Targeted Search Control in AlphaZero for Effective Policy Improvement

by

Alexandre Trudeau

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

© Alexandre Trudeau, 2023

Abstract

AlphaZero is a self-play reinforcement learning algorithm that achieves superhuman play in the games of chess, shogi, and Go via policy iteration. To be an effective policy improvement operator, AlphaZero's search needs to have accurate value estimates for the states that appear in its search tree. The accuracy of AlphaZero's value function depends upon the distribution of states encountered and trained upon. AlphaZero begins its self-play training matches 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. In this thesis, I 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 states throughout the game tree 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, further 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, I 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. I also compare Go-Exploit to KataGo, a more sample efficient reimplementation of AlphaZero, and show that Go-Exploit's search control strategy exhibits a greater sample efficiency than KataGo's. Furthermore, Go-Exploit's sample efficiency improves when KataGo's other innovations are incorporated.

Preface

The work presented in this thesis will be published as Alexandre Trudeau and Michael Bowling, "Targeted Search Control in AlphaZero for Effective Policy Improvement," in *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023)* to be held in London, United Kingdom from May 29 to June 2, 2023.

This thesis is dedicated to my friend Willis Sanchez-Dupont. Willis had an insatiable curiosity and a passion for artificial intelligence. Thank you for your friendship and endless encouragement.

Acknowledgements

I would like to begin by thanking my supervisor, Michael Bowling. One on one meetings with Mike are a treat, although they've been known to leave me rather sweaty! It takes a week to prepare for a meeting with Mike and another week thereafter to fully unpack what was discussed. Gaining insight into how Mike deconstructs and thinks about problems and solutions has been invaluable and has helped me mature as a scientist. I would also like to thank my parents, Ilana and Pierre, as well as my girlfriend, Christa. These past few years have been very challenging, particularly with all of my setbacks. Thank you for always believing in me and supporting me along the way. Finally, thank you to Compute Canada for offering the necessary computational resources to complete this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Games such as chess, checkers, and Go have been used as test environments for Artificial Intelligence (AI) since the inception of the field. AI pioneers including Alan Turing [1], Claude Shannon [2], and Arthur Samuel [3] sought to investigate whether machines could display intelligent behaviour and whether human thought and learning could be mechanized or automated. Early AI practitioners believed that games require intelligence to play skillfully, and thus, make ideal environments to test machine intelligence. On a more practical level, games present constrained, challenging problems to solve. The rules of a game are easy to implement in code and constrain the behaviour of an agent. Furthermore, the agent's goal is clear – to win the game. With modern computers, moves and games can be simulated very quickly, allowing algorithms to perform extensive look-ahead searches and to learn from large amounts of data. The ability to quickly simulate games also allows scientists to evaluate an algorithm's sample efficiency within days or hours.

A long-standing goal in artificial intelligence has been to create general algorithms that can learn for themselves without human intervention. Early AI research in games focused on tree search algorithms utilizing handcrafted evaluation functions [2–5]. These algorithms would simulate sequences of actions from the current state of a game, evaluate the resulting board positions with functions representing human game knowledge, and select the action with the greatest estimated value. Such methods

were central to the programs that first defeated the world checkers [6] and chess [7] champions. Despite the success of these programs, they could not be generalized to other games. Their underlying evaluation functions were game specific. Furthermore, the human knowledge encoded in these functions upper bounded the abilities of these programs.

Progress towards a general game playing program was made with the development of TD-Gammon [8]. TD-Gammon replaced the traditional handcrafted evaluation function used in tree search with a neural network that learned to represent backgammon strategy from games played against itself. The weights of the neural network were updated via temporal-difference (TD) learning [9], a reinforcement learning (RL) algorithm that learns to predict the future reward that could be obtained from a state by reducing the value estimation error between temporally successive predictions.

TD-Gammon ultimately paved the way for AlphaZero [10, 11]. AlphaZero is a model-based reinforcement learning algorithm that has achieved impressive results in two-player, zero-sum games, reaching superhuman play in chess, shogi, and Go. Similarly to TD-Gammon, AlphaZero synthesizes tree search, neural network evaluation, and reinforcement learning. 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 look-ahead 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. AlphaZero is now widely regarded as the state of the art in the domain of two-player zero-sum games. However, despite its success, AlphaZero's training suffers from sample inefficiency, requiring hundreds of millions of training samples to attain superhuman play in 19x19 Go ([11] Figure 1c).

1.1 Contributions

AlphaZero's sample efficiency depends upon the distribution of states encountered 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 [12], 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.

I hypothesize that AlphaZero could address these limitations and learn with greater sample efficiency if it utilized 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" ([12] pg. 163). In AlphaZero, this amounts to strategically choosing the starting state of its simulated trajectories

rather than always beginning from the initial state of a game. I 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 thesis, I introduce Go-Exploit, a novel search control strategy for AlphaZero. Go-Exploit takes inspiration from algorithms such as Go-Explore [13] and Exploring Restart Distributions [14], which begin simulated episodes from previously visited states sampled from a memory. Similarly, Go-Exploit maintains an archive of states of interest. At 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 thesis, I experiment with two definitions of "states of interest" and three archive structures.

In the games of Connect Four and 9x9 Go, I 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. I also compare and contrast Go-Exploit and KataGo [15], 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. I conclude by showing how Go-Exploit's adherence to the guiding principles enables it to learn more effectively than AlphaZero.

Chapter 2

Background

Since Go-Exploit builds off of AlphaZero, I begin by introducing the key algorithms and concepts that underpin the AlphaZero algorithm. In this chapter, I start by mathematically formalizing the games under consideration. Then, I describe the tree search algorithms that have influenced AlphaZero's search. Next, I introduce the exploration-exploitation trade-off and explain how it has been addressed in the bandit and search settings. After that, I present the fundamentals of reinforcement learning and highlight how policy iteration is used as a means of obtaining stronger policies. I conclude by introducing approaches to model-based reinforcement learning and detailing search control procedures that have been used in the tabular and function approximation settings. In Chapter 3, I present a full review of the AlphaZero algorithm. To learn more about the history of algorithmic developments that ultimately led to AlphaZero, please refer to Appendix A.

2.1 Games Under Consideration

Games are traditionally classified by properties that describe the number of players, the observability of states, the number of actions, the stochasticity, and the way the game proceeds over time [16]. In this thesis, I only consider classic board games such as Connect Four and Go that are two-player, zero-sum, perfect-information, sequential, discrete, and deterministic. Games are categorized as two-player zero-sum

if there are two players and the reward amongst all players adds up to zero. In zero-sum games, players are directly competing with each other for reward and victory. For example, heads-up poker is a two-player zero-sum game because when Player A wins $100, Player B loses $100 and the sum of the rewards is $0. Go is also a two-player zero-sum game because when Player A wins (+1), Player B loses (-1). Games are considered perfect-information when the state s of the game is fully observable to all players. For example, Go is perfect-information because all of the pieces that have been placed on the board are visible to both players. Poker, on the other hand, is an imperfect information game because each player has cards that they can see that are hidden from their opponent. In sequential games, players take turns selecting actions according to some predefined order. In the games under consideration, the two players alternate turns over a discretized time scale. The games begin from a start state s_0. At time t, the function $\rho(t)$ returns which player's turn it is. Player $\rho(t)$ observes the current state S_t and selects an action A_t, causing the game to transition into state S_{t+1}. This process is repeated until the game transitions into a terminal state, at which point the game ends. Games are considered discrete when the state space $\mathcal{S}$ and action space $\mathcal{A}(s)$ are finite. The rules of the game determine the set of legal states $\mathcal{S}$, constrain which legal actions $\mathcal{A}(s)$ can be taken from a given state, and dictate how actions change the state of the game. Finally, games are deterministic when there is no stochasticity in the state transition function $\Gamma(s, a)$ or reward function $\psi(s, a, s')$. In other words, $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}(s)$, $\exists s' \in \mathcal{S} \mid \Pr(\Gamma(s, a) = s') = 1$ and $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}(s)$, $\exists s' \in \mathcal{S}$, $\exists r \in \mathbb{R} \mid \Pr(\psi(s, a, s') = r) = 1$.

2.2 Tree Search

In the context of games, *tree search* is a term used to refer to decision-time planning algorithms that search through a tree of possible state-action trajectories beginning from the current state to determine the action that is taken on the given turn [12]. At time step t, player $\rho(t)$ builds a search tree from a root node corresponding to

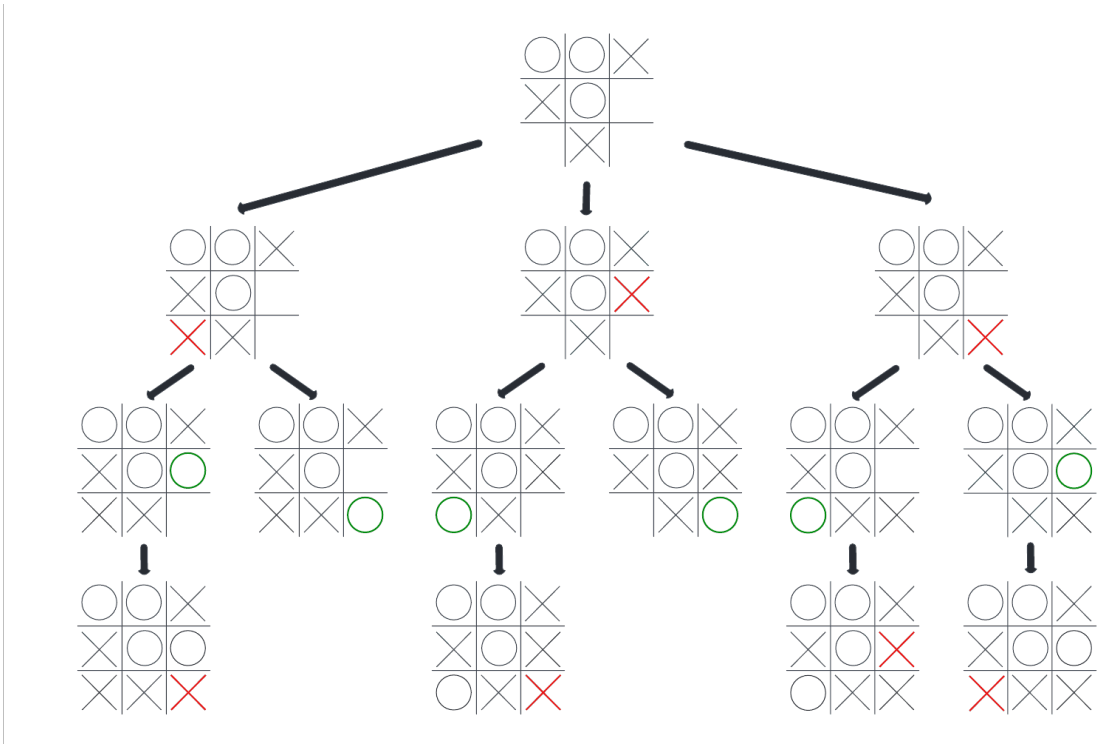


Figure 2.1: The complete game tree from a root state in Tic-Tac-Toe. Edges, representing a given action a, connect a state s to their successor state $\Gamma(s, a)$. The move taken is highlighted in red for 'x' and in green for 'o' in a successor state.

state S_t. When unconstrained by search time and memory, tree search algorithms often produce a complete game tree from the given root state consisting of all possible state-action trajectories. Complete game trees can be recursively produced from the root state S_t. For each legal action $a \in \mathcal{A}(S_t)$ that can be taken from S_t, there is an edge connecting the root node to the corresponding successor state $\Gamma(S_t, a)$. The remainder of the search tree is recursively built the same way. Terminal states (game ending states) form the leaf nodes of the complete game tree. The outcomes at the terminal states can be used to determine the optimal action(s) that can be played from the root state. In Figure 2.1, a complete game tree can be seen for a root state in Tic-Tac-Toe.

When the state space $\mathcal{S}$ and action space $\mathcal{A}(s)$ are prohibitively big for an exhaustive search, heuristics are used to help narrow the search to promising states and actions. Some tree search algorithms limit computation by simply producing a

complete game tree to a predetermined depth d. In other tree search algorithms, the traversal and expansion of the search tree is guided by a utility function [16]. These algorithms evaluate the utility of states at leaf nodes in the search tree. If a state's utility is greater than the utility of other states, it is estimated to be a more favourable position for the current player. The utilities assigned to the leaf nodes are backed up the search tree and are used to assign utilities to the edges (state-action pairs) or states further up the tree. The backed up utilities help the search algorithm focus on the most promising parts of the search tree. Ultimately, the utilities of the actions that can be taken from the root node determine the action that is selected from state S_t at time t.

2.3 Minimax Search

Minimax Search is a tree search algorithm used to select actions in two-player zero-sum games. Minimax Search assumes that both players are playing optimally. Accordingly, it is assumed that each player will select the action with maximum utility on their turn. Given the opponent will take the action with maximum utility, the best a player can do is take the action that minimizes the maximum utility available to the opponent. This is called the Minimax Principle [17] and is at the heart of Minimax Search.

Minimax Search builds a complete search tree from the current state S_t to a predetermined depth d. The nodes that correspond to the current player's turn are labelled "Max" nodes and the nodes that correspond to the opponent's turn are labelled "Min" nodes. Thus, "Max" nodes and "Min" nodes alternate at each depth of the search tree. Once the search tree is built, the Minimax Search algorithm works its way from the leaf nodes back up to the root node to determine the action taken by the current player. If a leaf node is a terminal state, it is assigned a utility corresponding to the outcome of the game for player $\rho(t)$. Wins are assigned a utility of $+\infty$, losses are assigned a utility of $-\infty$, and draws are assigned a utility of 0. If a leaf node is not

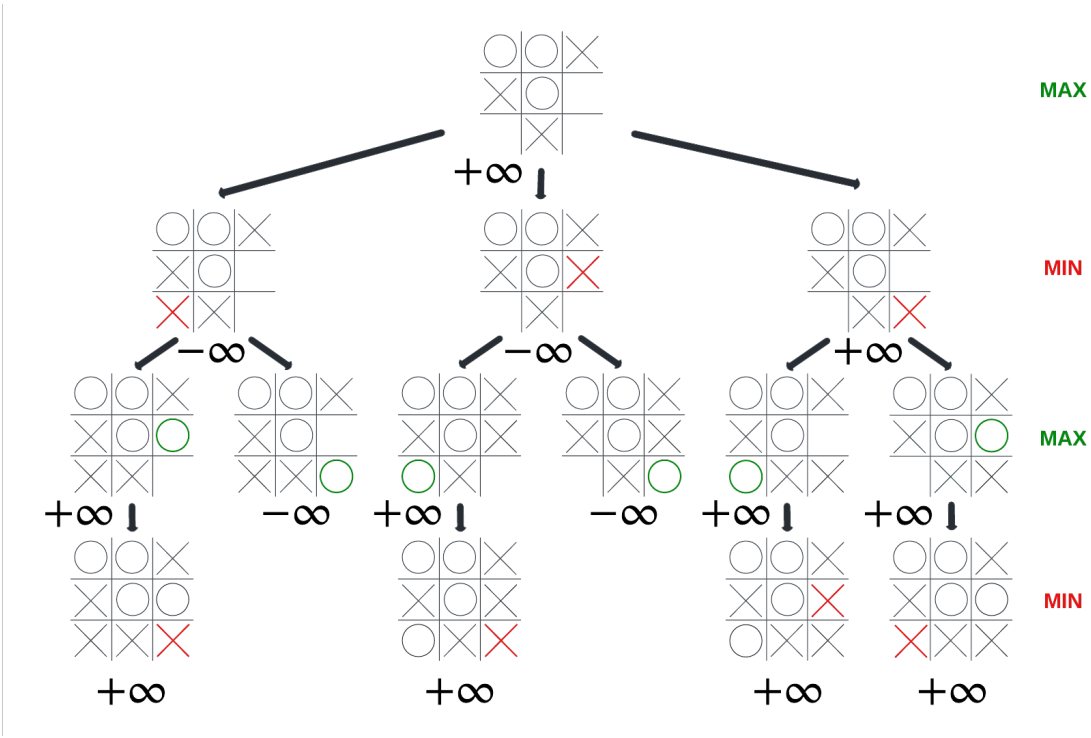


Figure 2.2: Minimax Search from a root state in Tic-Tac-Toe. Each of the leaf nodes evaluate to wins ($+\infty$) or losses ($-\infty$). 'Max' nodes take on the utility of the child with maximum utility. 'Min' nodes take on the utility of the child with minimum utility. Upon the completion of the search, the root state has a utility of ($+\infty$), meaning that the state is a proven win under perfect play.

a terminal state, an evaluation function is used to determine its utility. Once each of a node's children has been assigned a utility, the given node's Minimax value can be determined. If the node is a "Max" node, the node takes the value of the child with the maximum utility. This implements the notion that the current player must choose the action with maximum utility in order to play optimally. If the node is a "Min" node, it takes the value of the child with the minimum utility. This implements the notion that the opponent can only play optimally if it takes the action minimizing the maximum utility available to the current player. Once the root node has been assigned a utility, the current player's optimal action is to take an action corresponding to the edge leading to the child with the same utility. An example of Minimax Search can be seen in Figure 2.2. In this example, all of the leaf nodes are terminal states whose outcomes are wins or losses. These values are backed up the

search tree until, ultimately, the root node is assigned a utility of $+\infty$. This means the root state is a proven win and the optimal action is to place an 'x' in the bottom right cell of the Tic-Tac-Toe board.

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [18–20] is also a decision-time planning algorithm used to select actions in sequential decision problems with a finite action space. MCTS is a rollout algorithm, meaning that it relies upon sampled trajectories from the root state S_t to estimate the average sum of rewards, $Q(s, a)$, that can be obtained from each of the state-action pairs (s, a) within the search tree. $Q(s, a)$, also known as the action value of a state-action pair, usually factors into the action selection, encouraging the search to focus on the most promising state-action pairs. With more trajectories passing through the most promising state-action pairs, MCTS can better estimate their action values, allowing it to select the optimal action with more reliability.

MCTS incrementally builds out a non-uniformly expanded search tree over a finite number of iterations. The algorithm begins with a root node corresponding to the current state S_t. Each iteration of MCTS consists of four steps: selection, expansion, simulation, and backpropagation. In the selection step, MCTS begins a trajectory from the root node. Until a leaf state is encountered, MCTS selects an action using a tree policy that usually favours the selection of actions with large action values $Q(s, a)$. Once an unvisited action is selected from a leaf node s_L, the expansion step takes place. In the expansion step, the node s_L and the selected action a are passed to the transition function $\Gamma(s_L, a) = s'$ and the successor state s' is added to the search tree as a child of s_L. Then, the simulation step occurs. The remainder of the trajectory is played out starting from the expanded node s' using a rollout policy (often random action selection). Once the trajectory reaches a terminal state, the backpropagation step occurs. MCTS backs up the result of the simulation to the

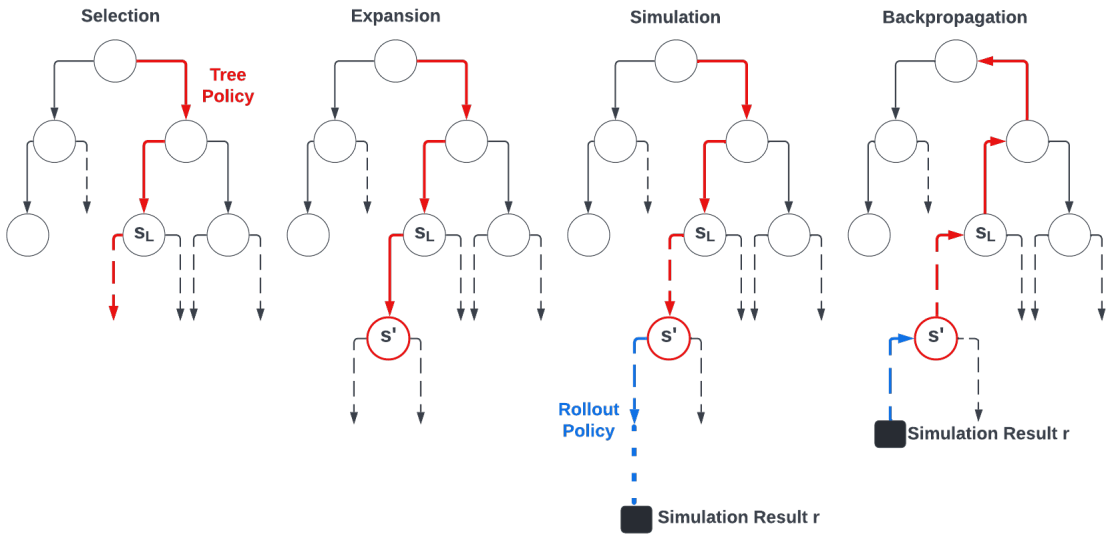


Figure 2.3: The four steps of MCTS: selection, expansion, simulation, and backpropagation. In the selection step, actions are selected with a tree policy (red edges) until an unvisited action is traversed. In the expansion step, the successor state s' is added to the search tree. In the simulation step, the remainder of the trajectory is simulated from s' using a rollout policy (blue edges). In the backpropagation step, the simulation result r is backed up to the edges that were traversed within the search tree and is used to update their action values.

state-action pairs that were traversed within the search tree, updating their action values $Q(s, a)$. Once the final iteration of MCTS is complete, the algorithm often selects the action from the root node that was either most visited or had the largest action value. The four steps of MCTS are visually broken down in Figure 2.3.

2.5 Exploration-Exploitation Trade-Off

The effectiveness of Monte Carlo Tree Search depends upon its ability to accurately estimate the action values of the legal actions that can be taken from the root state. Simply selecting the action with the largest action value $Q(s, a)$ does not guarantee that the optimal action will be taken. If the uncertainties on the action value estimates are too large, it is quite probable that the action with the largest action value will not be the optimal action. In order to reduce the uncertainty of a state-action pair's

action value estimate, the search must continue to traverse the state-action pair. Each time a state-action pair is traversed, its action value estimate is averaged over a larger sample size, reducing the uncertainty in the estimate. In MCTS, however, the uncertainties of the action values cannot be reduced to arbitrarily small values because MCTS operates over a finite number of iterations. MCTS' challenge is to carefully distribute the finite number of trajectories over the state-action pairs so that by the end, the uncertainties of the root state's action values are small enough that the action with the largest action value is indeed the optimal action with high probability. This challenge of intelligently distributing the trajectories amongst the state-action pairs is called the exploration-exploitation trade-off. In MCTS, the tree policy is responsible for navigating this exploration-exploitation trade-off. The tree policy must ensure that at a given state, each action is continuously traversed until its respective uncertainty is small enough that it can be confidently ruled out of having the largest action value.

Implementing tree policies which take into account the exploration-exploitation trade-off is easy, however, implementing tree policies which establish probabilistic guarantees for selecting the optimal action is much more difficult. In the next section, I introduce the multi-armed bandit, a simple sequential decision problem also requiring mastery of the exploration-exploitation trade-off. After outlining the multi-armed bandit problem, I present bandit algorithms that have influenced how search algorithms address the exploration-exploitation trade-off. I then introduce UCT, a variant of MCTS whose tree policy is inspired by a popular bandit algorithm called UCB1 and that bounds the failure probability of returning a suboptimal action upon the completion of the algorithm.

2.6 Multi-Armed Bandit Problem

Perhaps the simplest form of sequential decision making under uncertainty is the multi-armed bandit problem [21–24]. The multi-armed bandit is a term used to

describe a slot machine with multiple levers (arms). Formally, a k-armed stochastic stationary bandit consists of k reward distributions $P_a : a \in 1, ..., k$, each with a mean reward μ_a, that are unknown to an agent. The multi-armed bandit problem operates sequentially over a fixed horizon of n rounds. Each round $t \in 1, ..., n$, the agent selects an action A_t. A reward $X_t \sim P_{A_t}$ is then sampled from arm A_t's reward distribution and returned to the agent. The rewards X_t that are sampled over the n rounds have no dependence upon the history $H_{t-1} = (A_1, X_1, ..., A_{t-1}, X_{t-1})$ of actions and observed rewards. In stochastic stationary bandits, the sequence $X_{i,1}, X_{i,2}, \ldots$ of rewards drawn from an arm i are independent and identically distributed. When selecting an action each round, the agent can only make use of the observed history H_{t-1} to inform its decision. The agent's policy π_t in round t defines a probability distribution over the legal actions conditioned on the history H_{t-1}: $\pi_t(\cdot|H_{t-1})$.

Over the n rounds, the agent's goal is to maximize its total reward $W_n = \sum_{t=1}^{n} X_t$. A related performance measure that is traditionally used to evaluate an agent is the regret. Broadly, an agent's regret relative to a policy π is the difference between the expected cumulative reward by policy π over n rounds and the expected cumulative reward by the agent over the n rounds. The worst-case regret is measured relative to the optimal policy that pulls the optimal arm each round. The optimal arm is the arm whose reward distribution P_* has the largest mean payoff μ_*. Thus, the worst-case regret can be stated as $R_n = n\mu_* - \mathbb{E}[\sum_{t=1}^{n} X_t]$. Minimizing this worst-case regret is equivalent to maximizing the cumulative reward W_n.

In order to minimize the worst-case cumulative regret, an agent must manage the exploration-exploitation trade-off. Minimizing the worst-case regret requires pulling the optimal arm as frequently as possible. Since the agent has no knowledge of the reward distributions P_a, it must estimate which of the arms is optimal. Agents estimate the mean payoff for each arm by computing the sample mean of the observed rewards: $\hat{\mu}_i = \frac{1}{N(i)} \sum_{t=1}^{N(i)} X_{it}$, where $N(i)$ is the number of times arm i has been pulled. Since the agent's goal is to maximize its cumulative reward over the n trials, it must

frequently pull the arm it estimates to have the largest mean payoff (exploitation). However, in order to accurately determine the optimal arm, the agent must pull each arm multiple times and sufficiently reduce the uncertainties in their sample means (exploration). This exploration-exploitation trade-off in the bandit setting resembles the exploration-exploitation trade-off described in the context of MCTS. In MCTS, however, the exploration-exploitation trade-off is concerned with the accurate estimation of action values $Q(s, a)$ and the identification of promising actions from a root node.

A primary concern in the study of bandits is how an agent's regret grows as the horizon, n, increases. Good agents achieve sublinear regret, meaning that their regret R_n respects the limit $\lim_{n\to\infty} \frac{R_n}{n} = 0$. Lai and Robbins [25] proved that the best regret that can be achieved by an agent is $O(\log n)$. Agents that achieve this logarithmic regret bound are said to have solved the exploration-exploitation trade-off.

2.7 UCB

The Upper Confidence Bound algorithm (UCB) [24–29] addresses the exploration-exploitation dilemma by making use of the *optimism in the face of uncertainty* principle. Under this principle, the agent pulls the arm with the greatest *upper confidence bound* each round. The upper confidence bound overestimates the mean payoff μ_i by summing the estimated mean payoff $\hat{\mu}_i$ and the agent's uncertainty in its estimate. As the agent obtains more reward samples for each arm, its average reward estimates improve in accuracy and their uncertainties decrease, causing the upper confidence bounds to compresses towards the true mean rewards μ_i. Once a suboptimal arm's upper confidence bound falls below the optimal arm's mean reward μ_*, it is no longer pulled because in theory, the optimal arm's upper confidence bound should overestimate μ_*. This phenomenon limits the number of times suboptimal arms are pulled, allowing the UCB algorithm to achieve the optimal logarithmic regret bound.

The UCB algorithm uses the quantity $\hat{\mu}_i + \sqrt{\frac{2\log(1/\delta)}{N(i)}}$ as an upper confidence bound [24]. The constant δ is called the confidence level and it establishes the probability with which the upper confidence bound is an overestimate of μ_i. $N(i)$ represents the number of times arm i has been pulled prior to the current trial. The square root term is known as the exploration bonus or confidence width and measures the uncertainty in the average reward estimate. This leaves us with the UCB Algorithm found in Algorithm 1.

Algorithm 1 UCB

Parameters: The number of arms k, the horizon n, and confidence level δ.

1: **for** $t \in 1, ..., n$ **do**
2: $A_t = \text{argmax}_i \text{UCB}_i(\delta)$.
3: Pull arm A_t and receive reward X_t.
4: Update $\hat{\mu}_{A_t}$ with X_t.
5: $N(A_t) = N(A_t) + 1$
6: **end for**

$$\text{UCB}_i(\delta) = \begin{cases} \infty & \text{if } N(i) = 0 \\ \hat{\mu}_i + \sqrt{\frac{2\log(1/\delta)}{N(i)}} & \text{otherwise} \end{cases}$$

Looking at $\text{UCB}_i(\delta)$, it's easy to see how UCB navigates the exploration-exploitation dilemma. If an arm's average reward estimate $\hat{\mu}_i$ is significantly larger than the average reward estimates of other arms, then its UCB value will also be larger, leading the agent to exploitatively pull this arm. If an arm hasn't been pulled very much, its $N(i)$ value will be smaller compared to other arms, causing its exploration bonus to be larger. This leads the algorithm to explore arms that are perceived to be suboptimal if their true mean reward could reasonably be larger than the mean reward of the arm currently estimated to be optimal.

2.8 UCB1

A well-known variant of UCB is the UCB1 algorithm presented by Auer et al. [29]. Unlike the UCB algorithm introduced above, UCB1 assumes that the rewards X_t are

bounded to $[0, 1]$. Furthermore, UCB1 is not defined for a predetermined horizon n. In the UCB algorithm presented earlier, the confidence level δ is usually set to $\frac{1}{n^2}$ to achieve the tightest regret guarantees, causing the upper confidence bound to be a function of n. The upper confidence bound employed in UCB1, however, does not depend upon a horizon: $A_t = \arg\max_i \hat{\mu}_i + \sqrt{\frac{2\log(t)}{N(i)}}$. Instead, it depends upon the current round number t. Not knowing the horizon ahead of time makes it more difficult for the agent to manage the exploration-exploitation tradeoff, causing the expected regret of UCB1 to be larger than that of the UCB algorithm presented earlier. However, UCB1 still preserves the ideal $O(\log(n))$ regret bound.

2.9 PUCB

The Predictor + UCB algorithm (PUCB) [30] incorporates contextual information into UCB1 to bias action selection towards arms that are predicted to be promising. PUCB solves the *multi-armed bandit problem with episode context*, which differs slightly from the stochastic stationary multi-armed bandit introduced earlier. The multi-armed bandit problem with episode context is broken up into episodes consisting of n trials. The bandit consists of k arms, each with an unknown reward distribution whose rewards are bounded to $[0, 1]$. At the beginning of each episode, contextual information z is obtained that remains fixed throughout the episode. This differs from *contextual bandits* which receive a new context each trial. A predictor maps this context z to a vector of weights $\mathbf{M}$, where $M_i > 0$ and $\sum_{i=1}^{k} M_i = 1$. For example, a predictor could map the context z to a discrete probability distribution over the actions, where the probabilities represent the likelihood of an action being optimal. An arm's contextual weight M_i is used to compute a penalty $m(t, i)$ that is combined with UCB1 to obtain PUCB action selection:

$$A_t = \arg\max_i \hat{\mu}_i(t) + c(t, i) - m(t, i)$$

$$\hat{\mu}_i(t) = \begin{cases} \frac{1}{N(i)} \sum_{j=1}^{N(i)} X_{i,j} & \text{if } N(i) > 0 \\ 1 & \text{otherwise} \end{cases}$$

$$c(t,i) = \begin{cases} \sqrt{\frac{3\log(t)}{2N(i)}} & \text{if } N(i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$m(t,i) = \begin{cases} \frac{2}{M_i}\sqrt{\frac{\log(t)}{t}} & \text{if } t > 1 \\ \frac{2}{M_i} & \text{otherwise} \end{cases}$$

The contextual additive penalty $m(t,i)$ more substantially penalizes arms whose contextual weight M_i is smaller. The algorithm gains its effectiveness when small contextual weights M_i are assigned to arms with large suboptimality gaps $\Delta_i = \mu_* - \mu_i$ and the optimal arm A_* is given a large contextual weight M_*.

Unlike UCB1, PUCB does not pull each arm at the beginning of the algorithm. Initially, when $N(i) = 0$ for each arm, $c(t,i) = 0$ so the action selection solely depends upon the contextual weights $\mathbf{M}$. The contextual penalty initially encourages PUCB to pull the arms that are predicted to be optimal but if their sample means do not validate the prediction, PUCB explores other arms and ultimately finds the optimal arm with high probability.

2.10 UCT

The Upper Confidence Bounds for Trees algorithm (UCT) [20], is a popular variant of Monte Carlo Tree Search that balances exploration and exploitation. Recognizing the similarity of the exploration-exploitation trade-offs present in MCTS and multi-armed bandits, Kocsis and Szepesvari incorporated UCB1 action selection into the tree policy of MCTS so that the search algorithm could manage the exploration-exploitation trade-off and return the optimal action with high probability. In UCT, the agent models each action selection within the search tree as a separate stochastic multi-armed bandit problem. The legal actions that can be taken from a given node correspond to the arms of the bandit and the sum of rewards obtained in the remainder of the trajectory corresponds to the selected action's payoff. As MCTS progresses,

the action values of state-action pairs in the search tree are continuously updated. Action values usually factor into the tree policy and affect the action selection during search. If the sequence of actions taken from a given state-action pair changes over time, its payoff distribution also changes. Thus, UCB1's assumption of stationary reward distributions is violated. To deal with drifting reward distributions, UCT modifies UCB1 as follows:

$$\mathrm{UCT}(s,a) = \begin{cases} \infty & \text{if } N(s,a) = 0 \\ Q(s,a) + 2C_p\sqrt{\frac{\log(N(s))}{N(s,a)}} & \text{otherwise} \end{cases}$$

where $N(s,a)$ is the number of times state-action pair (s,a) has been traversed, $N(s)$ is the number of times state s has been visited, and C_p is a constant greater than 0. When C_p is set appropriately, $O(\log(n))$ regret in the number of MCTS iterations is achieved.

2.11 Reinforcement Learning

In addition to tree search, reinforcement learning [12] has also emerged as an effective approach to AI in games. Reinforcement learning is a term used to describe algorithms that learn to solve sequential decision problems under uncertainty from interaction with their environments. The sequential decision problems that RL solves are different from the multi-armed bandit problem. The multi-armed bandit has a fixed set of actions with stationary reward distributions. In RL, the set of actions available to the learner (called the 'agent') and their corresponding reward distributions depend upon the state of the environment. In order to maximize its sum of rewards over time, the agent must consider how the available actions affect the immediate reward as well as the reward that is available from subsequent states.

Reinforcement learning models discrete-time sequential decision problems in fully observable environments as Markov Decision Processes (MDPs) [31]. MDPs consist of an agent that interacts with its environment over discretized time steps (Figure 2.4). At each time step $t = 1, 2, 3, \ldots$, the agent observes the current state or configuration

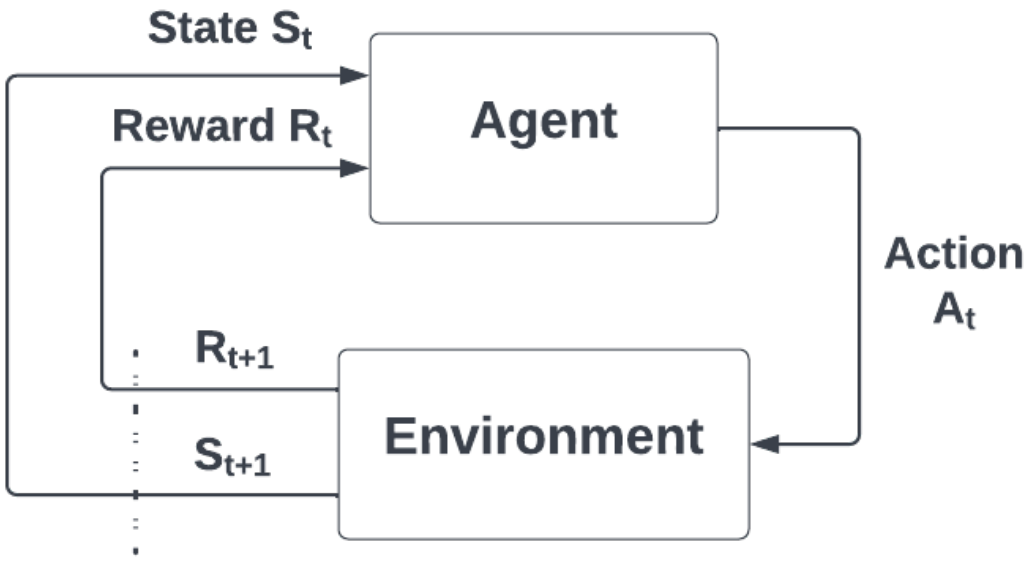


Figure 2.4: The interaction between an agent and its environment in an MDP.

of the environment $S_t \in \mathcal{S}$. Taking the current state S_t into account, the agent selects an action $A_t \in \mathcal{A}(S_t)$. As a consequence of its action, the environment responds by transitioning into state S_{t+1} and returning a reward $R_{t+1} \in \mathbb{R}$ to the agent at the next time step. When the set of states $\mathcal{S}$, state-dependent set of actions $\mathcal{A}(s)$, and set of rewards $\mathcal{R}$ are finite, all of the dynamics of the MDP can be expressed with the discrete probability distribution $p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$, for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The discrete probability distribution $p(s', r|s, a)$ is also known as a perfect model of the environment and is available to the agent in some RL problems.

Reinforcement learning agents learn how to solve predetermined tasks via the reward signal that is transmitted from the environment to the agent at each time step. In each RL problem, the agent's goal is to maximize its cumulative discounted sum of rewards over time. The return $G_t = R_{t+1} + \gamma R_{t+2} + \cdot + \gamma^{T-1} R_T$ is defined as the discounted sum of future rewards obtained after time step t. The discount factor

$0 \leq \gamma \leq 1$ indicates how the agent should balance immediate versus future reward. When γ is small, the agent prioritizes immediate reward. Time step T refers to the final time step of the sequential decision problem. In the context of board games, time step T represents a time step where the game transitions into a terminal state. Using this definition of the return, the agent's goal is formalized as maximizing the expected return $\mathbb{E}[G_t]$.

Reinforcement learning algorithms usually make use of value functions that estimate the expected return to be gained from being in a given state s or state-action pair (s, a). However, the expected return to be gained depends upon the actions selected by the agent in each state. An agent's policy $\pi(a|s)$ defines a discrete probability distribution over the legal actions $\mathcal{A}(s)$ in a given state s. In other words, the policy π defines the agent's behaviour or strategy. Reinforcement learning algorithms define how the agent's policy π changes as a function of the observed $(S_t, A_t, R_{t+1}, S_{t+1})$ tuples. An agent's value function under policy π at state s is $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ for all $s \in \mathcal{S}$. At terminal states, the value function evaluates to 0 because there are no future rewards to be gained. The action value function $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ represents the expected return to be gained from taking action a in state s and then subsequently following policy π. In problems with large state spaces, it is often necessary to represent the value functions as parameterized functions. The approximate value function $\hat{v}_\pi(s, \boldsymbol{\theta})$ estimates $v_\pi(s)$ and the approximate action value function $\hat{q}_\pi(s, a, \boldsymbol{\theta})$ estimates $q_\pi(s, a)$. The parameters $\boldsymbol{\theta}$ of the approximate value functions are updated in response to the observed rewards.

Solving reinforcement learning problems involves finding policies that maximize the expected return in each state. Value functions can be used to determine such policies. A policy π is said to be better than or equal to a policy π' if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Thus, an optimal policy π^* is better than or equal to all other policies. Value functions can be expressed in terms of the optimal policy as follows:

$$v_*(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S}$$

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$$

The last equation demonstrates how the optimal policy can be directly determined from the optimal action value function $q_*(s, a)$. By greedily selecting the action a that maximizes $q_*(s, a)$ in each state s, the agent follows the optimal policy.

To obtain optimal policies, many RL algorithms perform a process called policy iteration. Policy iteration consists of sequential steps of policy evaluation and policy improvement. Policy evaluation seeks to make the value function consistent with the current policy. To achieve this, the agent traverses the state space and updates its value function at the encountered states using rewards observed under the current policy or expected returns computed using the current policy. The policy improvement theorem guarantees that an improved policy π' will be obtained if the agent greedily selects actions which maximize the updated action value function: $\pi'(s) = \text{argmax}_a q_\pi(s, a)$. In the case where the policy $\pi(a|s)$ is stochastic, policy improvement is achieved by placing increased probability on the actions that maximize $q_\pi(s, a)$.

2.11.1 Model-Based Reinforcement Learning

In model-based reinforcement learning, the agent makes use of a model that mimics the dynamics of the environment. Distribution models take a state-action pair as input and return a prediction of the probabilities of each possible next state and reward. Sample models also take a state-action pair as input but return a next state and reward sampled from the predicted underlying dynamics distribution of the environment. In some model-based RL problems, a perfect model of the environment is given to the agent. However, in other model-based RL problems, the agent learns

a model from its experienced interactions with the environment. In either setting, the agent uses the model for planning. Sutton and Barto define planning as "any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment" ([12] pg. 160). This creates the subtle distinction between planning, which updates value functions using simulated experience generated by a model, and learning, which updates value functions using real experience obtained from interactions with the environment. Planning agents typically perform significantly more planning updates than learning updates because generating simulated experience is usually much faster than interacting with the environment.

Planning algorithms vary by the type of model they use and in their applicability to certain problems. For example, Dynamic Programming (DP) algorithms [32, 33] utilize perfect distribution models and are only feasible in smaller MDPs. DP algorithms represent value functions as tables, with one entry for each state in the environment. Policy evaluation is achieved by making systematic sweeps of the state space and updating the value of each state using the Bellman equation for v_π as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Iterative policy evaluation converges to v_π in the limit so long as each state is updated an infinite number of times [34, 35]. However, in practice, iterative policy evaluation is terminated once the maximum change in value over all states falls below some threshold. Policy improvement is achieved by acting greedily with respect to the value function estimate v_k:

$$\pi'(s) = \arg\max_a q_\pi(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Policy iteration [33] interleaves sweeps of policy evaluation and policy improvement in order to produce successively improved policies that ultimately converge to the optimal policy in the limit.

Despite the convergence guarantees of policy iteration in the DP setting, it is not a very practical algorithm. Policy evaluation steps require multiple sweeps of the state space to reduce the changes in value estimates below some threshold. Furthermore, policy improvement steps require one sweep of the state space to identify the greedy action for each state. Value Iteration [32] is an algorithm that improves the efficiency of policy iteration by combining policy evaluation and policy improvement. Value Iteration uses the update rule:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

where the action a that maximizes the expression is saved for the improved policy. Nevertheless, Value Iteration still suffers from the same pitfalls of repeated exhaustive sweeps of the state space.

Asynchronous DP algorithms [35–37] address this problem by non-uniformly distributing state value updates over the state space. Systematic sweeps of the state space inherently allocate the same number of value updates to each state. This causes value updates to be distributed equally across states that are relevant and irrelevant to optimal policies. Furthermore, systematic sweeps do not sequence value updates in an order that maximizes the propagation of values. Asynchronous DP algorithms update state values in any order and can allocate more value updates to the most pertinent states in the MDP. This significantly improves sample efficiency in the search for improved policies. However, in order to converge to an optimal policy, DP algorithms must still update each state infinitely many times.

A popular asynchronous DP algorithm is Real-Time Dynamic Programming (RTDP) [38]. RTDP distributes Value Iteration updates along sampled on-policy trajectories. This enables the agent to more accurately evaluate the states arising under its current policy and to subsequently improve its action selection from these states. However, only updating states along on-policy trajectories causes the same set of states to be repeatedly updated. Once the values of these states converge, it leads to wasted

computation.

Dyna [39] is another popular planning framework used in RL problems in which a perfect model of the environment is unavailable. Dyna agents interact with their environment and use the real experience to update their value function and model. Dyna employs a tabular model that assumes deterministic transitions. When a state-action pair (S_t, A_t) is experienced in the environment, the resulting next state S_{t+1} and reward R_{t+1} are associated with (S_t, A_t) in the tabular model. For every environmental interaction, a Dyna agent typically performs multiple planning updates. Each planning update involves sampling an experienced transition $(S_t, A_t, R_{t+1}, S_{t+1})$ from the tabular model uniformly at random. This tuple provides enough information to perform a one-step bootstrapped planning update to the value function.

2.11.2 Search Control

Synchronous DP control algorithms and standard Dyna suffer from similar drawbacks. As previously mentioned, synchronous DP algorithms perform systematic sweeps of the state space, wasting computation on states that are irrelevant to optimal behaviour. Similarly, for its planning updates, Dyna samples experienced transitions from its tabular model uniformly at random. This procedure assigns equal importance to all experienced transitions, which can be an inefficient use of computational resources. Planning efficiency can be substantially improved if planning updates are sequenced and/or distributed amongst states in a more intelligent way. One way of improving the sample efficiency of planning is through *search control*. Sutton and Barto define the term *search control* as the process that selects the starting state or state-action pair for planning [12]. While search control can greatly improve sample efficiency, it remains an under-researched topic in the RL community.

A simple example of search control in the episodic, tabular RL setting is *exploring starts* [12]. Under exploring starts, a planning algorithm begins its simulated episodes from randomly sampled state-action pairs. Then, the remainder of the episode is

traditionally produced on-policy. This allows the planning algorithm to concentrate its updates on the relevant state-action pairs appearing under its current policy, while still updating every single state-action pair infinitely many times in the limit. This enables convergence to the optimal value function and policy. However, exploring starts suffer from a similar problem to the DP algorithms that employ exhaustive sweeps - they treat every state-action pair equally, causing sample inefficiency.

Most research conducted in search control has focused on using visited states or predecessor states for planning. Perhaps the most popular form of search control is Prioritized Sweeping [40, 41]. Prioritized Sweeping is commonly used in the tabular RL setting and assumes that bootstrapped planning updates are performed to update the value function. Prioritized Sweeping maintains a priority queue of state-action pairs that is sorted by the estimated changes in value to the given state-action pairs. When a planning update is performed, the highest priority state-action pair is popped from the queue and a bootstrapped update is performed at the given state-action pair. Given the assumption of a tabular value function and bootstrapping, when a state-action pair (S, A)'s value is updated, it only affects the values of the state-action pairs that transition into (S, A). Prioritized Sweeping takes this into account by adding the state-action pairs $(\bar{S}, \bar{A})$, which transition into (S, A), to the priority queue with their respective estimated changes in value. Prioritized Sweeping can, therefore, be seen as a "backward focusing" algorithm as it focuses its planning updates on the predecessors of states that have been updated. Prioritizing planning updates on state-action pairs with large estimated changes in value enables values to efficiently propagate and for stronger policies to be efficiently learned. Prioritized Sweeping has been extended to linear function approximation [42] but no extensions to non-linear function approximation currently exist.

In the non-linear function approximation setting, search control algorithms often fall into two categories. The first category performs planning updates at states along hill climbing trajectories of the value function. This approach was first introduced by

Pan et al. [43] and was later extended to focus planning updates at states in high-frequency regions of the value function [44]. Performing planning updates along these trajectories effectively propagates value from high-value regions and concentrates updates in the most relevant parts of the MDP. Concentrating planning updates in high-frequency regions of the value function improves sample efficiency because high frequency regions of functions require more samples to accurately estimate.

The second category of search control algorithms in the non-linear function approximation setting take inspiration from *experience replay* [45, 46]. Experience replay maintains a circular buffer of the most recently experienced transitions. During training, batches of transitions are uniformly sampled from the experience replay buffer and are used to update the agent's value function. Experience replay helps improve sample efficiency because it enables an agent to train upon infrequently visited transitions multiple times. Experience replay also helps stabilize the learning process. The stochastic gradient descent (SGD) algorithms often used to optimize a neural network's parameters assume that the data trained upon is i.i.d [46–48]. Training upon data that is not i.i.d can slow or destabilize the learning process. In reinforcement learning, states arising within a given trajectory are highly correlated, and therefore, violate this i.i.d assumption. Uniformly sampling batches of transitions from the experience replay buffer enables the value function to be trained upon data that is more i.i.d, stabilizing the learning process.

Tavakoli et al. took inspiration from experience replay when developing *exploring restart distributions* [14]. Exploring restart distributions maintain a *restart memory* of visited states and combine it with the environment's initial state distribution to form the starting state distribution in the simulated environment. The initial state of a simulated episode is sampled from this exploring restart distribution. Tavakoli et al. experimented with three different versions of exploring restart distributions. Of relevance are *Uniform Restart*, which involves uniformly sampling the initial state of an episode from the circular restart memory and *Prioritized Restart*, which uses the

TD error of experienced transitions to assign priorities to experienced states in the restart memory.

Go-Explore [13] can also be interpreted as a search control algorithm inspired by experience replay. Go-Explore addresses two shortcomings experienced by RL algorithms in challenging exploration environments. The first shortcoming, called detachment, refers to agents forgetting how to return to previously visited promising states. The second shortcoming, known as derailment, refers to an agent's exploration mechanisms interfering with its ability to return to a promising state. Go-Explore addresses these problems by maintaining an archive of previously visited states weighted by the scores of their associated trajectories. At the beginning of each episode, Go-Explore samples a state from its archive, loads it into its simulator, takes exploratory actions from this state to identify higher scoring trajectories, and adds the newly encountered states to its archive. The archive prevents Go-Explore from forgetting about previously visited states and loading sampled states in the simulator allows Go-Explore to reliably return to and explore from promising states. Go-Explore then trains a policy either by learning from demonstrations [49] or via self-imitation learning [50] on the highest scoring trajectories. With this approach, Go-Explore achieved state of the art results in single agent Atari games [51] known to be challenging exploration problems.

Chapter 3

AlphaZero

AlphaZero [11] has established itself as the benchmark algorithm in the domain of two-player zero-sum games. Knowing only the rules of a game and beginning from random play, AlphaZero has demonstrated the ability to achieve superhuman play in chess, shogi, and Go using the same general algorithm with only hyperparamter tuning. In this chapter, I begin by giving a technical overview of the AlphaZero algorithm. First, I present the neural network that is used to represent learned game knowledge. Then, I describe the look-ahead search that is performed each turn and explain how the neural network guides the search. Afterward, I discuss how training on the improved policies returned by search and on the self-play trajectories produced under these policies enables AlphaZero to improve its play via policy iteration. Once the technical overview is complete, I highlight how exploration is incorporated into AlphaZero and discuss the limitations of AlphaZero's training procedure.

3.1 Neural Network

AlphaZero represents its learned game knowledge with a deep residual neural network consisting of convolutional layers. AlphaZero uses this neural network to bias its look-ahead search towards promising actions and to evaluate states in the search tree. Using a neural network as the search's evaluation function enables AlphaZero to be easily applied to many games. The parameters that make up AlphaZero's neural

network are not game-specific and can be trained to represent complex non-linear relationships of the input features. As long as the board of a two-player zero-sum game has a grid shape, AlphaZero can take the same steps to produce a feature vector and perform inference.

AlphaZero's neural network $f_{\boldsymbol{\theta}}(s) = (\boldsymbol{p}, v)$ is modeled after the deep convolutional neural networks that are state of the art in image recognition. The input to the neural network is a feature vector $\mathbf{x}$ representing a game state s and enough history to capture the full observability of the state. States are represented as an $N \times N \times (2MT + L)$ stack of gridded binary feature planes. Each feature plane is an $N \times N$ grid that mirrors the $N \times N$ game board. The first $2MT$ feature planes describe the positions of each of the M types of pieces over the T previous time steps for the two players. The final L feature planes represent additional information such as which colour's turn it is. This representation is convenient because the positions on the game board are easily mapped to the same positions on the feature planes. Furthermore, this gridded representation suits the gridded computation of the convolutional layers of the neural network.

AlphaZero's neural network $f_{\boldsymbol{\theta}}$ consists of a 'torso' that then branches into a 'policy head' and a 'value head' (Figure 3.1). The torso is composed of a convolutional block followed by multiple residual blocks. The convolutional block consists of a convolutional layer followed by batch normalization and a rectifier nonlinearity activation function. The residual blocks consist of two sequential convolutional blocks. However, a skip connection adds the input of the residual block to the output of the second convolutional block's batch normalization. The policy head is composed of a convolutional block followed by a fully connected layer that outputs a vector of logits over the actions. To produce the vector of action probabilities $\boldsymbol{p}$ over the legal actions, illegal moves are masked and then a Softmax is applied to the remaining vector of logit probabilities. The elements $p_a = \Pr(a|s)$ of $\boldsymbol{p}$ describe the probability of selecting action a from state s and estimate the policy π that would be returned by

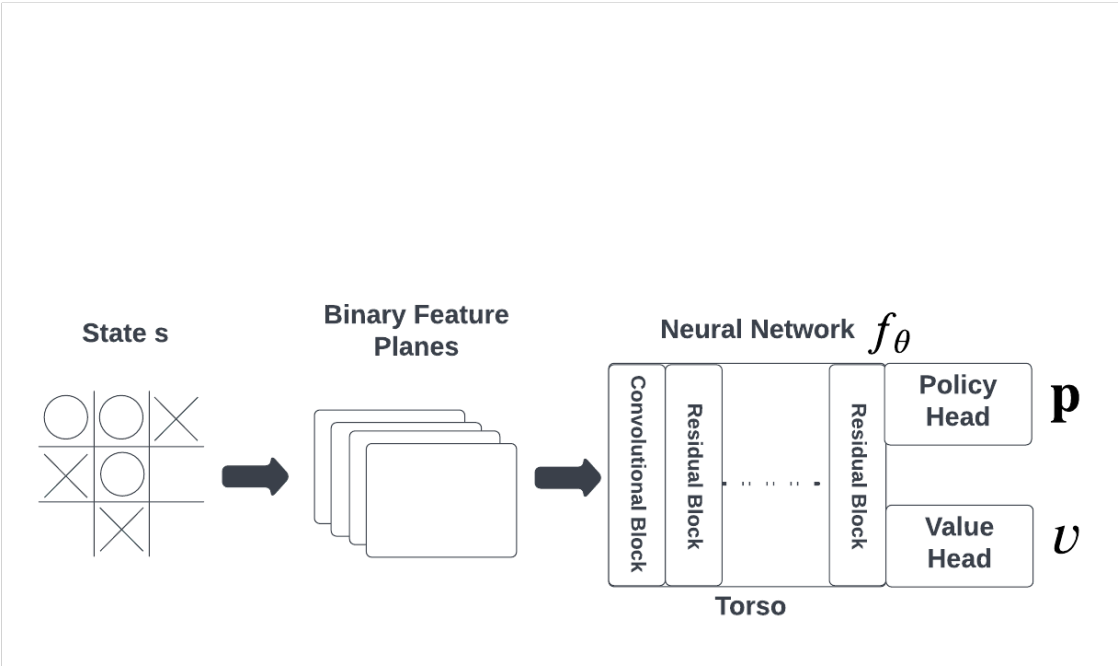


Figure 3.1: In AlphaZero, a game state s is transformed into a stack of binary feature planes prior to being input into the neural network $f_{\boldsymbol{\theta}}$. $f_{\boldsymbol{\theta}}$ outputs a vector of action probabilities $\boldsymbol{p}$ from its policy head and a value estimate v for input state s.

search at the input state s. The value head consists of a convolutional block followed by a fully connected layer, a rectifier nonlinearity activation function, another fully connected layer, and a tanh activation function which outputs a scalar v in the range $[-1, 1]$. The value estimate v estimates the current player's expected outcome of the game from state s under AlphaZero's current policy.

3.2 Search

AlphaZero is a Monte Carlo planning algorithm that progressively improves its play by training upon complete simulated self-play matches produced on-policy. These self-play matches begin from the initial state of a game s_0. On each turn t, AlphaZero conducts a search from the current state s_t in order to determine the action a_t that is played. AlphaZero employs a variant of Monte Carlo Tree Search. Nodes in the search tree represent game states and each node is initialized with a set of edges

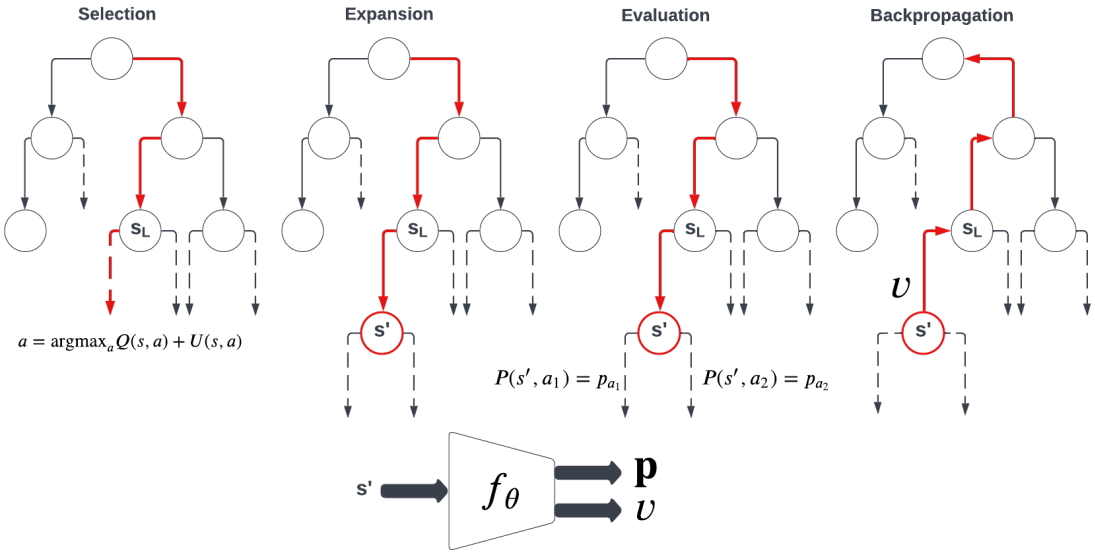


Figure 3.2: AlphaZero's variation of Monte Carlo Tree Search

corresponding to the legal actions that can be taken from the given state. Each edge stores a set of statistics $\{N(s,a), Q(s,a), P(s,a)\}$ that factor into the traversal of the search tree. $N(s,a)$ 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.

When AlphaZero begins a search, it initializes the search tree with a root node corresponding to the current game state s_t. Each search iteration, the search tree is traversed from the root node using the action selection rule:

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

where $N(s)$ is the number of times state s has been visited during the search and $c_{\text{pucb}} > 0$ is an exploration constant (Selection step in Figure 3.2). Similarly to UCT, this action selection rule forms an upper confidence bound on the action value of a state-action pair. This action selection rule also takes inspiration from PUCB by incorporating predictions that bias the action selection. In the original PUCB algorithm, a context is mapped to a vector of weights $\boldsymbol{M} : \sum_i M_i = 1$. The contextual weights M_i are used to form additive penalties that bias the action selection towards

actions with larger contextual weights. In AlphaZero, the neural network $f_{\boldsymbol{\theta}}$ is used as a predictor that biases action selection during search. The components p_a of $\boldsymbol{p}$ correspond to the weights M_i but are included in the exploration bonus term rather than a separate term. 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 $N(s,a)$, with c_{pucb} controlling the level of exploration vs. exploitation.

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: $f_\theta(s') = (\mathbf{p}, v)$ (Expansion and Evaluation steps in Figure 3.2). 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. Afterward, the value estimate v is backed up to the state-action pairs that were traversed in the given iteration to update their action values $Q(s,a)$ (Backpropagation step in Figure 3.2). Their respective edge statistics are updated as follows: $\{N(s,a) = N(s,a) + 1, Q(s,a) = Q(s,a) + \frac{1}{N(s,a)}(v - Q(s,a))\}$. $Q(s,a)$ averages the value estimates v of the states in the subtree of (s,a) and 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 that is used to select the action a_t that is played from state s_t. The components of π_t are proportional to the search visits over the root state's actions: $\pi_t(a|s_t) = \frac{N(s_t,a)^{\frac{1}{\tau}}}{\sum_b N(s_t,b)^{\frac{1}{\tau}}}$. The temperature hyperparameter $\tau > 0$ helps control the level of exploration vs. exploitation in the produced policies π_t. 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.

3.3 Policy Iteration

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 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. For a given training sample (s_t, π_t, z), the neural network uses the policy π_t as a learning target for the policy head and the self-play match outcome z as a learning target for the value head at the input s_t. 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. The first term of the loss function is the squared error between the self-play game outcome z and the neural network's value estimate v at state s_t. The second term in the loss function is the cross-entropy loss between the policy $\boldsymbol{\pi}_t$ and the neural network's action probabilities $\boldsymbol{p}$ at state s_t. The third term in the loss function is a regularization term that ensures that the neural network's parameters $\boldsymbol{\theta}$ do not get too big to overfit the training data. 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 over time. 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. Upon the completion of search, AlphaZero selects an action a_t with respect to the improved policy π_t. Training f_θ's value head on self-play match 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 progressively stronger policies. 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.

3.4 Exploration In AlphaZero

In order for AlphaZero to produce improved policies π_t, the value estimates used to guide search need to be accurate under the current policy. 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 of the state space by introducing stochasticity into its action selection. In the following subsections, I describe the exploration mechanisms AlphaZero uses to diversify its training and then discuss their limitations.

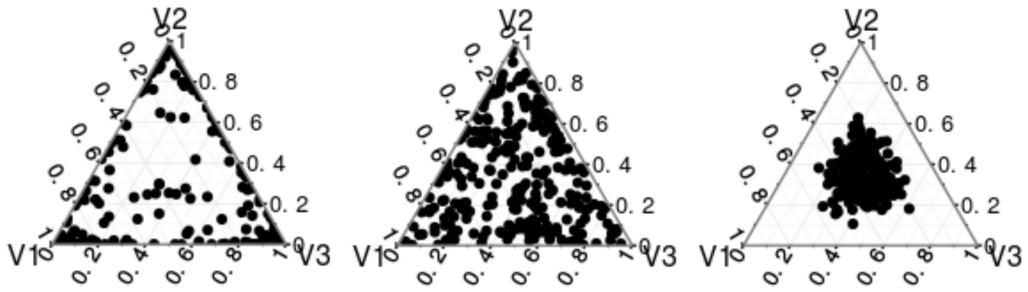
3.4.1 Dirichlet Noise

In its search, AlphaZero perturbs the priors over the root node's actions with noise sampled from a Dirichlet distribution. At the beginning of each search, the root node s_{root} is evaluated by the neural network $f_{\boldsymbol{\theta}}(s_{\text{root}}) = (\boldsymbol{p}, v)$ and the vector of action probabilities $\boldsymbol{p}$ is perturbed by a noise vector $\boldsymbol{d} \sim \text{Dir}(\boldsymbol{\alpha})$ sampled from a Dirichlet distribution. The perturbed priors $P(s_{\text{root}}, a)$ over the root node's actions are computed using the equation

$$P(s_{\text{root}}, a) = (1 - \epsilon)p_a + \epsilon d_a$$

where p_a and d_a are the components of $\boldsymbol{p}$ and $\boldsymbol{d}$, respectively, and $0 < \epsilon < 1$ is a constant controlling the magnitude of the noise. The perturbed priors $P(s_{\text{root}}, a)$ play a prominent role in the action selection from the root node during search. Randomly perturbing the priors over the root node's actions causes the distribution π_t of search visits over the root node's actions to also be perturbed, introducing randomness into AlphaZero's action selection during training.

The Dirichlet distribution $\text{Dir}(\boldsymbol{\alpha}), \boldsymbol{\alpha} \in \mathbb{R}^l$ is a distribution over l-dimensional discrete probability distributions. When sampled, the Dirichlet distribution returns an l-dimensional vector $\boldsymbol{d}$ whose components form a discrete probability distribution. In



(a) $\boldsymbol{\alpha} = 0.2$ (b) $\boldsymbol{\alpha} = 1.0$ (c) $\boldsymbol{\alpha} = 10.0$

Figure 3.3: The 3-dimensional Dirichlet distribution over different values of $\boldsymbol{\alpha}$

.

the context of AlphaZero, l is set to the number of legal actions that can be taken from the root node so that the dimension of the sampled noise vector $\boldsymbol{d}$ matches the dimension of the vector of action probabilities $\boldsymbol{p}$. The vector of positive real numbers $\boldsymbol{\alpha}$ that parameterizes the Dirichlet distribution affects the shape of the discrete probability distributions sampled (Figure 3.3). For example, in AlphaZero, each component of $\boldsymbol{\alpha}$ is given the same value which means that the probability density of the Dirichlet distribution is not biased toward any particular action. When $\boldsymbol{\alpha} < 1$, the noise vectors sampled concentrate most of the probability on one of the actions (Figure 3.3a). When $\boldsymbol{\alpha} = 1$, each of the possible noise vectors is sampled with equal probability (Figure 3.3b). When $\boldsymbol{\alpha} > 1$, the sampled noise vectors are more uniform (Figure 3.3c). In the published AlphaZero paper, the authors used $\boldsymbol{\alpha} = 0.3, 0.15$, and 0.03 for chess, shogi, and Go respectively. Since $\boldsymbol{\alpha} < 1$, the noise vectors that are sampled concentrate most of the probability on one component, encouraging exploration toward a random action. The extent of this exploration is controlled by the value of ϵ, which was set to 0.25 in all three games.

3.4.2 c_{pucb}

The constant, c_{pucb}, used in the search's action selection rule impacts the level of exploration vs. exploitation in the returned policies. The primary role of the c_{pucb} constant is to balance exploration and exploitation in search. When c_{pucb} is relatively

small, the search favours the traversal of state-action pairs with large action values $Q(s,a)$. When c_{pucb} is relatively large, the search favours the traversal of state-action pairs with large priors $P(s,a)$ and few search visits $N(s,a)$. Balancing exploration and exploitation within AlphaZero's search is necessary for identifying promising actions. However, it also affects the distribution of search visits over the root state's actions, and thus, the produced policies π_t. When c_{pucb} is relatively small, a greater number of search visits pass through the root action with the greatest action value $Q(s,a)$. This causes AlphaZero to produce policies π_t that are more exploitative. When c_{pucb} is relatively large, the distribution of search visits over the root state's actions resembles the priors $P(s,a)$ over the root state's actions. Early on in training, the priors tend to be more uniform, causing AlphaZero to produce policies π_t that are more exploratory.

3.4.3 Action Sampling

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 $\pi_t(a|s_t) = \frac{N(s_t,a)^{\frac{1}{\tau}}}{\sum_b N(s_t,b)^{\frac{1}{\tau}}}$. 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 $P(s,a)$.

3.4.4 Exploration-Exploitation Trade-Off

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 c_{pucb} constant used in search, the temperature parameter 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.

Chapter 4

Go-Exploit

In the previous chapter, I identified three primary limitations in AlphaZero's training procedure:

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 k 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.

AlphaZero's struggle to visit and revisit states deeper in the game tree is reminiscent of *detachment* and *derailment*, earlier introduced in the context of Go-Explore. When AlphaZero visits and trains upon a promising state deeper in the game tree, it does not guarantee that AlphaZero will learn a policy that leads it back to that state for further exploration (detachment). Furthermore, the Dirichlet noise and action sampling interfere with AlphaZero's ability to return to and further explore from promising states (derailment).

Given these limitations, I adopted the following guiding principles in designing a new training strategy for AlphaZero. 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.

I hypothesized that I could implement these guiding principles in AlphaZero and improve its sample efficiency with a novel search control strategy. Since AlphaZero is a planning algorithm that trains upon simulated self-play matches, it can intelligently select the starting state of its simulated experience. I took inspiration from Go-Explore [13] and Exploring Restart Distributions [14] by incorporating an archive of states of interest in AlphaZero. My algorithm, called Go-Exploit, modifies AlphaZero by beginning its 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, I 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 thesis, I explore this approach and

experiment with three archive structures and two definitions of "states of interest" to see how they respectively impact the sample efficiency of Go-Exploit.

4.1 Archive Types

The structure of the archive is characterized by its capacity and the mechanism for adding/removing states of interest. The simplest archive is the *Expanding Archive* which contains every observed state of interest. Fixed-size archives are necessary when there is insufficient memory to store all encountered states of interest during training. This issue presents itself in larger games that require more training iterations or in variants of Go-Exploit that have less restrictive definitions for states of interest. When a fixed-size archive reaches its capacity $|A|$, there must be a process for determining which states of interest should be inserted and removed from the archive. In this thesis, I experiment with two different fixed-size archives. The *Circular Archive* removes the oldest state of interest to make way for the newest state of interest. The Circular Archive enables Go-Exploit to reliably return to states encountered under the most recent policies and to improve action selection from these states. The *Reservoir Archive* uses Reservoir Sampling [52] to determine whether a new state of interest replaces an older state in the archive. Reservoir Sampling is an algorithm used to produce a random sample of elements from an unknown population observed one at a time. Once the archive is at capacity, the Reservoir Archive uniformly samples an integer i between 1 and n, where n is the number of states of interest that have been observed. If $i < |A|$, the new state of interest replaces the ith entry in the archive. The Reservoir Archive approximates the distribution of states that would be stored in an Expanding Archive.

4.2 States of Interest

The way "states of interest" are defined naturally 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. In this thesis, I introduce four variants of Go-Exploit that use these two definitions of "states of interest". The first two variants fall under Go-Exploit Visited States and the remaining two variants fall under Go-Exploit Search States. 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. Pseudocode for the different variants of Go-Exploit can be found in Algorithms 2, 3, 4, 5, 6.

4.3 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 (Algorithm 2 line 3). This archive is shared amongst "training actors" that generate self-play trajectories (Algorithm 3). At the beginning of each self-play trajectory, Go-Exploit uniformly generates 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 sampled from the archive uniformly at random (Algorithm 3 lines 4-7). The second difference is that 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 (Algorithm 3 lines 11-12). Finally, once a self-play trajectory completes, Go-Exploit Visited States adds the nonterminal

states that were visited to the archive A (Algorithm 5 lines 10, 15-16). In this thesis, I experimented with two variants of Go-Exploit Visited States that used two different archives. Go-Exploit Visited States Expanding Archive (GEVE) uses an Expanding Archive consisting of every single visited state. Go-Exploit Visited States Circular Archive (GEVC) employs a fixed-size Circular Archive consisting of the most recently visited states.

4.4 Go-Exploit Search States

Go-Exploit Search States makes similar modifications to AlphaZero. It also employs training actors that sample start states from archive A and produce training data for f_θ (Algorithm 3). However, Go-Exploit Search States' training actors do not add visited states or search states to the archive (Algorithm 5 lines 15-16). Go-Exploit Search States, instead, concurrently runs "archive actors" that are responsible for populating the archive (Algorithm 4). The archive actors always play out complete self-play matches beginning from s_0 (Algorithm 4 line 4). Once an archive actor's self-play match is complete, it adds all of the nonterminal states that appeared during search into archive A (Algorithm 4 lines 6, 12, 16-17). In this thesis, I experimented with two variants of Go-Exploit Search States. Go-Exploit Search States Reservoir Archive (GESR) uses a fixed-size archive and Reservoir Sampling [52] to determine which states are added/removed from the archive. A Reservoir Archive is used instead of an Expanding Archive due to the sheer number of search states observed during training. Go-Exploit Search States Circular Archive (GESC) employs a fixed-size Circular Archive consisting of the most recently observed search states.

4.5 Related Work

Although Go-Exploit is inspired by Go-Explore [13], the two algorithms work very differently. In Go-Explore, the "Go" step is exploitative because it loads a state

associated with a high scoring trajectory. Exploratory actions are taken from this loaded state to discover higher scoring trajectories that are then used to train a policy. 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 state, Go-Exploit can then produce the remainder of its self-play trajectories under more exploitative policies. Hence the name Go-Exploit.

Go-Exploit is more aligned with the work introduced in Exploring Restart Distributions [14] but applies it in the new setting of AlphaZero. Go-Exploit Visited States is analogous to *Uniform Restart* because they both begin their simulated episodes from previously visited states sampled from a memory. However, in two-player, zero-sum games, beginning self-play trajectories from previously visited states may not result in the most efficient learning. Go-Exploit Search States extends Exploring Restart Distributions beyond visited states. Go-Exploit deliberately uses the notion of "states of interest" when defining which states can be included in its archive in order 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 [53], 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, unlike Go-Exploit, MuZero Reanalyze does not simulate new self-play trajectories from these previously visited states. While planning from previously visited states improves MuZero's sample efficiency, its sample efficiency could potentially be further improved by simulating new self-play trajectories from these previously visited states

and training upon the states explored in these trajectories.

Other than MuZero, KataGo [15] is the only other algorithm I am aware of that incorporates search control in the setting of two-player zero-sum games. I will describe KataGo's search control procedure in Chapter 5 and then evaluate its sample efficiency relative to Go-Exploit.

Chapter 5

Experiments

Experiments were conducted in Connect Four and 9x9 Go to evaluate the sample efficiencies of the four variants of Go-Exploit and to understand how Go-Exploit's exploration of the state space differs from AlphaZero's. By performing experiments in Connect Four and 9x9 Go, I evaluated Go-Exploit in two domains with different characteristics. Connect Four has a smaller search space than 9x9 Go but has a greater percentage of terminal states throughout its game tree. In this chapter, I begin by detailing the experimental setup that was used in each experiment. Then, I present a few definitions of sample efficiency and discuss which one was best suited for my experiments. Afterward, I describe the experiments that were performed to measure and compare the sample efficiencies of Go-Exploit and AlphaZero. Next, I introduce KataGo [15], a reimplementation of AlphaZero that improved AlphaZero's sample efficiency, and describe experiments that were performed to evaluate KataGo's search control procedure relative to Go-Exploit's. I also share experiments that were conducted to assess the compatibility of KataGo's other innovations with Go-Exploit. Finally, I describe experiments that were performed to measure Go-Exploit's adherence to the guiding principles relative to AlphaZero.

5.1 Experimental Setup

Go-Exploit and certain elements of KataGo were coded on top of DeepMind's OpenSpiel [54] implementation of AlphaZero. OpenSpiel is an open source repository consisting of environments and algorithms meant for research in search and reinforcement learning in games. Most of OpenSpiel's environments and algorithms are offered in Python and C++ and some algorithms even have separate TensorFlow and Pytorch/Libtorch implementations. Go-Exploit and certain elements of KataGo were coded on top of OpenSpiel's C++/Libtorch version of AlphaZero in order to take advantage of C++'s faster execution. Experiments were run using OpenSpiel's accompanying versions of Connect Four and Go. All Connect Four experiments were run on Compute Canada's Cedar cluster using one compute node consisting of 32 cores (2 x Intel Silver 4216 Cascade Lake @ 2.1GHz), 187G of memory, and 4 NVIDIA V100 GPUs. All 9x9 Go experiments were run on Compute Canada's Beluga cluster using one compute node consisting of 40 cores (2 x Intel Gold 6148 Skylake @ 2.4 GHz), 186G of memory, and 4 NVIDIA V100 GPUs.

5.2 Sample Efficiency

Sample efficiency can be measured in many different ways. For example, it can be measured by the number of training samples required to attain a target performance level. Alternatively, sample efficiency can be measured by the average performance level achieved over a computational budget. Sample efficiency can also be measured by the performance level achieved in the final learning step. However, this metric suffers from two major drawbacks. First, this metric poorly differentiates between algorithms that converge to similar asymptotic performance levels. Logically, a greater sample efficiency is achieved by the algorithm that converges to this asymptotic performance level more quickly. Second, this metric is sensitive to the selected training horizon. The strength of AlphaZero's learned policies generally increases with training time,

however, it does not do so monotonically. The oscillations in AlphaZero's performance level from learning step to learning step can lead to the selection/preference of an overall inferior hyperparameter value or algorithm.

Given these considerations, I measured an algorithm's sample efficiency by calculating its average performance level over a computational budget. In my experiments, training runs lasted 600 learning steps in Connect Four and 900 learning steps in 9x9 Go. These training horizons were arbitrarily chosen but established a fixed computational budget for each training run. One way of representing the average performance level over a computational budget is with the "area under the curve" (AUC) in a performance level vs. learning step graph. Depending on the training horizon, this metric favours different learning characteristics. If the training horizon is relatively short, the AUC favours algorithms that are able to quickly achieve good performance levels. If the training horizon is relatively long, the AUC favours algorithms that achieve and sustain the greatest performance levels. The AUC is able to differentiate between algorithms achieving similar asymptotic performance levels because it takes into account how quickly these algorithms attain this performance level. Furthermore, the AUC is less sensitive to the training horizon because it factors in an algorithm's performance level over the entire computational budget rather than at an arbitrarily chosen final learning step.

In my experiments, an algorithm's performance level was measured by its win rate against a fixed reference opponent. In each training run, 50 evaluator threads played evaluation matches against the fixed reference opponent, MCTS-Solver [55], over the course of training. MCTS-Solver is a variant of MCTS that proves the game-theoretical value of states in the search tree, allowing it to concentrate its search iterations on unproven parts of the search tree. Evaluation matches were played against different difficulty levels of MCTS-Solver with 1x, 10x, 100x, and 1000x as many search iterations as AlphaZero, Go-Exploit, and KataGo. An equal number 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.

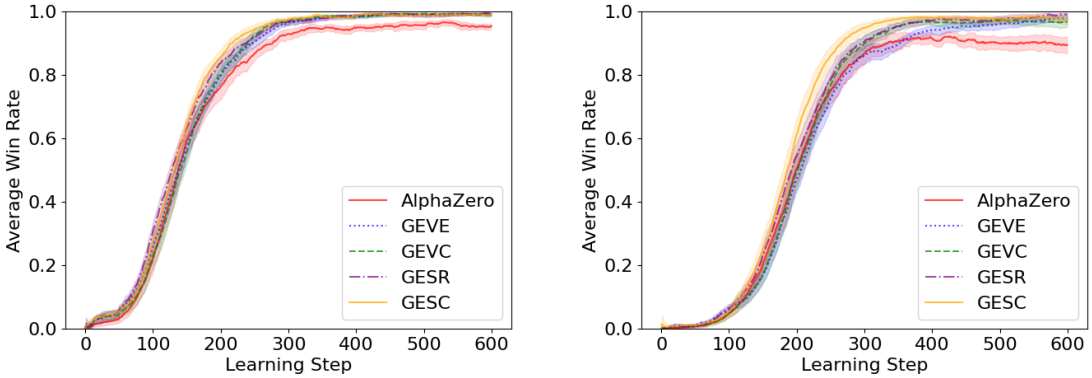
When hyperparameter sweeps were conducted, 10 independent training runs were executed for each hyperparameter setting. Upon their completion, their win rates against each difficulty level of MCTS-Solver were averaged at each learning step. The average win rates were summed over all learning steps to 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 average win rates against MCTS-Solver 1x, 10x, 100x, and 1000x were computed to see how the algorithms performed against different fixed reference opponents.

5.3 Go-Exploit vs. AlphaZero

To compare the sample efficiencies of the four variants of Go-Exploit relative to AlphaZero, I first performed hyperparameter sweeps to identify the best performing values for each algorithm in both Connect Four and 9x9 Go. The hyperparameters that were held fixed do not directly affect the distribution of states encountered and trained upon, and thus, are not pertinent to my main investigation. These include hyperparameters affecting the architecture of f_θ, the number of threads, and batch sizes. The full list of fixed hyperparameters can be found in Table C.1.

The hyperparameters swept over were 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_{pucb} in search, the number of action sampling moves k, the probability λ of beginning self-play trajectories from s_0, the archive size $|A|$, and the action sampling temperature τ. Since there were so many hyperparameters to sweep over, a grid search was infeasible.



(a) MCTS-Solver 10x

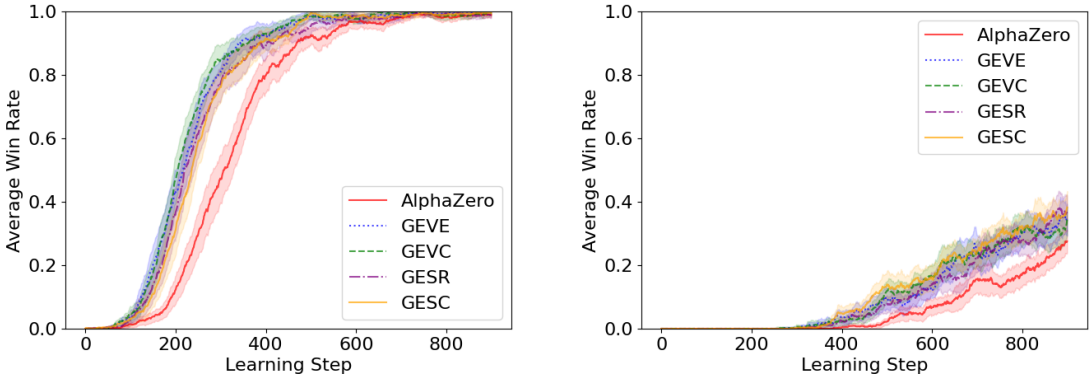
(b) MCTS-Solver 1000x

Figure 5.1: AlphaZero and Go-Exploit's win rates against MCTS-Solver 10x and 1000x in Connect Four. The win rates were averaged over the 30 validation runs and the shaded regions represent 95% confidence intervals.

.

Instead, I swept over one hyperparameter at a time. In each sweep, the hyperparameters were swept in the order listed above (lr first and τ last). Prior to being swept over, the hyperparameters were set to the bolded values in Tables C.2 and C.4 for Connect Four and 9x9 Go, respectively. 10 independent runs were executed for each hyperparameter value with randomly chosen seeds. Once a hyperparameter was swept over, it was set to the best performing value for the remainder of the sweep. The hyperparameter values swept over and the best performing hyperparameter values appear in Tables C.2, C.3, C.4, and C.5.

Once the best performing hyperparameter values were identified, an additional 30 validation runs were executed for each algorithm. The average win rates against each difficulty level of MCTS-Solver were used to produce the learning curves appearing in Figures 5.1 and 5.2 with shaded 95% confidence intervals. In Figures 5.1a and 5.1b, 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 speeds, but as training progresses, AlphaZero's learning curve levels out to a lower asymptotic win rate than the four variants of Go-Exploit. It should be noted that during the hyperparameter sweeps, I



(a) MCTS-Solver 10x (b) MCTS-Solver 1000x

Figure 5.2: AlphaZero and Go-Exploit's win rates against MCTS-Solver 10x and 1000x in 9x9 Go. The win rates were averaged over the 30 validation runs and the shaded regions represent 95% confidence intervals.

.

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 5.1a and 5.1b 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 that focuses training on the states observed under the most recent policies.

In Figures 5.2a and 5.2b, 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 5.2a 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 Figures 5.1 and 5.2 also reveals that Go-Exploit's gain in

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

Table 5.1: Algorithm 1's win rates in head-to-head matches

sample efficiency is much greater in 9x9 Go than in Connect Four. This suggests that Go-Exploit's gains in sample efficiency may be even greater in larger games. This may be due to the fact that when the search space increases in size, AlphaZero wastes even more samples to reach and train upon new states deeper in the game tree.

To further measure Go-Exploit's learning efficiency relative to AlphaZero, I conducted head-to-head matches between AlphaZero and each variant of Go-Exploit. In Connect Four, head-to-head matches were played using the saved neural network checkpoints from learning steps 300 and 600 in the validation runs. In 9x9 Go, head-to-head matches were played using the saved neural network checkpoints from learning steps 300 and 900 in the validation runs. Each algorithm's 30 neural network checkpoints played the other algorithms' 30 neural network checkpoints in one game as player 1 and in another game as player 2. The results of the Connect Four and 9x9 Go tournaments appear in Table 5.1. The listed win rates are from the perspective

of Algorithm 1.

The results of the Connect Four and 9x9 Go tournaments reaffirm what was observed in Figures 5.1 and 5.2. 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 5.1a and 5.1b. At checkpoint 600, each variant of Go-Exploit outperformed AlphaZero but none of the variants 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 5.1a and 5.1b. In the 9x9 Go tournament, each variant of Go-Exploit dominated AlphaZero at checkpoints 300 and 900, although Go-Exploit won by a greater margin at checkpoint 300. This is consistent with Go-Exploit's superior win rate against MCTS-Solver in Figures 5.2a and 5.2b. 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 5.2a.

5.4 Go-Exploit vs. KataGo

KataGo [15] is an open-source reimplementation of AlphaZero that introduces multiple modifications to the original AlphaZero algorithm that improve its sample efficiency. In this section, I compare Go-Exploit to the search control procedures introduced in KataGo. Then, I argue that the remainder of the modifications introduced in KataGo are compatible with Go-Exploit and can be combined with it to achieve even greater learning efficiency.

5.4.1 KataGo's Search Control Strategy

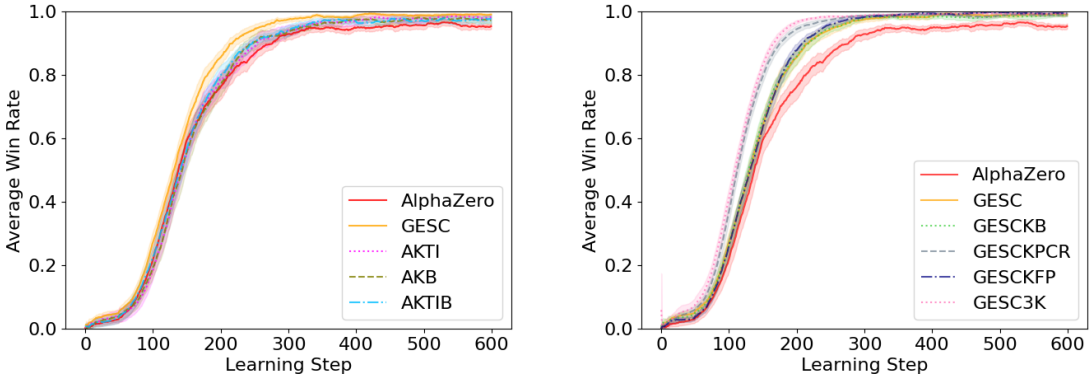
In the original KataGo paper and its subsequent release notes, key modifications to AlphaZero are highlighted. An additional change introduced in KataGo, which is not emphasized, is a search control strategy. KataGo's search control procedure involves self-play trajectory initialization and position branching. At the beginning

of each self-play trajectory, KataGo samples a random number r from an exponential distribution with $\mu = 0.04 \times \text{board\_width} \times \text{board\_length}$. KataGo then samples the first $\lfloor r \rfloor$ actions of the game from the policies $\mathbf{p}$ output by f_θ to determine the start state of the self-play trajectory. In 2.5% of visited positions, KataGo branches the self-play trajectory to try a different action from the one that was originally selected. Afterward, KataGo performs a search from the new branched position and uses the returned policy π as a training target for the policy head and the root node's action value as a learning target for the value head. A random quarter of these branches are recursively continued. KataGo also branches from an early position in 5% of self-play trajectories. To select the branched action, KataGo samples 3 to 10 moves uniformly at random, evaluates their resulting board positions with f_θ, and then selects the action with the greatest value estimate. The remainder of the branched trajectory is played out normally.

To compare the effectiveness of Go-Exploit's search control procedure to KataGo's, I tried running OpenSpiel's AlphaZero implementation with KataGo's search control procedure in Connect Four. I first tried running AlphaZero with KataGo's trajectory initialization (AKTI), then with KataGo's branching scheme (AKB), and then with both together (AKTIB). A new hyperparameter sweep was conducted to determine the hyperparameter values that achieve the best AUCs for each algorithm. Afterward, an additional 30 validation runs were performed. The learning curves for AlphaZero, AlphaZero with KataGo's search control procedures, and GESC appear in Figure 5.3a. 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.

5.4.2 Go-Exploit's Compatibility With KataGo

Excluding KataGo's trajectory initialization, KataGo's other modifications are compatible with Go-Exploit. KataGo's key modifications alter AlphaZero's neural network architecture, loss function, feature representation, and search, which are orthog-



(a) KataGo's search control strategy (b) Go-Exploit + KataGo's modifications

Figure 5.3: (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.

onal to Go-Exploit. In order to provide evidence that KataGo's other modifications are complementary with Go-Exploit, I incorporated three of KataGo's modifications into GESC to see if they could help it achieve greater learning efficiency. The first modification I incorporated into GESC was KataGo's branching scheme. While Go-Exploit's trajectory initialization resembles branching, it is not equivalent to KataGo's branching procedure. The second KataGo modification I implemented was "Playout Cap Randomization". Playout Cap Randomization randomly varies between performing "fast searches" with fewer search iterations and "full searches" with a much greater number of search iterations. Only states where "full searches" are performed are used for training, which increases the number of independent value targets available for training. In my experiments, "fast searches" were conducted 75% of the time and performed 20 search iterations whereas "full searches" performed 100 search iterations. The third KataGo modification I implemented was "Forced Playouts and Policy Target Pruning". During search, KataGo forces visits to root actions that have already been traversed in order to possibly overcome small initial action value estimates. While the forced playouts help identify promising root actions, they also

inflate the visit counts to poor actions, hindering policy improvement. To overcome this, KataGo performs "Policy Target Pruning", which subtracts search visits from root actions that would not have otherwise been taken with normal PUCT action selection.

To test the compatibility of Branching (GESCKB), Playout Cap Randomization (GESCKPCR), and Forced Playouts + Policy Target Pruning (GESCKFP) with Go-Exploit, I ran GESC with each modification individually. Then, I ran GESC with all three modifications (GESC3K) to see if an even greater sample efficiency could be achieved. For each variant, I ran the standard hyperparameter sweep and additional 30 validation runs. Their respective learning curves appear in Figure 5.3b. This figure illustrates that GESC maintains a similar AUC when combined with branching. On the other hand, 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 of these modifications. While not definitive, this supports my argument that KataGo's modifications to AlphaZero, other than its search control procedure, are complementary with Go-Exploit.

5.5 Understanding Go-Exploit

To understand why Go-Exploit learns more efficiently than standard AlphaZero, I collected statistics on the distribution of states visited during self-play and observed during search in the validation runs. In the following subsections, I appeal to these collected statistics and the guiding principles to try to establish why Go-Exploit outperforms AlphaZero in both Connect Four and 9x9 Go.

5.5.1 Greater Exploration of the State Space

I 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

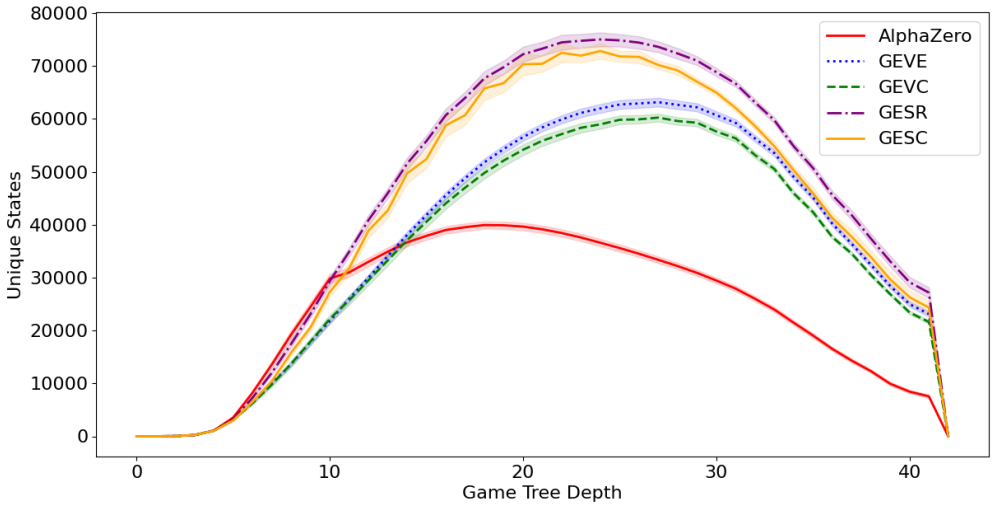


Figure 5.4: The aggregate number of unique states visited during self-play games, as a function of game tree depth, over 600 learning steps in Connect Four.

states deeper 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 5.4 and 5.5 depict each algorithm's distribution of unique nonterminal states visited by game tree depth in Connect Four and 9x9 Go, respectively. Computing the area under a curve yields the total number of unique nonterminal states visited by a given algorithm. In these plots, we observe that each variant of Go-Exploit visits a greater total number of unique nonterminal states than AlphaZero. Go-Exploit particularly visits a much greater number of unique nonterminal states deeper in the game tree than AlphaZero. At earlier game tree depths, we can see that AlphaZero visits more unique states than Go-Exploit in both Connect Four and 9x9 Go. 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 5.4, 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

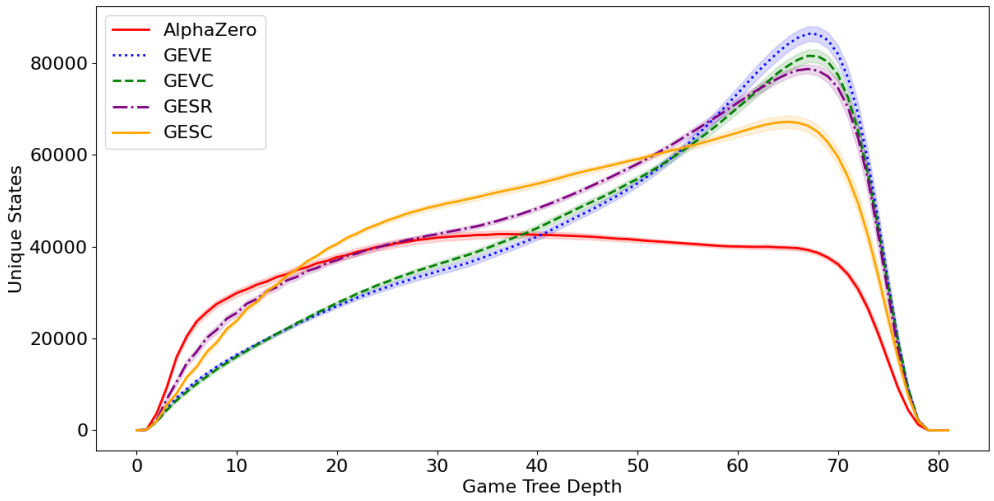


Figure 5.5: The aggregate number of unique states visited during self-play games, as a function of game tree depth, over 900 learning steps in 9x9 Go.

actions over the first k moves of a self-play match hinders AlphaZero's exploration. After game tree depth 18 in Connect Four, the number of unique nonterminal states visited by AlphaZero steadily decreases. This is due to the fact that in Connect Four, AlphaZero often transitions into a terminal state before reaching and exploring states deeper in the game tree. On the other hand, in Figure 5.5, we can see that the number of unique nonterminal states visited by AlphaZero remains fairly level over most game tree depths, reflecting the fact that there are fewer terminal states throughout 9x9 Go's game tree. Even though AlphaZero has more success exploring deeper in 9x9 Go's game tree than in Connect Four's, Go-Exploit outperforms AlphaZero by a greater margin in the larger game of 9x9 Go. 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,

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

Go-Exploit Search States visits a greater number of unique states than Go-Exploit Visited States 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, I compared their value losses over visited states and search states. To establish a fair comparison, I 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, I 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 5.2 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 illustrates that its value function can better predict match outcomes under its current policy and at a greater set of states than AlphaZero. I 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.

5.5.2 More Independent Value Targets

Another potential reason why Go-Exploit has a smaller value loss over visited states and search states than AlphaZero can be due to 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 trajectories per learning step, on average. Similarly, in 9x9 Go, AlphaZero completes an average of 74.83 trajectories per learning step whereas each variant

[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.

of Go-Exploit completes over 147 trajectories per learning step, 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 than AlphaZero (guiding principle (d)) may enable Go-Exploit to train a more accurate value function, enabling search to be a more effective policy improvement operator.

Producing shorter self-play trajectories also helps stabilize and speed up Go-Exploit's learning process. Both AlphaZero and Go-Exploit optimize their policy-value network's parameters using stochastic gradient descent (SGD). SGD assumes that the data used to estimate its gradients is independent and identically distributed (i.i.d) [48]. In AlphaZero and Go-Exploit, the data produced within a given self-play trajectory is highly correlated. Simultaneously training upon all the states from a given trajectory would violate the i.i.d assumption and could slow or destabilize the policy-value network's training. AlphaZero and Go-Exploit alleviate this problem by using experience replay. Under experience replay, experienced transitions from the most recent episodes are stored within a circular memory. Training batches are uniformly sampled from this memory, significantly reducing the dependence between samples within a training batch. Go-Exploit further reduces the dependence between training samples stored in the experience replay buffer. Since Go-Exploit's self-play trajectories are shorter, on average, than AlphaZero's, fewer correlated training samples are added to the experience replay buffer per self-play trajectory. The fact that Go-Exploit produces and trains upon data that is more i.i.d than AlphaZero may help stabilize and speed up its training.

5.5.3 Training Under More Exploitative Policies

In Figures 5.4 and 5.5, we observed that Go-Exploit is able to more effectively explore the game tree than AlphaZero since it begins its self-play trajectories from

states throughout the game tree. Since there is exploration inherent in the sampling of a start state from the archive, I hypothesized that Go-Exploit would require less stochasticity in its action selection than AlphaZero (guiding principle (c)). My hypothesis was mostly confirmed in Connect Four but the results were less clear in 9x9 Go. In the hyperparameter sweeps performed in Connect Four, AlphaZero achieved its best AUC against MCTS-Solver 10x with ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 10, \alpha = 1.0, \epsilon = 0.25$). GEVE, GEVC, GESR, and GESC achieved their best AUCs with ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 5, \alpha = 1.0, \epsilon = 0.25$), ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 10, \alpha = 1.0, \epsilon = 0.1$), ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 2, \alpha = 1.0, \epsilon = 0.25$), and ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 10, \alpha = 1.0, \epsilon = 0.25$), respectively. GEVE was tuned more exploitatively than AlphaZero because it used fewer action sampling moves and had the remaining hyperparameters set identically to AlphaZero's. GEVC was tuned more exploitatively than AlphaZero because its Dirichlet noise magnitude ϵ was smaller than AlphaZero's and the remaining hyperparameters were set identically to AlphaZero's. GESR was tuned more exploitatively than AlphaZero because it used fewer action sampling moves and had the remaining hyperparameters set identically to AlphaZero's. GESC, on the other hand, performed best when all of its hyperparameters were set identically to AlphaZero's. Thus, three of the four variants of Go-Exploit were tuned more exploitatively than AlphaZero in Connect Four. This suggests that Go-Exploit relies less upon stochastic action selection than AlphaZero to sufficiently explore the state space. This enables Go-Exploit to produce and train under policies that are inherently more exploitative, accelerating policy iteration.

In 9x9 Go, AlphaZero achieved its best AUC with ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 2, \alpha = 0.03, \epsilon = 0.1$). GEVE, GEVC, GESR, and GESC achieved their best AUCs with ($c_{\text{pucb}} = 2.0, \tau = 1.0, k = 1, \alpha = 0.03, \epsilon = 0.1$), ($c_{\text{pucb}} = 2.0, \tau = 1.0, k = 1, \alpha = 0.03, \epsilon = 0.1$), ($c_{\text{pucb}} = 1.0, \tau = 1.0, k = 2, \alpha = 0.03, \epsilon = 0.1$), and ($c_{\text{pucb}} = 2.0, \tau = 1.0, k = 1, \alpha = 0.03, \epsilon = 0.1$), respectively. Each algorithm's Dirichlet noise was parameterized identically and each algorithm used the same temperature τ. Three of

the four variants of Go-Exploit used fewer action sampling moves than AlphaZero, however, AlphaZero used a smaller c_{pucb} value than these algorithms. It's difficult to assess whether AlphaZero was more exploitative with its smaller c_{pucb} value or whether three of the four variants of Go-Exploit were more exploitative with fewer action sampling moves.

Chapter 6

Conclusions, Recommendations, & Future Work

In this thesis, I have identified limitations in AlphaZero's training procedure and introduced a novel search control algorithm, called Go-Exploit, that helps mitigate them. AlphaZero inadequately explores a game tree because it begins each of its self-play matches from the initial state of a game and often transitions into a terminal state before reaching and exploring states deeper in the game tree. This issue is exacerbated by the fact that AlphaZero only samples actions for the first k moves of a self-play match. In addition, AlphaZero's policy iteration is hindered by its exploration mechanisms. The Dirichlet noise interferes with the search's ability to produce improved policies and the action sampling causes exploratory actions to be selected during self-play. These two factors cause AlphaZero's neural network to train upon weaker, exploratory policies π_t, and on self-play match outcomes z produced under these weaker policies. AlphaZero's value training also suffers from the lack of independent value targets. AlphaZero only produces a single value target z per self-play match whereas a unique policy target π_t is produced at each visited state. The scarcity of the independent value targets and their noisiness from the exploration mechanisms hamper AlphaZero's ability to train an accurate value function.

To address these limitations, I developed Go-Exploit, a new search control strategy for AlphaZero. Instead of beginning each self-play match from the initial state of a

game s_0, Go-Exploit samples the start state of a self-play trajectory from an archive of "states of interest". Afterward, Go-Exploit produces the remainder of the self-play trajectory identically to AlphaZero. In this thesis, I experimented with two definitions of "states of interest". Go-Exploit Visited States considers all states visited during self-play as states of interest in order to improve action selection from the states visited under the current policy. Go-Exploit Search States considers search states appearing in trajectories beginning from s_0 as states of interest in order to improve the value estimates of the states that influence the policies π_t returned by search. In this thesis, I also experimented with three archive configurations. The Expanding Archive stores all encountered states of interest and is only viable when there are no memory constraints. The Reservoir Archive is a fixed-size archive that approximates the distribution of states in the Expanding Archive. The Circular Archive is a fixed-size archive that only stores the most recently observed states of interest, focusing training on the states encountered under the most recent policies.

To evaluate the different variants of Go-Exploit relative to AlphaZero, experiments were conducted in Connect Four and 9x9 Go. Ultimately, in both Connect Four and 9x9 Go, all variants of Go-Exploit achieved greater AUCs than AlphaZero in evaluation matches played against MCTS-Solver over the course of training. In Connect Four, Go-Exploit exhibited its greatest learning efficiency with a Circular Archive consisting of search states. In 9x9 Go, the performances of the four variants of Go-Exploit were more even but Go-Exploit performed marginally better with a Circular Archive consisting of visited states. These results were reaffirmed in head-to-head matches played between the variants of Go-Exploit and AlphaZero.

In this thesis, I also compared Go-Exploit to KataGo, a reimplementation of AlphaZero that improves AlphaZero's sample efficiency. Go-Exploit's search control procedure resulted in faster learning than KataGo's search control mechanisms of trajectory initialization and position branching in Connect Four. On the other hand, Go-Exploit's sample efficiency improved when combined with some of KataGo's other

innovations, illustrating their compatibility.

Finally, I performed experiments to understand why Go-Exploit exhibits a greater sample efficiency than AlphaZero. The results of these experiments reveal that Go-Exploit does indeed mitigate the limitations identified in AlphaZero's training procedure. First, Go-Exploit improves upon AlphaZero's ability to explore states deeper in game trees. By sampling the start state of its self-play trajectories from its archive, Go-Exploit begins its self-play trajectories from states throughout the game tree and then utilizes AlphaZero's exploration mechanisms to explore new states. This was evidenced by Go-Exploit visiting more unique states over a majority of game tree depths compared to AlphaZero. Second, Go-Exploit trains upon more independent value targets than AlphaZero. Since Go-Exploit begins its self-play trajectories from states throughout the game tree, its self-play trajectories are much shorter, on average, than AlphaZero's. This allows Go-Exploit to produce and train upon more independent value targets, improving its value training. Finally, Go-Exploit often trains under more exploitative policies than AlphaZero, accelerating policy iteration. Since exploration is incorporated into Go-Exploit's trajectory initialization, Go-Exploit relies less upon stochastic action selection for exploration than AlphaZero. This was observed in the Connect Four hyperparameter sweep where three of the four variants of Go-Exploit tuned k or ϵ more exploitatively than AlphaZero. However, in 9x9 Go, the values of competing hyperparameters made it more difficult to conclude whether Go-Exploit was tuned more exploitatively than AlphaZero.

In this thesis, I 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 performs with a more theoretically sound policy improvement operator [56], with a learned model [53], and in non-deterministic or imperfect information games [57].

Bibliography

[1] A. M. Turing, "Intelligent machinery," *National Physical Laboratory*, 1948.

[2] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[3] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, no. 3, 210–229, 1959, ISSN: 0018-8646. DOI: 10.1147/rd.33.0210. [Online]. Available: https://doi.org/10.1147/rd.33.0210.

[4] B. Copeland, J. Bowen, M. Sprevak, and R. Wilson, *The Turing Guide*, ser. The Turing Guide. Oxford University Press, 2017, ISBN: 9780198747826. [Online]. Available: https://books.google.ca/books?id=dlwjDgAAQBAJ.

[5] A. Newell, J. C. Shaw, and H. A. Simon, "Chess-playing programs and the problem of complexity," *IBM Journal of Research and Development*, vol. 2, no. 4, pp. 320–335, 1958.

[6] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2-3, pp. 273–289, 1992.

[7] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[8] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[9] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[10] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

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

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "First return, then explore," *Nature*, vol. 590, no. 7847, pp. 580–586, 2021.

[14] A. Tavakoli, V. Levdik, R. Islam, C. M. Smith, and P. Kormushev, "Exploring restart distributions," *arXiv preprint arXiv:1811.11298*, 2018.

[15] D. J. Wu, "Accelerating self-play learning in go," *arXiv preprint arXiv:1902.10565*, 2019.

[16] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[17] J. v. Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische annalen*, vol. 100, no. 1, pp. 295–320, 1928.

[18] C. B. Browne *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[19] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *International Conference on Computers and Games*, Springer, 2006, pp. 72–83.

[20] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning*, Springer, 2006, pp. 282–293.

[21] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3-4, pp. 285–294, 1933.

[22] R. R. Bush and F. Mosteller, "A stochastic model with applications to learning," *The Annals of Mathematical Statistics*, pp. 559–585, 1953.

[23] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, pp. 527–535, 1952.

[24] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

[25] T. L. Lai, H. Robbins, *et al.*, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[26] T. L. Lai, "Adaptive treatment allocation and the multi-armed bandit problem," *The Annals of Statistics*, pp. 1091–1114, 1987.

[27] M. N. Katehakis and H. Robbins, "Sequential choice from several populations.," *Proceedings of the National Academy of Sciences*, vol. 92, no. 19, pp. 8584–8585, 1995.

[28] R. Agrawal, "Sample mean based index policies by o (log n) regret for the multi-armed bandit problem," *Advances in Applied Probability*, vol. 27, no. 4, pp. 1054–1078, 1995.

[29] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multi-armed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[30] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[31] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[32] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[33] R. A. Howard, "Dynamic programming and markov processes.," 1960.

[34] M. L. Puterman and S. L. Brumelle, "On the convergence of policy iteration in stationary dynamic programming," *Mathematics of Operations Research*, vol. 4, no. 1, pp. 60–69, 1979.

[35] D. Bertsekas and J. Tsitsiklis, *Parallel and distributed computation: numerical methods*. Athena Scientific, 2015.

[36] D. Bertsekas, "Distributed dynamic programming," *IEEE transactions on Automatic Control*, vol. 27, no. 3, pp. 610–616, 1982.

[37] D. P. Bertsekas, "Distributed asynchronous computation of fixed points," *Mathematical Programming*, vol. 27, no. 1, pp. 107–120, 1983.

[38] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *Artificial intelligence*, vol. 72, no. 1-2, pp. 81–138, 1995.

[39] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine learning proceedings 1990*, Elsevier, 1990, pp. 216–224.

[40] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine learning*, vol. 13, no. 1, pp. 103–130, 1993.

[41] J. Peng and R. J. Williams, "Efficient learning and planning within the dyna framework," *Adaptive behavior*, vol. 1, no. 4, pp. 437–454, 1993.

[42] R. S. Sutton, C. Szepesvári, A. Geramifard, and M. P. Bowling, "Dyna-style planning with linear function approximation and prioritized sweeping," *arXiv preprint arXiv:1206.3285*, 2012.

[43] Y. Pan, H. Yao, A.-m. Farahmand, and M. White, "Hill climbing on value estimates for search-control in dyna," *arXiv preprint arXiv:1906.07791*, 2019.

[44] Y. Pan, J. Mei, and A.-m. Farahmand, "Frequency-based search-control in dyna," *arXiv preprint arXiv:2002.05822*, 2020.

[45] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3, pp. 293–321, 1992.

[46] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[47] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," *Advances in neural information processing systems*, vol. 20, 2007.

[48] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[49] T. Hester *et al.*, "Deep q-learning from demonstrations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[50] J. Oh, Y. Guo, S. Singh, and H. Lee, "Self-imitation learning," in *International Conference on Machine Learning*, PMLR, 2018, pp. 3878–3887.

[51] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[52] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[53] J. Schrittwieser *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[54] M. Lanctot *et al.*, "Openspiel: A framework for reinforcement learning in games," *arXiv preprint arXiv:1908.09453*, 2019.

[55] M. H. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo tree search solver," in *International Conference on Computers and Games*, Springer, 2008, pp. 25–36.

[56] I. Danihelka, A. Guez, J. Schrittwieser, and D. Silver, "Policy improvement by planning with gumbel," in *International Conference on Learning Representations*, 2021.

[57] M. Schmid *et al.*, "Player of games," *arXiv preprint arXiv:2112.03178*, 2021.

[58] L. V. Allis *et al.*, *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[59] J. Tromp, "The number of legal go positions," in *International Conference on Computers and Games*, Springer, 2016, pp. 183–190.

[60] B. Brügmann, "Monte carlo go," Citeseer, Tech. Rep., 1993.

[61] M. Enzenberger, M. Müller, B. Arneson, and R. Segal, "Fuego—an open-source framework for board games and go engine based on monte carlo tree search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 259–270, 2010.

[62] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[63] C. Clark and A. Storkey, "Training deep convolutional neural networks to play go," in *International conference on machine learning*, PMLR, 2015, pp. 1766–1774.

[64] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, "Move evaluation in go using deep convolutional neural networks," *arXiv preprint arXiv:1412.6564*, 2014.

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

Appendix A: History

Over the first few decades of AI research, heuristic search was the most popular approach used in game playing programs. Working as a codebreaker during World War II, Alan Turing devoted his free time to thinking about how to create an intelligent chess program [4]. Turing's ideas were heavily inspired by his codebreaking. As a codebreaker, Turing's job was to decipher the daily knob settings of the Germans' Enigma machine which was used to encrypt messages. To efficiently search through the permutations of the knob settings, Turing built a mechanical machine called 'the bombe'. Turing quickly realized that performing an exhaustive search would take way too long so he implemented a mechanical version of heuristic search. Funnily enough, the heuristics used made assumptions about human behaviour (particularly human laziness) and sufficiently limited the search so that it was tractable within a day.

In working on the bombe, Turing realized that machine intelligence could be achieved via a guided search. This inspired Turing to use heuristic search in his original chess playing programs. Turing recognized that under optimal play, players always act to maximize their gains and to minimize their losses from potential moves their opponent could make. This intuition was originally formalized in John von Neumann's Minimax Theorem in 1928 [17]. In 1948, Turing ultimately concretized his thoughts into his first chess playing program, Turochamp, with his collaborator David Champernowne [4]. To determine which move to make, Turochamp evaluated each possible state that was two moves ahead, extending the search deeper for states that met certain conditions. Leaf nodes in the search tree were evaluated using a

handcrafted evaluation function that computed the ratio of white's piece values to black's piece values. The Minimax principle was then applied to back up state values up the search tree and to ultimately decide which move to make. Variations of this Minimax search algorithm were also used in Claude Shannon's chess program [2] and Arthur Samuel's checkers player [3].

In 1958, Newell, Shaw, and Simon [5] augmented Minimax search with Alpha-Beta pruning. Alpha-Beta pruning eliminates a given state's unexplored subtrees once it has been proven that an alternative move is more promising. Despite the improved search efficiency, their NSS chess program still played at a very weak level. Decades later, programs using these same ideas became very powerful when paired with computers that were significantly faster and had much larger memory. For example, in 1989, the University of Alberta's Jonathan Schaeffer created a program called CHINOOK [6] that defeated the world checkers champion using a database of opening moves, Alpha-Beta search, and an endgame database. In 1997, Deep Blue [7] defeated the world chess champion, Gary Kasparov, using Alpha-Beta search.

Heuristic search with a handcrafted evaluation function was seminal to the successes in checkers and chess but further progress was limited by computational power and by the quality of the handcrafted evaluation function. More informed action selections are made when the breadth and depth of search increases, however, this is only possible if a greater search time is allocated or if a machine with faster processing speed is used. The quality of the action selection also depends upon the value estimates that are being assigned to states by the handcrafted evaluation function. AI researchers often teamed up with expert human players to design complex evaluation functions for a given game [7]. The quality of these handcrafted evaluation functions was limited by human knowledge and by the ability to represent the human knowledge in functional form. To overcome these two obstacles, new ideas were needed.

New progress was made when learning was incorporated into the heuristic search algorithms that had previously shown success. Arthur Samuel was the first to experi-

ment with learning in his checkers player [3]. Like its predecessors, Samuel's checkers player performed a Minimax search to select its moves. The program's evaluation function was a linear polynomial consisting of a subset of parameters from a pool of features believed to be relevant to checkers. Samuel's checkers player would learn from matches in two ways. The first method, called "rote-learning", consisted of saving the backed-up Minimax values of played states in memory. When these states appeared as leaf nodes in future searches, they were assigned their saved Minimax values, effectively deepening the search. The second form of learning was "learning by generalization" and was used to update the coefficients of the evaluation function. This form of learning helped inspire temporal-difference learning [9] because it sought to make the values of states consistent with the values of the likeliest successor states. To achieve this, Samuel used the Minimax value of the current game state as a learning target for the value function estimate of the state two moves prior.

Gerald Tesauro extended Samuel's ideas in his backgammon program TD-Gammon [8]. Unlike its predecessors, TD-Gammon made use of a neural network rather than a handcrafted linear evaluation function. The neural network took a board position as input and it output the estimated outcome of the game from the given board position. The neural network was trained using TD(λ), a reinforcement learning algorithm. At each time step, TD-Gammon would compute the difference between the outputs of the neural network at time t and time t+1. This temporal difference error was then multiplied by an eligibility trace vector to help determine the change in weights for the neural network. Once a self-play game was complete, the neural network's weights were updated using the difference between the actual outcome of the self-play game and the final neural network output, allowing the neural network's value estimates to be based on the true value of terminal states. To select moves within self-play games, TD-Gammon used its neural network to evaluate all possible board positions from a given dice roll and selected the action with the highest estimated value. TD-Gammon's success illustrated the feasibility and the potency of reinforcement learning

when paired with a nonlinear value function like a neural network.

Despite the successes in checkers, chess, and backgammon, progress was still limited in the much larger game of Go. Checkers has 10^{20} unique legal states, an average branching factor of 2.8, and an average game length of 70 [58]. Chess has 10^{44} unique legal states, an average branching factor of 35, and an average game length of 70 [2]. On the standard 19×19 board, Go has 10^{170} unique legal states, an average branching factor of 250, and an average game length of 150 [59]. Scientists tried applying the approaches that were successful in checkers, chess, and backgammon to Go but the significant jump in complexity limited their success. The increased branching factor forced the search algorithms to look fewer moves ahead because there were more unique states to investigate per ply. Furthermore, scientists struggled to devise handcrafted evaluation functions that accurately assessed the relative values of states.

Significant progress in computer Go was made with the development of Monte Carlo Tree Search. In 1993, Bernd Brugmann [60] released a Go program, called Gobble, that simulated numerous random games from the current board position and then selected the move with the best average outcome. Rémi Coulom [19] combined Brugmann's Monte Carlo position evaluation with tree search to develop the first version of what is now known as Monte Carlo Tree Search. In the same year, Kocsis and Szepesvari [20] extended Monte Carlo Tree Search with their bandit inspired algorithm called UCT (UCB Applied to Trees). UCT iteratively builds out a search tree via simulated games much like Coulom's algorithm, however, moves are selected using an action selection rule inspired by the UCB1 algorithm [29]. This action selection rule balances the exploration of infrequently visited actions with uncertain value estimates and the exploitation of actions with large value estimates so that it can refine their value estimates and confidently return the optimal action. For the next few years, computer Go programs using MCTS/UCT dominated international competitions [61].

The emergence of deep convolutional neural networks (DCNNs) enabled game playing programs to harness more powerful and more generalizable evaluation functions. In 2012, Krizhevsky et al. [62] published work on a DCNN that significantly outperformed traditional computer vision algorithms in the ImageNet image classification competition. This inspired computer Go researchers to investigate whether DCNNs could be used to represent Go knowledge. Researchers had previously attempted to use CNNs as an evaluation function in computer Go programs but these neural networks were limited to a single hidden layer. In 2014, two research groups concurrently tried to use DCNNs to learn to represent human Go knowledge. Storkey and Clark [63] and Maddison et al. [64] independently trained deep convolutional neural networks on databases of human expert moves so that it could learn to predict the move a human expert would make when given a board position. This approach yielded a test accuracy of 55%, which was a new benchmark at the time. Maddison et al. then detailed preliminary attempts at integrating their DCNN into MCTS, which ultimately inspired AlphaGo and AlphaZero.

In 2016, DeepMind published a paper on their Go playing program AlphaGo [65]. The paper revealed that AlphaGo had defeated Fan Hui, the European Go champion, five games to zero – the first time a Go program had ever defeated a professional Go player. Later that year, a slightly modified version of AlphaGo defeated Lee Sedol, the world Go champion, four games to one. To determine the action it plays, AlphaGo performs a Monte Carlo Tree Search that is guided by a policy network and a value network. AlphaGo's policy network outputs a discrete probability distribution over the actions that can be taken from an input state and is initially trained upon a dataset of human expert moves. AlphaGo's value network is randomly initialized and predicts the outcome of the game from the input state under its current policy. During search, the policy network's action probabilities are used to assign priors to actions in order to bias the search to promising actions. The value estimates used during search are weighted averages of value estimates output by the value network

and rollouts from leaf nodes in the search tree. As self-play games complete, the outcomes of the games are used to update the weights of both neural networks to place greater probabilities on actions that lead to wins and to improve the value estimates under the improving policy.

While AlphaGo achieved incredible success, it still needed to train upon human expert moves to surpass human level play. The next step was to develop a program that could achieve superhuman play by exclusively learning from games played against itself. This feat was achieved in 2017 and 2018 with DeepMind's AlphaGo Zero [10] and AlphaZero [11] algorithms. Both AlphaGo Zero and AlphaZero achieve superhuman play in Go knowing only the rules of the game and beginning from random play. Their search is guided by a single policy-value network that outputs a policy from its policy head and a value estimate from its value head. Action values in the search solely depend upon the value estimates output by the policy-value network (no rollouts). The distribution of search visits are converted into a policy π that is used to select an action during self-play and also serves as a training target for the policy head. The outcomes of self-play games are used as training targets for the value head. The action selection rule used during search concentrates search visits on actions with large value estimates, ensuring that the policy-value network is trained upon improving policies π and on self-play game outcomes produced under the improved policy. This process, called policy iteration, enables AlphaGo Zero and AlphaZero to learn progressively better policies over time and to ultimately achieve superhuman play.

Appendix B: Pseudocode

Algorithm 2 Go-Exploit

Parameters: total_steps, num_training_actors, num_archive_actors, A_{type}, $|A|$, use_search_states, $|B|$, b_{step}, b_{batch}, λ, iters, α, ϵ, c_{pucb}, τ, k, lr, c

Initialize trajectory queue Q
Initialize policy-value network f_θ
$A = \text{initialize\_archive}(A_{\text{type}}, |A|, s_0)$
for $i \in 1, ..., \text{num\_training\_actors}$ **do**
    $\text{training\_actor}(A, \lambda, \text{iters}, \alpha, \epsilon, c_{\text{pucb}}, \tau, k, Q)$
end for
for $i \in 1, ..., \text{num\_archive\_actors}$ **do**
    $\text{archive\_actor}(A, A_{\text{type}}, \text{use\_search\_states}, \text{iters}, \alpha, \epsilon, c_{\text{pucb}}, \tau, k)$
end for
$\text{learner}(\text{total\_steps}, |B|, b_{\text{step}}, b_{\text{batch}}, Q, A, A_{\text{type}}, \text{use\_search\_states}, lr, c)$

Algorithm 3 training_actor

Parameters: A, λ, iters, α, ϵ, c_{pucb}, τ, k, Q

while True **do**
  trajectory = []
  $t = 0$
  s_t = initialize_game()
  r = rand_num(0, 1)
  if $r > \lambda$ **then**
    $s_t = A$.sample()
  end if
  while s_t is not terminal **do**
    π_t = search(iters, $\alpha, \epsilon, c_{\text{pucb}}, \tau$)
    if $t < k$ **then**
      $a_t \sim \pi_t$
    else
      $a_t = \arg\max_a \pi_t[a]$
    end if
    trajectory.add(s_t, π_t)
    s_t = take_action(s_t, a_t)
    $t = t + 1$
  end while
  $z = s_t$.outcome()
  trajectory.set_outcome(z)
  Q.push(trajectory)
end while

Algorithm 4 archive_actor

Parameters: A, A_{type}, use_search_states, iters, α, ϵ, c_{pucb}, τ, k

while True **do**
  $A_{\text{temp}} = []$
  $t = 0$
  $s_t = \text{initialize\_game}()$
  while s_t is not terminal **do**
    π_t, search_states $= \text{search}(\text{iters}, \alpha, \epsilon, c_{\text{pucb}}, \tau)$
    if $t < k$ **then**
      $a_t \sim \pi_t$
    else
      $a_t = \arg\max_a \pi_t[a]$
    end if
    A_{temp}.add(search_states)
    $s_t = \text{take\_action}(s_t, a_t)$
    $t = t + 1$
  end while
  if use_search_states **then**
    A.update$(A, A_{\text{temp}}, A_{\text{type}})$
  end if
end while

Algorithm 5 learner

Parameters: total_steps, $|B|$, b_{step}, b_{batch}, Q, A, A_{type}, use_search_states, lr, c

B = initialize_buffer($|B|$)
for step $\in 1, ..., \text{total\_steps}$ **do**
  $A_{\text{temp}} = []$
  step_states = 0
  while step_states $< b_{\text{step}}$ **do**
    trajectory = Q.pop()
    for $(s_t, \boldsymbol{\pi}_t, z) \in$ trajectory **do**
      B.add$(s_t, \boldsymbol{\pi}_t, z)$
      step_states = step_states + 1
      A_{temp}.add(s_t)
    end for
  end while
  $b_{\text{train}} = B$.sample(b_{batch})
  f_θ.update$(b_{\text{train}}, lr, c)$
  if NOT use_search_states **then**
    A.update$(A, A_{\text{temp}}, A_{\text{type}})$
  end if
end for

Algorithm 6 Archive update()

Parameters: A, A_{temp}, A_{type}

for $s \in A_{\text{temp}}$ **do**
  if A_{type} == "Expanding" **then**
    A.push(s)
  else if A_{type} == "Circular" **then**
    if A.size() $< |A|$ **then**
      A.push(s)
    else
      A.pop()
      A.push(s)
    end if
  else if A_{type} == "Reservoir" **then**
    if A.size() $< |A|$ **then**
      A.push(s)
    else
      $i \sim \text{rand\_int}(0, n-1)$
      if $i < |A|$ **then**
        $A[i] = s$
      end if
    end if
    $n = n + 1$
  end if
end for

Appendix C: Hyperparameter Sweeps

	Connect Four	**9x9 Go**
Training actors	700	700
Archive actors	50	50
f_θ**'s depth**	10 residual blocks	10 residual blocks
f_θ**'s width**	256 filters	256 filters
$\|B\|$	2^{17}	2^{17}
b_{step}	2^{12}	2^{12}
b_{batch}	8 mini-batches of 2^9	8 mini-batches of 2^9
Search iterations	100	400
Learning steps	600	900

Table C.1: Fixed hyperparameter values

	AlphaZero	GEVE	GEVC	GESR	GESC
lr	$\leftarrow [10^{-2}, 10^{-3}, 10^{-4}] \rightarrow$				
c	$\leftarrow [10^{-4}, \mathbf{10^{-5}}, 10^{-6}] \rightarrow$				
α	$\leftarrow [0.03, \mathbf{1.0}, 5.0] \rightarrow$				
ϵ	$\leftarrow [0.05, 0.1, \mathbf{0.25}, 0.5] \rightarrow$				
c_{pucb}	$\leftarrow [0.5, \mathbf{1.0}, 2.0, 4.0] \rightarrow$				
k	$[5, \mathbf{10}, 20, 30]$	$[2, 5, \mathbf{10}, 20]$	$[5, \mathbf{10}, 20]$	$[1, 2, 5, \mathbf{10}, 20]$	$[5, \mathbf{10}, 20]$
λ	N/A	$\leftarrow [0, 0.01, \mathbf{0.1}, 0.25] \rightarrow$			
$\|A\|$	N/A	N/A	$\leftarrow [10^5, \mathbf{10^6}, 2{\times}10^6] \rightarrow$		
τ	$\leftarrow [0.5, 0.75, \mathbf{1.0}, 2.0] \rightarrow$				

Table C.2: Hyperparameters swept over in Connect Four. The hyperparameters were swept over in descending order in the table (lr first and τ last). When a set of hyperparameter values is bounded with arrows ($\leftarrow [\dots] \rightarrow$), it indicates that this set of hyperparameter values was swept over by each algorithm in the column. The values that are bolded were the values that hyperparameters were set to prior to being swept over.

	AlphaZero	GEVE	GEVC	GESR	GESC
lr	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}
c	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
α	1.0	1.0	1.0	1.0	1.0
ϵ	0.25	0.25	0.1	0.25	0.25
c_{pucb}	1.0	1.0	1.0	1.0	1.0
k	10	5	10	2	10
λ	N/A	0.1	0.1	0.0	0.01
$\|A\|$	N/A	N/A	10^6	10^6	10^5
τ	1.0	1.0	1.0	1.0	1.0

Table C.3: Best hyperparameter values in Connect Four

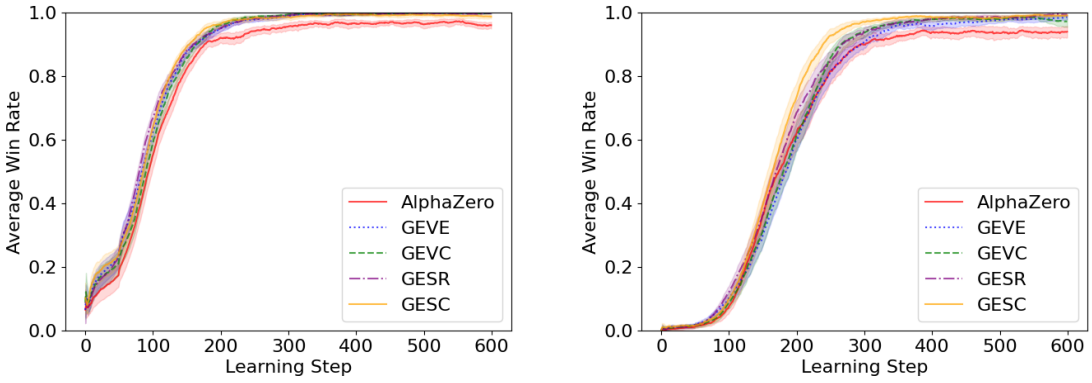
	AlphaZero	GEVE	GEVC	GESR	GESC
lr	$\leftarrow [10^{-2}, 10^{-3}, 10^{-4}] \rightarrow$				
c	$\leftarrow [10^{-4}, \mathbf{10^{-5}}, 10^{-6}] \rightarrow$				
α	$\leftarrow [\mathbf{0.03}, 1.0, 5.0] \rightarrow$				
ϵ	$\leftarrow [0.05, \mathbf{0.1}, 0.25, 0.5] \rightarrow$				
c_{pucb}	$\leftarrow [0.5, 1.0, \mathbf{2.0}, 4.0] \rightarrow$				
k	$\leftarrow [1, 2, \mathbf{5}, 10] \rightarrow$				
λ	N/A	$\leftarrow [0.01, \mathbf{0.1}, 0.25] \rightarrow$			
$\|A\|$	N/A	N/A	$[10^5, \mathbf{10^6}, 2{\times}10^6]$	$\leftarrow [10^5, \mathbf{10^6}, 2{\times}10^6, 5{\times}10^6] \rightarrow$	
τ	$\leftarrow [0.5, 0.75, \mathbf{1.0}, 2.0] \rightarrow$				

Table C.4: Hyperparameters swept over in 9x9 Go. The hyperparameters were swept over in descending order in the table (lr first and τ last). When a set of hyperparameter values is bounded with arrows ($\leftarrow [\dots] \rightarrow$), it indicates that this set of hyperparameter values was swept over by each algorithm in the column. The values that are bolded were the values that hyperparameters were set to prior to being swept over.

	AlphaZero	GEVE	GEVC	GESR	GESC
lr	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}
c	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
α	0.03	0.03	0.03	0.03	0.03
ϵ	0.1	0.1	0.1	0.1	0.1
c_{pucb}	1.0	2.0	2.0	1.0	2.0
k	2	1	1	2	1
λ	N/A	0.1	0.1	0.1	0.1
$\|A\|$	N/A	N/A	10^6	2×10^6	2×10^6
τ	1.0	1.0	1.0	1.0	1.0

Table C.5: Best hyperparameter values in 9x9 Go

Appendix D: Additional Plots

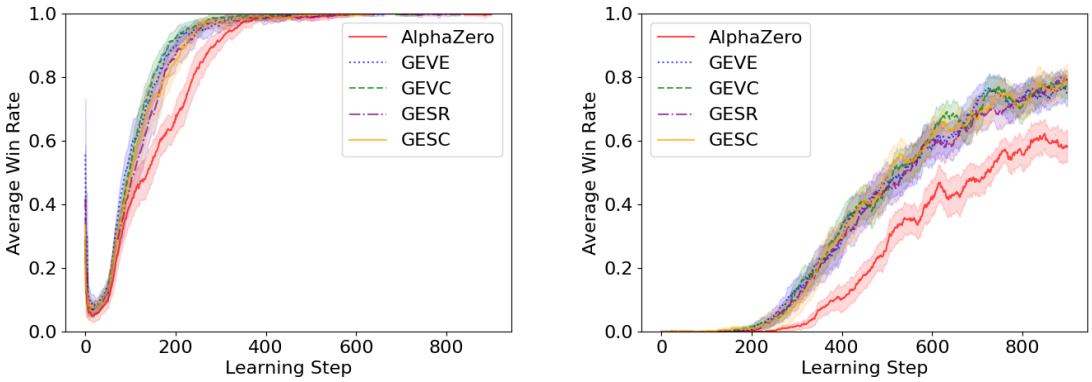


(a) MCTS-Solver 1x

(b) MCTS-Solver 100x

Figure D.1: AlphaZero and Go-Exploit's win rates against MCTS-Solver 1x and 100x in Connect Four. The win rates were averaged over the 30 validation runs and the shaded regions represent 95% confidence intervals.

.



(a) MCTS-Solver 1x

(b) MCTS-Solver 100x

Figure D.2: AlphaZero and Go-Exploit's win rates against MCTS-Solver 1x and 100x in 9x9 Go. The win rates were averaged over the 30 validation runs and the shaded regions represent 95% confidence intervals.

.