$, MSE

Figure 3: A toy example on $n$-level binary tree. (a) and (c) show the average rewards during the training, while (b) and (d) show the mean squared error (MSE) of the optimal Q-values during the training. The shaded area is a 95% confidence interval for the mean.

## 4 Experiments

### 4.1 Toy Example

We first investigate search control in a toy environment, an $n$-level sparse-reward binary tree, where each leaf node is assigned an expected reward value $p \in [0, 1]$. The agent starts from the root and selects nodes until reaching a leaf. Upon reaching a leaf node, it receives a stochastic binary reward: 1 with probability $p$ and 0 with probability $1 - p$. Among all leaf nodes, exactly one node is assigned to $p = 1$; thus, the objective in this environment is to discover the unique path that always guarantees a reward 1. Next, we train Q-learning on this environment with three search control methods: (a) No search control, always starting from the root; (b) Random, uniformly sampling from visited nodes; and (c) Regret, sampling nodes in proportion to their regret. For the regret, we simply use $|\hat{Q}(s) - Q(s,a)|$, where $\hat{Q}(s)$ is the empirical maximum expected value estimated from all child nodes, and $Q(s,a)$ is the current Q-value of state $s$ with action $a$. Figure 3 shows the results for the 5- and 6-level binary trees. The Regret method achieves higher average rewards than both Random and No search control. These results demonstrate the importance of prioritizing states with high learning potential and show the effectiveness of the regret-guided search control. Detailed settings of the toy environment are provided in Appendix D.

### 4.2 RGSC in Board Games

We compare RGSC against two baseline methods: (a) AlphaZero, which is trained without search control, and (b) Go-Exploit with its GEVC variant described in subsection 2.2, across three board games, including 9x9 Go, 10x10 Othello, and 11x11 Hex. All methods use a 3-block residual network (He et al., 2016) and 200 MCTS simulations per move during self-play. Training runs for 300 iterations, with 160,000 states collected per iteration in 9x9 Go (due to its higher complexity) and 120,000 states in the other two games. We fix the number of training states rather than the number of self-play games per iteration to ensure fairness, since AlphaZero without search control requires more computation to generate a self-play game. Detailed settings are provided in Appendix A. In summary, each training requires approximately 150 NVIDIA RTX A6000 GPU hours.

Figure 4 shows the Elo curves for each method across the three board games. For each game, all models are evaluated against the 150-iteration AlphaZero model, whose Elo rating is fixed at 1000 as the reference point. When comparing the final checkpoint across all methods, RGSC consistently outperforms both baselines in all three games. In 9x9 Go, RGSC surpasses AlphaZero and Go-Exploit by 76 and 96 Elo points, respectively; in 10x10 Othello, the improvements are 70 and 50 Elo points; and in 11x11 Hex, the differences are 84 and 122 Elo points.

Interestingly, we observe that although Go-Exploit achieves a higher Elo than AlphaZero in the early stages of training, its advantage diminishes as training converges. This phenomenon is also evident in the original Go-Exploit experiments. We hypothesize that, during early training, many states exhibit high regret since the model has much to learn, making it easy to select informative states with uniform sampling. As training progresses, however, the number of unfamiliar states decreases, thus uniform sampling becomes less effective. In contrast, RGSC continues to prioritize

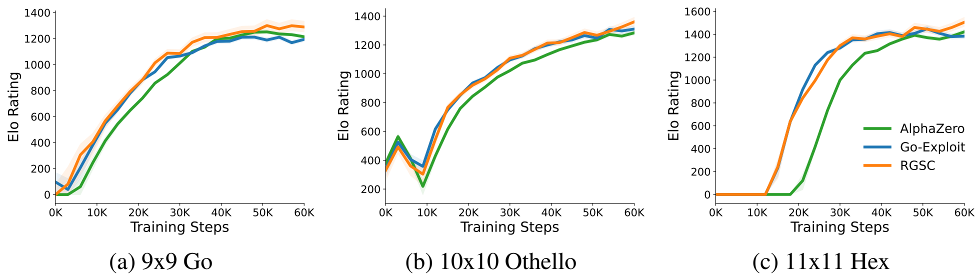


Figure 4: Playing performance of AlphaZero, Go-Exploit, and RGSC on three different board games. The shaded area is a 95% confidence interval for the mean.

the remaining high-regret states, allowing it to focus on difficult states and maintain its advantage in the later stages of training.

Table 1: Win rate against established open-source programs on three board games.

| | AlphaZero | Go-Exploit | RGSC |
|---|---|---|---|
| 9x9 Go | 45.5%±1.5% | 49.5%±2.0% | **53.6%±2.4%** |
| 10x10 Othello | 51.7%±2.5% | 52.9%±3.3% | **57.8%±3.2%** |
| 11x11 Hex | 83.6%±1.6% | 89.2%±1.8% | **91.1%±2.0%** |

Furthermore, to assess whether the observed improvements remain consistent, we evaluate the final checkpoint by playing against established open-source programs across all three games. We select KataGo (Wu, 2020), one of the strongest open-source Go programs, for 9x9 Go; an alpha-beta search implementation in Ludii (Piette et al., 2020) for 10x10 Othello; and MoHex (Huang et al., 2014), a MCTS-based Hex program that won Computer Olympiad championships, for 11x11 Hex. Detailed settings for each program are listed in subsection A.1. Table 1 summarizes the win rate against these opponents. The results are consistent with the findings in Figure 4, showing that RGSC consistently outperforms both AlphaZero and Go-Exploit. Overall, these experiments demonstrate that RGSC offers a more efficient search control mechanism, resulting in higher training efficiency and stronger playing performance.

### 4.3 RGSC on Well-Trained Models

Building on the findings in subsection 4.2, where Go-Exploit showed early improvement but failed to yield significant progress as training converged, we now investigate whether RGSC can provide further improvements when starting from an already well-trained model. It is worth noting that mistakes become increasingly rare in such models, making it particularly challenging for the agent to identify and learn from the remaining high-regret states.

To investigate this, we select a large 15-block baseline model trained with the AlphaZero algorithm on 9x9 Go, which required approximately 1,060 NVIDIA RTX A6000 GPU hours and already achieves a strong playing strength. When compared against a KataGo model of the same block size, the baseline achieves a win rate of 69.3%. Similarly, we adopt AlphaZero, Go-Exploit, and RGSC using the same baseline model as the initial weight to ensure a fair comparison. For RGSC, since the original baseline model does not include the regret network, we add the regret network and generate additional self-play games to train it, while keeping the policy and value networks frozen. All three methods are then continued for 40 iterations under identical settings, requiring approximately 100 NVIDIA RTX A6000 GPU hours. Additional training details are provided in subsection A.2.

Figure 5 shows the results of continued training. Similar to the findings in subsection 4.2, RGSC achieves the strongest performance, while Go-Exploit performs even worse than AlphaZero. At the final checkpoint, RGSC surpasses AlphaZero by 42 Elo points and Go-Exploit by 87 points. Furthermore, we evaluate the final checkpoint models against KataGo. The original baseline model achieves a win rate of $69.3\% \pm 2.6\%$. After continued training, AlphaZero reaches $70.2\% \pm 2.7\%$

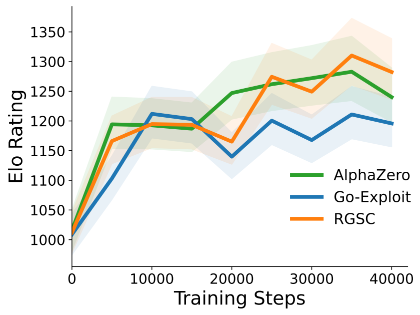


Figure 5: Playing performance of AlphaZero, Go-Exploit, and RGSC when continued from well-trained models. The shaded area is a 95% confidence interval for the mean.

and Go-Exploit $69.2\% \pm 2.7\%$, showing no meaningful improvement. In contrast, RGSC achieves a substantially higher win rate of $78.2\% \pm 2.5\%$, significantly outperforming both baselines. To conclude, these results indicate that RGSC can effectively track remaining self-mistake states even in a well-trained model, thereby achieving further performance improvements.

### 4.4 Comparison between Ranking and Regret in RGSC

Both the regret value network and regret ranking network can be used to identify candidate states for restarting, but their effectiveness may differ. The regret value network directly estimates regret values, whereas the regret ranking network emphasizes relative ordering. In this subsection, we analyze their differences in identifying high-regret states and examine the impact on search control.

We first train an RGSC variant that relies only on the regret value network for both state selection and regret initialization. Figure 6 presents the training results, showing that the regret ranking network outperforms the regret value network across all three games.

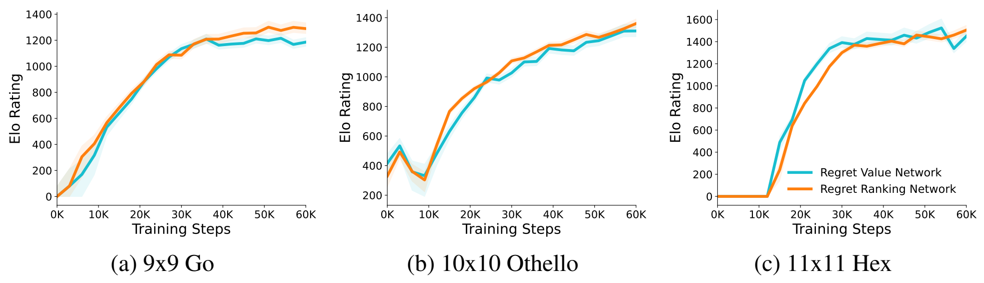


Figure 6: Playing performance of RGSC using regret value network and regret ranking network. The shaded area is a 95% confidence interval for the mean.

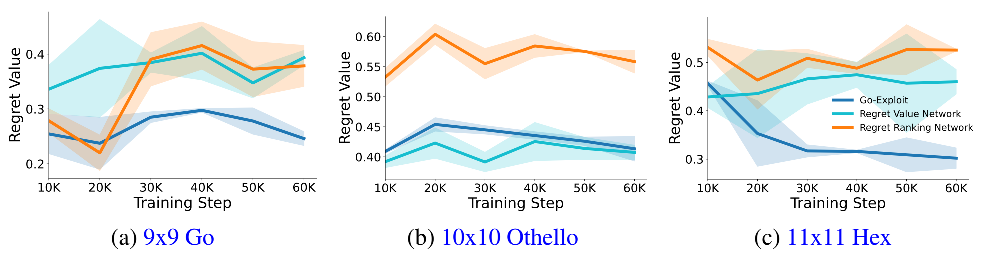
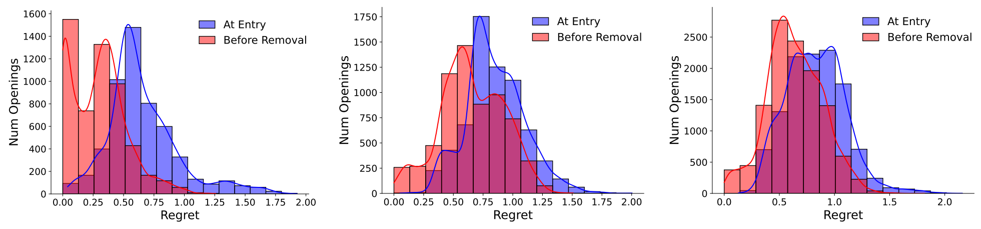


Figure 7: The regret values for nodes selected by Go-Exploit, regret value and ranking network. The shaded area is a 95% confidence interval for the mean.

Next, we examine whether the states selected by the two networks indeed correspond to high-regret states. We collect all states from self-play trajectories at each training step and evaluate them by both networks. Since these states come directly from trajectories, their true regret values can be computed according to Equation 2. We then rank the states separately with the regret value and ranking networks, and select the top 2,000 states predicted by each. The true regret values of these selected states are averaged to obtain the average regret of the states identified by each network. Figure 7 shows the average regret of the states selected by each method during training. Overall, the regret ranking network consistently selects states with higher true regret than the regret value network, especially in 10x10 Othello. For convenience, we also include the Go-Exploit approach as a baseline, where the average regret is obtained by randomly sampling 2,000 states. As expected, the uniform sampling approach results in the lowest average regret among all methods. Moreover, the average regret of Go-Exploit decreases during training, especially in Hex, corroborating our hypothesis that Go-Exploit becomes less effective in later stages. In contrast, the regret ranking network maintains a substantially higher average regret even at late training steps, indicating its ability to continually identify difficult states. In summary, these results demonstrate that the ranking objective improves the quality of selected states by prioritizing those with greater learning potential.

### 4.5 Regret Change in Prioritized Regret Buffer

This subsection examines whether the high-regret states in the PRB gradually decrease during training, i.e., whether the model can actually correct its mistakes by repeatedly revisiting those states. Specifically, we record the regret of each state when it first enters the PRB and compare it with its final regret before removal. Figure 8 shows the regret distributions at these two points across all three games. Generally, the distributions consistently shift toward the left (lower regret values) as training progresses. This confirms that states initially associated with high regret are eventually corrected through repeated replay, resulting in reduced regret over time. In addition, by comparing the average regret values, we observe that the average regret decreases significantly across all games: from 0.655 to 0.296 in 9x9 Go, from 0.828 to 0.638 in 10x10 Othello, and from 0.848 to 0.657 in 11x11 Hex. These results demonstrate that RGSC continuously identifies states where the agent struggles, allows them to be self-corrected through repeated revisits until mastered, and then refreshes the buffer with new challenging states. More analyses on high-regret states and the game length of restart states in PRB are provided in Appendix F and Appendix G, respectively.



(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 8: Regret distributions of states in the prioritized regret buffer at entry and before removal.

## 5 Discussion

This paper proposes *Regret-Guided Search Control* (RGSC), an extension of AlphaZero that identifies high-regret states. By integrating a regret network and a prioritized regret buffer, RGSC allows the agent to repeatedly focus on correcting its most critical mistakes, mimicking how humans learn. Experimental results show that RGSC outperforms AlphaZero and Go-Exploit, achieving an average of 77 and 89 Elo, respectively. Furthermore, RGSC successfully improves the win rate against KataGo on a well-trained 9×9 Go model from 69.3% to 78.2%, while both baselines show no improvement. These results demonstrate the learning efficiency and robustness of RGSC.

Although our study focuses on board games, AlphaZero and its successor MuZero (Schrittwieser et al., 2020) are general frameworks, suggesting that RGSC could be applied to more applications beyond games. Specifically, our preliminary experiments (shown in Appendix H) applying RGSC to MuZero on one of the Atari games, *Pac-Man*, show that under the same training budget, RGSC-MuZero reaches 5166 points, compared to 3704 for MuZero. This demonstrates the potential of RGSC to improve learning efficiency beyond board games. Future work can extend RGSC to more domains, such as stochastic environments (Antonoglou et al., 2021) and continuous control tasks (Hubert et al., 2021). The regret network also provides interpretability by revealing specific weaknesses in the agent's learning. Furthermore, the ability of RGSC to improve even on a well-trained model indicates its scalability to more complex environments such as 19×19 Go or large-scale sequential decision-making problems. We believe RGSC is a promising direction for advancing RL field.

## Ethics Statement

We do not foresee any ethical issues in this work.

## Reproducibility Statement

To reproduce this work, we provided the details of the algorithm in Appendix B, and hyperparameters in Appendix A. The source code, trained models used in the experiment, along with a README file, will be released to ensure reproducibility once this paper is accepted.

## The Use of Large Language Models (LLMs)

Large language models (LLMs) were used only for grammar correction and proofreading in the preparation of this paper.

## References

Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K. Hubert, and David Silver. Planning in Stochastic Environments with a Learned Model. In *International Conference on Learning Representations*, October 2021.

Jónsson Björnsson, Y. Expediting Self-Play Learning in AlphaZero-Style Game-Playing Agents. In *ECAI 2023*, pp. 263–270. IOS Press, 2023.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

Zhendong Chu, Renqin Cai, and Hongning Wang. Meta-Reinforcement Learning via Exploratory Task Clustering. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(10):11633–11641, March 2024.

Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, Lecture Notes in Computer Science, pp. 72–83, Berlin, Heidelberg, 2007. Springer.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *International Conference on Learning Representations*, April 2022.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design. In *Advances in Neural Information Processing Systems*, volume 33, pp. 13049–13061. Curran Associates, Inc., 2020.

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

Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse Curriculum Generation for Reinforcement Learning. In *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 482–495. PMLR, October 2017.

Abhishek Gupta, Benjamin Eysenbach, Chelsea Finn, and Sergey Levine. Unsupervised Meta-Learning for Reinforcement Learning, April 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Shih-Chieh Huang, Broderick Arneson, Ryan B. Hayward, Martin Müller, and Jakub Pawlewicz. MoHex 2.0: A Pattern-Based MCTS Hex Player. In *Computers and Games*, volume 8427, pp. 60–71. Springer International Publishing, Cham, 2014.

Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and Planning in Complex Action Spaces. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 4476–4486. PMLR, July 2021.

Donald Joseph Hejna Iii and Dorsa Sadigh. Few-Shot Preference Learning for Human-in-the-Loop RL. In *Proceedings of The 6th Conference on Robot Learning*, pp. 2014–2025. PMLR, March 2023.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized Level Replay. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 4940–4950. PMLR, July 2021.

Xiaozhen Niu, Akihiro Kishimoto, and Martin Müller. Recognizing Seki in Computer Go. In *Advances in Computer Games*, Lecture Notes in Computer Science, pp. 88–103, Berlin, Heidelberg, 2006. Springer.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving Curricula with Regret-Based Environment Design, September 2023.

Éric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H. M. Winands, and Cameron Browne. Ludii – The Ludemic General Game System, February 2020.

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

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bull.*, 2(4):160–163, 1991.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. MIT Press, Cambridge, MA, USA, 2 edition, November 2018.

Arash Tavakoli, Vitaly Levdik, Riashat Islam, Christopher M. Smith, and Petar Kormushev. Exploring Restart Distributions, August 2020.

Alexandre Trudeau and Michael Bowling. Targeted Search Control in AlphaZero for Effective Policy Improvement. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '23, pp. 842–850, Richland, SC, May 2023. International Foundation for Autonomous Agents and Multiagent Systems.

Pedro A. Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. Human-Level Reinforcement Learning through Theory-Based Modeling, Exploration, and Planning, July 2021.

Ikechukwu Uchendu, Ted Xiao, Yao Lu, Banghua Zhu, Mengyuan Yan, Joséphine Simon, Matthew Bennice, Chuyuan Fu, Cong Ma, Jiantao Jiao, Sergey Levine, and Karol Hausman. Jump-Start Reinforcement Learning. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 34556–34583. PMLR, July 2023.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn, January 2017.

David J. Wu. Accelerating Self-Play Learning in Go. In *Proceedings of the AAAI Workshop on Reinforcement Learning in Games*, November 2020.

Ti-Rong Wu, Hung Guei, Pei-Chiun Peng, Po-Wei Huang, Ting Han Wei, Chung-Chin Shih, and Yun-Jui Tsai. MiniZero: Comparative Analysis of AlphaZero and MuZero on Go, Othello, and Atari Games. *IEEE Transactions on Games*, 17(1):125–137, March 2025.

## A Training Details

### A.1 Training RGSC in Board Games

In this section, we describe the details for training models used in the experiments. The trainings in section 4 are conducted on a machine with two Intel Xeon Silver 4516Y+ CPUs and four NVIDIA RTX A6000 GPUs. For the implementation of RGSC, and the two baseline networks, AlphaZero and Go-Exploit, each network employs the Gumbel AlphaZero algorithm (Danihelka et al., 2022). Three methods are implemented based on an open-sourced AlphaZero framework (Wu et al., 2025). During training, 4 self-play workers and 1 optimization worker are used. For RGSC, each worker maintains its own prioritized regret buffer (PRB).

In the experiments of subsection 4.2, we use hyperparameters shown in Table 2 to train all three methods. For the training of RGSC, the additional hyperparameters are set as follows: the sampling probability $\lambda$ is 0.5, the buffer sampling temperature $\tau$ is 0.1, the buffer size $\kappa$ is 100, and the EMA coefficient $\alpha$ is 0.5.

Table 2: Hyperparameters for training in subsection 4.2.

| Parameter | Go | Hex | Othello |
|---|---|---|---|
| Board size | 9 | 11 | 10 |
| Optimizer | | SGD | |
| Optimizer: learning rate | | 0.02 | |
| Optimizer: momentum | | 0.9 | |
| Optimizer: weight decay | | 0.0001 | |
| MCTS simulation | | 200 | |
| Softmax temperature | | 1 | |
| Iteration | | 300 | |
| Self-Play states per iteration | 160,000 | 120,000 | 120,000 |
| Optimizations per iteration | | 200 | |
| Batch size | | 1024 | |
| # Residual blocks | | 3 | |
| # Residual blocks filters | | 256 | |
| Replay buffer size | | 20 | |
| Dirichlet noise ratio | | 0.25 | |

In the experiments of subsection 4.3, we train all three methods from already well-trained models with the hyperparameters shown in subsection 4.2. In this training setup, the buffer size is set to 500. To warm up the PRB in RGSC, we generate 1,000 self-play games with a buffer rate $\lambda = 0.5$, excluding these warm-up games from the training data.

Regarding the computational cost of RGSC, although RGSC adds two additional heads (regret value and ranking heads), they share the same backbone as the policy/value network, so the additional computation is minimal, especially in larger models. We measured both the neural network inference time and the per-iteration wall-clock time on Go. Specifically, for the 3-block model (used in subsection 4.2), RGSC is 1.35x slower in inference and 1.25x slower per iteration compared to AlphaZero/Go-Exploit. However, for the 15-block model (used in subsection 4.3), RGSC is only 1.02x slower in inference, and the per-iteration time is nearly identical ($\sim$1.00x) to AlphaZero/Go-Exploit. In realistic settings (e.g., AlphaZero used 20 blocks and KataGo used 20-40 blocks), this overhead becomes negligible. Therefore, RGSC adds almost no extra cost while improving training efficiency.

### A.2 Evaluation

In the evaluation of each method, we repeat each experiment twice per game and average the results to reduce stochastic variance. The number of MCTS simulations is set to 400, with the action softmax temperature set to 1.

Table 3: Hyperparameters for the training in subsection 4.3.

| Parameter | Go |
|---|---|
| Board size | 9 |
| Optimizer | SGD |
| Optimizer: learning rate | 0.001 |
| Optimizer: momentum | 0.9 |
| Optimizer: weight decay | 0.0001 |
| MCTS simulation | 400 |
| Softmax temperature | 1 |
| Iteration | 40 |
| Self-Play states per iteration | 160,000 |
| Optimizations per iteration | 1000 |
| Batch size | 256 |
| # Residual blocks | 15 |
| # Residual blocks filters | 128 |
| Replay buffer size | 20 |
| Dirichlet noise ratio | 0.25 |

### A.2.1 Evaluation Against the AlphaZero

For the experiments in Figure 4 of subsection 4.2, and Figure 6 of subsection 4.3, all methods are evaluated against the AlphaZero baseline every 15 iterations, with 200 games played per evaluation. In the following paragraph, we will describe the setting of evaluation on Table 1.

### A.2.2 Evaluation Against Open Source Programs

In subsection 4.2 and subsection 4.3 of main text, we evaluated all the methods against open-sourced programs across all three games. The setup of evaluations for each game is outlined below:

**KataGo.** For the evaluation of 9x9 Go in Table 1 of subsection 4.2, we selected KataGo (Wu, 2020) models from ID 1 to 4 as baselines. For each baseline, we conduct 200 evaluation games with 100 games as Black and 100 games as White, with the simulation count for KataGo fixed at 400. The four selected KataGo models are listed in Table 4.

For the evaluation against KataGo in subsection 4.3, we pick KataGo with ID 5 in Table 4. In this experiment, actions are selected without applying softmax, and 1,200 evaluation games are conducted for each method.

Table 4: The versions of the selected KataGo models for 9x9 Go.

| **ID** | **Version** | **# blocks** | **Elo ratings** |
|---|---|---|---|
| 1 | `kata1-b6c96-s152505856-d23152636` | 6 | $9833.3 \pm 16.1$ |
| 2 | `kata1-b6c96-s165180416-d25130434` | 6 | $9900.6 \pm 16.2$ |
| 3 | `kata1-b6c96-s175395328-d26788732` | 6 | $9958.6 \pm 16.9$ |
| 4 | `kata1-b10c128-s41138688-d27396855` | 10 | $10138.6 \pm 18.3$ |
| 5 | `kata1-b15c192-s86740736-d72259836` | 15 | $11180.1 \pm 16.1$ |

**Alpha-Beta Algorithm in Ludii.** For the evaluation of 10x10 Othello in Table 1 of subsection 4.2, we use the Alpha-Beta algorithm from Ludii (Piette et al., 2020) with search levels set to 2, 3, and 4 as our baselines. For each method in Table 1, a total 300 games are played, with 150 games as Black and 150 games as White.

**MoHex.** For the evaluation of 11x11 Hex in Table 1 of subsection 4.2, all methods fight against MoHex (Huang et al., 2014). For the MoHex setup, the maximum thinking time is set to 1 second, without using a cache book. Additionally, we use two MoHex baselines with the following search settings: a search width of 15 with a maximum search depth of 5, and a search width of 25 with a maximum search depth of 8.

## B RGSC Algorithm

In this section, we describe the details of the RGSC algorithm in Algorithm 1. It contains value network $\mathcal{R}$, regret ranking network $\rho$ within the prioritized regret buffer (PRB). Lines 5–12 specify the procedure of buffer sampling, while Lines 13–30 outline the self-play process, during which all nodes in the search tree are evaluated by the regret network and describe how an opening $s$ is selected. Line 31–40 describe how an opening $s$ is updated and how it inserted into buffer. In this procedure, search tree nodes are inserted into the PRB, enabling us to exploit the search tree while exploring previously unseen states with potentially high regret.

**Algorithm 1** RGSC Algorithm

```
Require: Buffer $\beta$, Buffer size $N$, Buffer rate $\lambda$, Buffer Sample Rule $\Psi(\beta)$, Regret ranking network
    $\rho$, Regret value network $R$
 1: Initialize Buffer $\beta$
 2: while Self-Play do
 3:     Reset the environment
 4:     Initial Buffer candidate $s'$ with ranking $R'$ and regret $r'$
 5:     Sample random number $p \in (0, 1)$
 6:     if $p < \lambda$ then
 7:         Sample a opening $s$ from $\beta$ with $\Psi(\beta)$
 8:         Set $s$ as the starting state of self-play
 9:         Update sampled times of opening $s$
10:     else
11:         Set empty state $s_0$ as the starting state of self-play
12:     end if
13:     while The environment is not terminal do
14:         Perform MCTS and selected an action
15:         Obtain regret ranking value $\rho(s)$ and predicted regret $r \leftarrow \mathcal{R}(s)$ for every search node $s$
16:         if A search node $s$ with regret ranking $\rho(s) > R'$ then
17:             update $r' \leftarrow r$
18:             update $s' \leftarrow s$
19:         end if
20:     end while
21:     Use the final return to compute the regret $r$ of every node $s$ on the trajectory
22:     if Using regret ranking network then
23:         Obtain regret ranking value $\rho(s)$ for every node $s$ on the self-play trajectory
24:         if A trajectory node $s$ with Ranking $\rho(s) > R'$ then
25:             update $r' \leftarrow r$
26:             update $s' \leftarrow s$
27:         end if
28:     end if
29:     if $p < \lambda$ then
30:         Update the regret of opening $s$ using the EMA rule in Equation 8
31:     else
32:         Store candidate $s'$ into the buffer with regret $s'$
33:     end if
34: end while
```

## C Detail of Regret Ranking Head Training Objective

A step-by-step derivation of Equation 6 is provided below.

$$\mathcal{L}_{\text{rank}} = -\log \sum_{s \in \mathcal{S}} \rho(s \mid \mathcal{S}) \exp\left(\mathcal{R}(s)\right) \tag{9}$$

$$= -\log \left( \sum_{s \in \mathcal{S}} \frac{\exp(\gamma_s)}{\sum_{s' \in \mathcal{S}} \exp(\gamma_{s'})} \exp\left(\mathcal{R}(s)\right) \right) \tag{10}$$

$$= -\log \sum_{s \in \mathcal{S}} \left( \exp\left( \log \left( \frac{\exp(\gamma_s)}{\sum_{s' \in \mathcal{S}} \exp(\gamma_{s'})} \right) \right) \exp\left(\mathcal{R}(s)\right) \right) \tag{11}$$

$$= -\log \sum_{s \in \mathcal{S}} \left( \exp\left( \log \left( \frac{\exp(\gamma_s)}{\sum_{s' \in \mathcal{S}} \exp(\gamma_{s'})} \right) + \mathcal{R}(s) \right) \right) \tag{12}$$

$$= -\log \sum_{s \in \mathcal{S}} \left( \exp\left( \log\left(\text{softmax}(\gamma_s)\right) + \mathcal{R}(s) \right) \right) \tag{13}$$

## D Detail for toy model experiment

In the experiments in subsection 4.1, we implemented a simple Q-learning example on sparse-reward binary trees with five and six levels. In Q-learning, the learning rate is set to 0.1, $\epsilon$ is set to 0.1 in epsilon greedy, discount factor $\gamma$ is also set to 0.1. We compare the training speed of three opening sampling strategies for selecting the starting state in self-play. The first one always starts from the root. The second one uniformly samples a non-terminal state from a fixed-size first-in-first-out buffer that stores past trajectories with buffer rate 0.5, similar with the GEVC method in Go-Exploit. For random method and regret method, buffer rate of 0.5 is applied and regret method samples states simply proportion to their regrets. For each method, 6000 iterations is trained and for every 100 iterations 6000 games is played for evaluation and the average reward is calculated. We done the entire progress for 25 different seeds to reduce stochastic variance.

In Figure 3b, we compare the difference between the root's estimated Q-value and its theoretical optimal Q-value across the three methods.

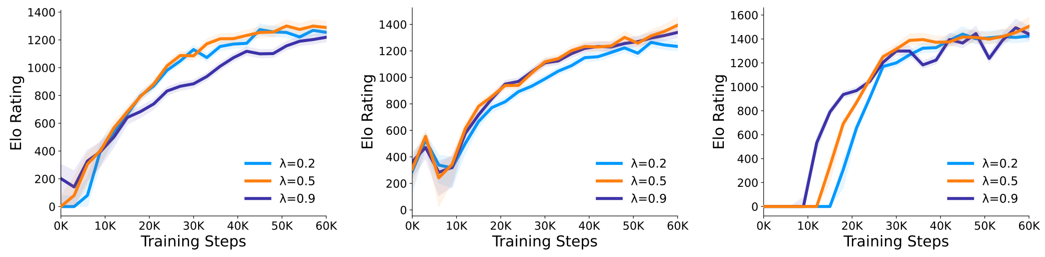
## E Performance under Different Hyperparameter Settings in RGSC

In this section, we explore the hyperparameters in RGSC, including the sampling probability ($\lambda$), the buffer sampling temperature ($\tau$), the buffer size ($\kappa$), and the EMA coefficient ($\alpha$) across 9x9 Go, 10x10 Othello, and 11x11 Hex. In the ablation study, we aim to choose the setting with less computational cost, better performance. Additionally, we select the setting that demonstrates stable and consistent results across different games.

### E.1 Sampling Probability

We evaluate different sampling probabilities ($\lambda$), which represents the probability (as introduced in Section 3.3) of starting a self-play trajectory from a state sampled from the PRB. We evaluate $\lambda \in \{0.2, 0.5, 0.9\}$, as shown in Figure 9. Overall, RGSC with $\lambda = 0.5$ performs consistently well across the three games, so we use $\lambda = 0.5$ for our final setting. In 11x11 Hex (Figure 9c), a high sampling probability ($\lambda = 0.9$) yields an early improvement around 10K training steps, but subsequently causes significant performance instability toward the end of the training process. Moreover, in 9x9 Go, it also shows that $\lambda = 0.9$ exhibits marginally reduced stability (Figure 9a). These may be due to overly frequent sampling of recent self-play data from the PRB.

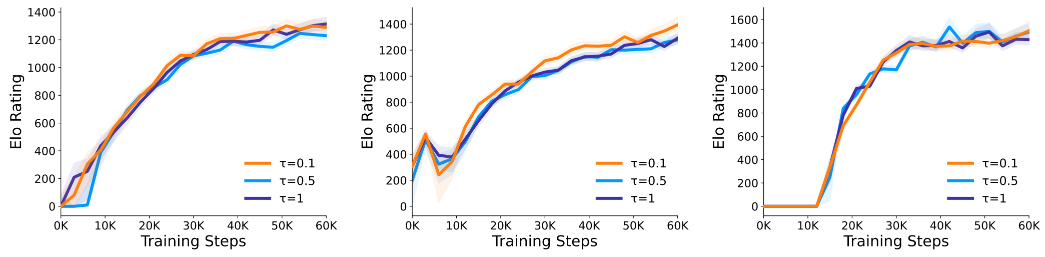
### E.2 Buffer Sampling Temperature



(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 9: Playing performance of different sampling probabilities ($\lambda$) across three games. The orange curve is the final RGSC setting.

Furthermore, we investigate the effect of the buffer sampling temperature ($\tau$) described in subsection 3.3, where a higher value leads to a more uniform softmax distribution for the PRB. We test $\tau \in \{0.1, 0.5, 1\}$, as shown in Figure 10, and the results show that $\tau = 0.1$ achieves the best performance.

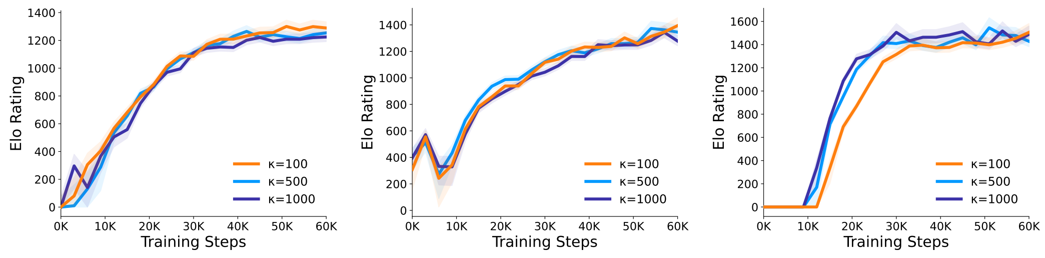


(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 10: Playing performance of different buffer sampling temperatures ($\tau$) across three games. The orange curve is the final RGSC setting.

## E.3 Buffer Size

We evaluate different buffer sizes, $\kappa \in \{100, 500, 1000\}$ in RGSC, as shown in Figure 11. The results show no significant difference in performance, so we use $\kappa = 100$ in our final setting to minimize computational cost.

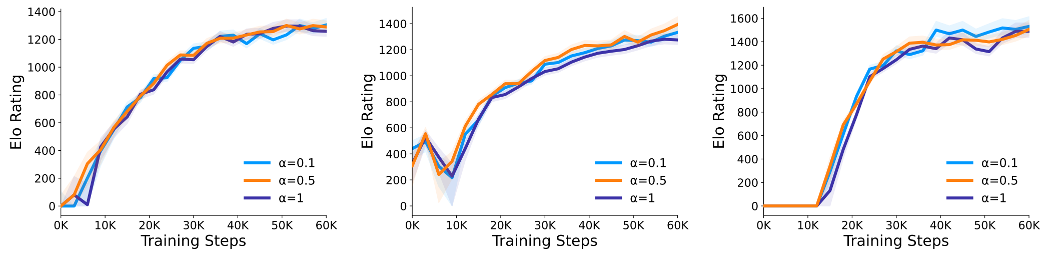


(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 11: Playing performance of different buffer sizes ($\kappa$) across three games. The orange curve is the final RGSC setting.

## E.4 EMA Coefficient

For the EMA coefficient used to update the regret values in Equation 8 of the main text, we test $\alpha \in \{0.1, 0.5, 1\}$, as shown in Figure 12, finding that $\alpha = 0.5$ generally yields the best performance in RGSC. Note that $\alpha = 1$ only uses the regret calculated from the newly finished self-play game, explaining its slightly unstable performance in 10x10 Othello (Figure 12b), and 11x11 Hex (Figure 12c).

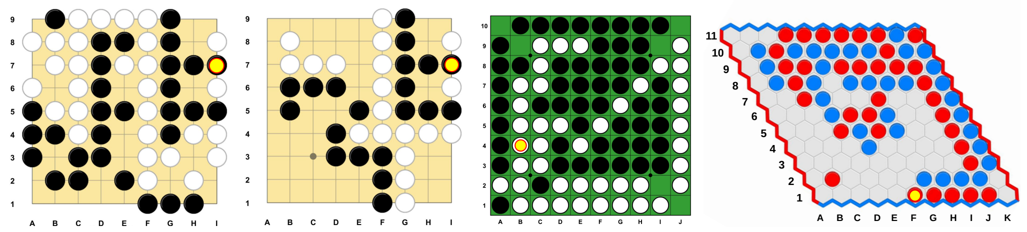


(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 12: Playing performance of different EMA coefficients ($\alpha$) across three games. The orange curve is the final RGSC setting.

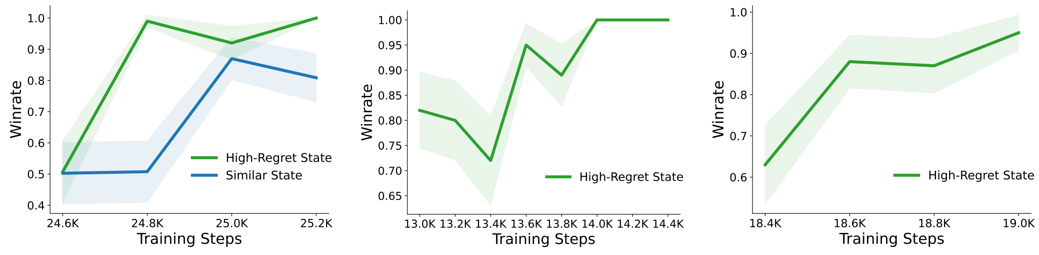
In conclusion, these experiments show that RGSC is not highly sensitive to hyperparameter choices and that our recommended configuration works well across different games, demonstrating the overall robustness of the method.

## F Analysis of High-Regret States



(a) 9x9 Go (White's Turn) (b) A state similar to (a) (White's Turn) (c) 10x10 Othello (Black's Turn) (d) 11x11 Hex (Blue's Turn)

Figure 13: Example of high-regret states.



(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

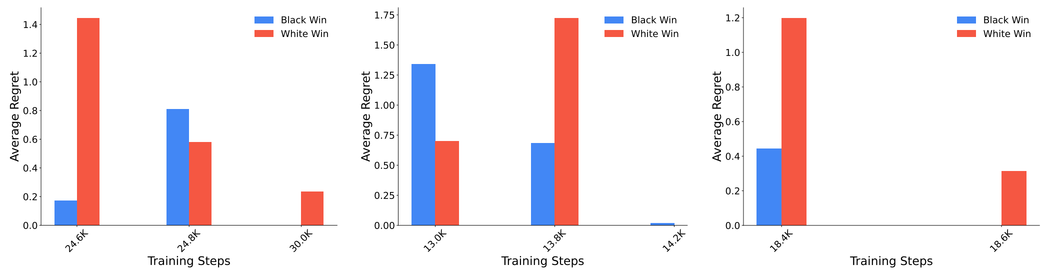
Figure 14: Win rates of high-regret states during RGSC training.

To further examine the high-regret states stored in the regret buffer, we analyze states trained in 9x9 Go, 10x10 Othello and 11x11 Hex For 9x9 Go, the high-regret state in Figure 13a appears at 2,600 training steps, 124 iterations and is repeatedly selected in the buffer, which is White's turn.

In this state, White can only win through keeping the stones at I8 and I6 alive. Once White plays at H9, there would be a seki (Niu et al., 2006) situation at the top-right corner. In a seki situation, Black and White cannot capture each other's stones. The player who plays inside the seki area first will have their stones captured by the opponent. Forming the seki situation is the only way that leads to White's victory. To evaluate whether the agent has learned the technique for solving this high-regret state, we use AlphaZero models trained with 60,000 training steps as baselines to fight against and start playing games from these states for evaluation. Once the agent makes no mistakes on these openings, the winrate will be 1. In 9x9 Go, the winrate on the high-regret opening increases significantly from 47% to 99% after the first update, and for the subsequent three weights it remains consistently above 90%.

To test whether the agent has truly learned to solve such seki situations, we provide a 9x9 Go state (Figure 13b) that shares the same pattern as the one in Figure 13a, where White can only win by playing H9. Remarkably, after training with the weights with 25,000 training steps, White's winrate in this state increases from 47% to 85%. These results demonstrate that the prioritized regret buffer enables the agent to identify weaknesses in its current policy and correct them automatically, while also generalizing to structurally similar situations.

In 11x11 Hex, another high-regret example is observed as shown in Figure 13d. In the Figure 13d, the Blue stone at G2 can connect through D4 and G5, guaranteeing a win for Blue. However, the first update on 18,400 training steps causes the winrate on this opening to surge to 88%. After one additional round of training, at the 94th iteration, the winrate rose to 91%, reflecting a 14% improvement within a single iteration. For a high-regret state in 10x10 Othello, the first update around 13,000 training steps shows little improvement; however, after the second update at 13,800 training steps, the winrate jumps to 100%. These experiments demonstrate that our method not only enables the agent to correct its mistakes but also enhances the interpretability of the AlphaZero framework.



(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

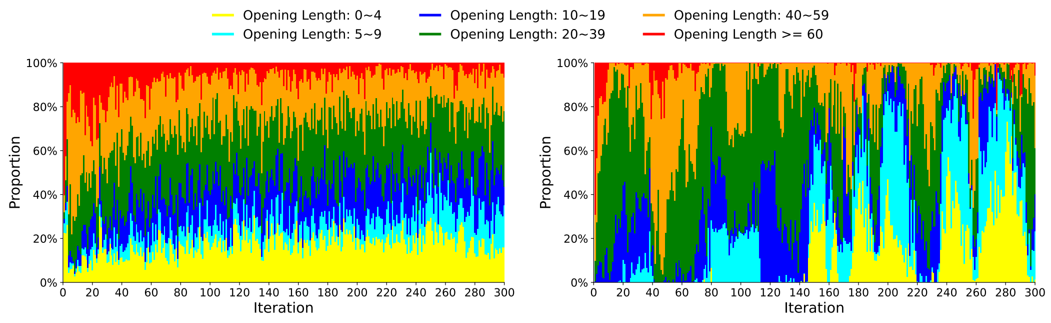
Figure 15: Relationship between game outcomes and regrets during training.

We also plot the relationship between the game outcome and the corresponding regret in **??**. For a state where White always wins under the optimal play, the regret associated with White's victory decreases as the agent's predictions on that state become more accurate. At the same time, the proportion of self-play games won by White also increases significantly as training progresses. As shown in Figure 15a and Figure 15c, where White wins under the optimal policy, the regret for White's victory diminishes with more training iterations on these openings, while the regret for Black's victory correspondingly increases. Conversely, in cases where Black has a guaranteed win, such as in Figure 15b, the variation evolves in the opposite direction.

## G Analysis of Prioritized Regret Buffer

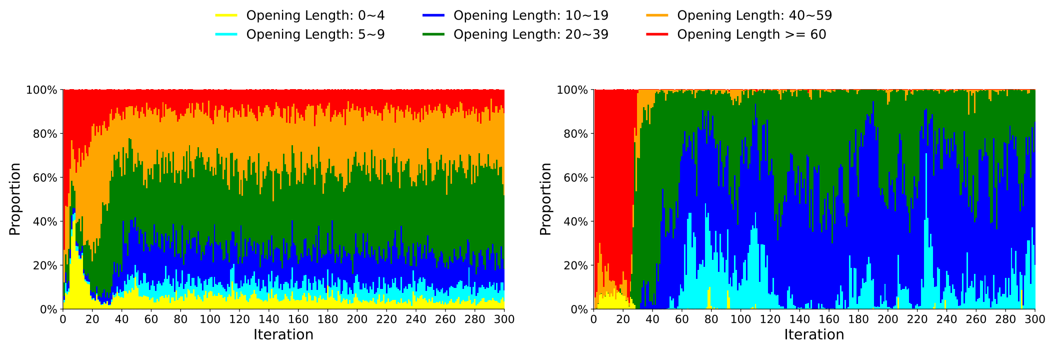
In this section, we investigate the openings' attributes in PRB. In subsection G.1, we investigate the distribution of opening lengths during training and how the model adapts to different phases of the game. Finally, in subsection G.2, we track how the regret values of the openings evolve throughout training, demonstrating that the model becomes increasingly familiar with the high-regret states and refines its policy accordingly.

### G.1 Opening-Length Distributions in Training Process



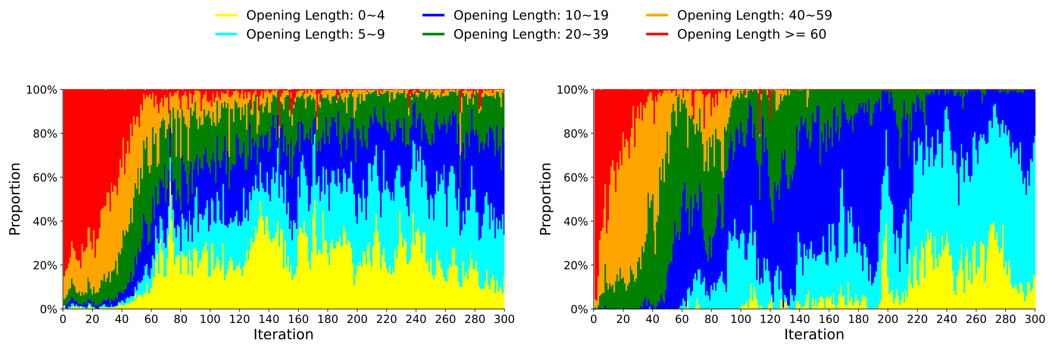
(a) RGSC w/o regret ranking network (b) RGSC

Figure 16: Change in the proportion of openings with different lengths across training in 9×9 Go.



(a) RGSC w/o regret ranking network (b) RGSC

Figure 17: Change in the proportion of openings with different lengths across training in 10×10 Othello.



(a) RGSC w/o regret ranking network (b) RGSC

Figure 18: Change in the proportion of openings with different lengths across training in 11×11 Hex.

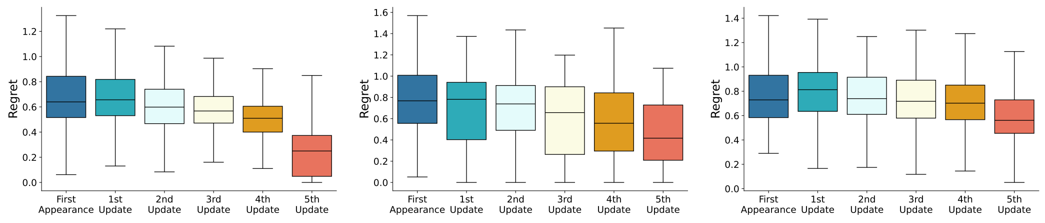
In board games such as Go, Hex, and Othello, the game is typically divided three phases: early game, midgame, and endgame. Among these, the midgame typically involves the most complex tactics and combinatorial challenges, offering the greatest potential for learning. To see which phase of the game is preferred for learning during training, we examine the length of openings used in PRB throughout the training process.

Figure 16 shows the dynamics of the proportion of openings with different lengths throughout the training process in RGSC with and without regret ranking network in 9x9 Go. RGSC without regret ranking network shows no significant preference for opening lengths, as shown in Figure 16a. In contrast, RGSC exhibits a clear phase-aware curriculum. In the early stages of training, it favors openings with opening lengths ranging from 20 to 39, corresponding to midgame positions, which are the most complex and informative. As training nears the end, the model's overall strength improves, allowing self-play games to reach the midgame phase with fewer steps. As a result, the distribution shifts towards shorter opening lengths ranging from 5 to 9. This indicates that RGSC still targets the most worth-learning states as the MCTS value estimation improves.

Figure 17 and Figure 18 show the results of the analysis in Hex and Othello. We observe the same qualitative pattern: relative to the RGSC without regret ranking network, RGSC progressively biases toward shorter opening lengths as training advances. The consistency across various games indicates that RGSC selects openings based on the model's playing strength, allowing it to identify the most valuable learning states at different training stages.

The dynamics of opening lengths show that RGSC induces an implicit, data-driven curriculum on game phases: initially focusing on complex midgame positions, then shifting shorter openings as the model's playing strength improves. In contrast, RGSC without regret ranking network remains less sensitive to the phases and difficulty of the game.

## G.2 Decreasing Regret Values Across Training



(a) 9x9 Go (b) 10x10 Othello (c) 11x11 Hex

Figure 19: Changes in the values of openings from their first appearance to the final update during training.

In Figure 19, we analyze how the regret values of the openings in the regret buffer change throughout the training process. Specifically, we track the regret values of openings from their first appearance to their final removal, after being updated across five different iterations. The results show that the regret values consistently decrease with each update across the three games, indicating that the model has learned and become familiar with these high-regret openings. By the final update, the regret distribution has shifted significantly towards the lower values, resulting in the lowest regret value across all iterations, just before the opening is removed from the buffer. It shows that the model progressively focuses more on learning these challenging states and reduces their regret over time to optimize as training progresses.

## H RGSC ON ATARI GAMES

RGSC can also be generalized to MuZero (Schrittwieser et al., 2020) or other AlphaZero-variants. To demonstrate this, we further integrate RGSC into MuZero in a large-scale domain, the Atari benchmark. Specifically, we select one of the Atari games, *Ms. Pac-Man*, in our evaluation. Unlike board games, where the outcome is only available at the terminal state, we use the internal rewards to compute n-step return for $z$ in Equation 2 for all intermediate states in Atari games.

Since Go-Exploit is implemented only within AlphaZero for board games, our experiment focuses on comparing RGSC with MuZero. Throughout the entire training process, RGSC significantly outperforms MuZero, as shown in Figure 20. At the final iteration, RGSC achieves an average score of 5166 points, while MuZero only achieves 3704 points, demonstrating the generality of RGSC when applied to other AlphaZero-like style algorithms and highlighting its ability to handle complex tasks.

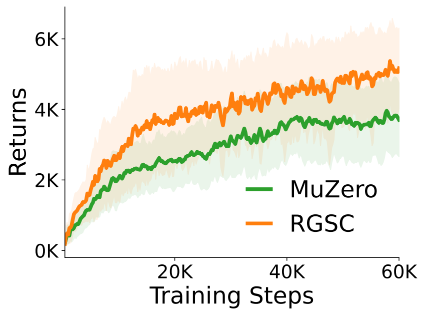


Figure 20: Playing performance of MuZero, and RGSC on Ms. Pac-Man. The shaded area is a 95% confidence interval for the mean.