Single-Agent Optimization with Monte-Carlo Tree Search and Deep Reinforcement Learning

by

Arta Seify

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Single-agent optimization tasks, also referred to as single-player games, include any domain with an agent whose goal is to maximize an objective function(s), without interference from any other agents. Such tasks have been studied for decades. For example, in 2006, NASA automated the design of antennas by framing the problem as a single-agent optimization task.

The combination of Monte-Carlo Tree Search (MCTS) and deep reinforcement learning is state-of-the-art in zero-sum two-player perfect-information games. In this thesis, my goal is to bring the success of these algorithms to single-player games. I begin by introducing a variant of MCTS that is suitable for games where the bounds on rewards is not known, which is the case in many optimization problems. My enhancements include using a general action-value normalization technique, as well as a virtual loss function, which enables effective search parallelization. I then introduce Policy-MCTS, which uses a deep policy network trained by generations of self-play to guide the search. Lastly, I provide initial work on using value estimations to entirely replace the rollouts of MCTS.

I gauge the effectiveness of my methods in “SameGame”, an NP-hard single-player test domain. I demonstrate that Policy-MCTS is competitive with state-of-the-art search-based methods on a benchmark set of positions.

Preface

Parts of the work presented in Chapters 4 and 5 were accepted for publication in the Thirty-Fourth AAAI Conference on Artificial Intelligence Workshop on Reinforcement Learning in Games.

I have not failed. I've just found 10,000 ways that won't work.

– Thomas A. Edison

Acknowledgements

I would like to first thank my supervisor, Dr. Michael Buro, for his advice and mentorship throughout my graduate studies. The work in this thesis would not have been possible without his help and guidance. I would also like to acknowledge my undergraduate supervisor, Dr. Peter Høyer, whose advice on research and writing I follow to this day. I also appreciated the friendship and help from my peers Chris Solinas, Douglas Rebstock, Varun Bhatt, and Kristen Yu.

Additional acknowledgments go to the funding I received from the Government of Canada in the form of an NSERC, and computational resources provided by Compute Canada, without which I would not have been able to perform many of the experiments present in this thesis.

Lastly, I would like to send lots of love and thanks to my family. Your support means more to me than you will ever know. And to Paige, whose love and support kept me sane throughout my graduate studies. Thank you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Though we may not be completely aware of it, in our daily lives we are often concerned with optimization tasks. From selecting the precise order of actions in the morning to maximize sleep time, to finding the best routes to take during rush hour traffic, we are usually good at optimizing tasks such that we receive the highest reward (such as sleep), or the lowest amount of negative outcome (such as frustration).

There are a wide variety of optimization tasks, each with different goals and challenges. In this thesis, I am concerned with tasks that only have a single agent, and in which there is only one reward signal to maximize, otherwise known as single-agent optimization tasks. An everyday example of such a task is taking the grocery bags from the car to the house, while taking the least number of trips. The agent in this case is the person carrying the bags, and the reward signal is given by the number of trips, with a higher amount of trips giving a lower total reward.

Many interesting problems fit within this description. For example, NASA automated the design of antennas [Hor+06] by modeling the problem as a single-agent optimization task. Another notable task, given the increase in online shopping over the last decade, is the Warehouse Order Picking Problem, which has been studied for decades [DLR07]. A related problem is the 1D, 2D, and 3D Bin Packing Problems, where the goal is to maximize the number of packages while minimizing the surface used. Problems of interest to theoretical computing scientists, such as the Travelling Salesman Problem (TSP), can also be modeled as single-agent optimization tasks. Algorithms for solving TSP have many practical uses, such as computer chip design and order-picking in warehouses [The+10].

Given the importance of these problems, the field of Artificial Intelligence (AI) has studied such tasks for decades. Many areas within AI provide algorithms for solving optimization problems. Such algorithms include graph-based search A* [HNR68] and IDA* [Kor85], evolutionary inspired algorithms such as Genetic Algorithms [Hol92] and evolutionary strategies, and those based on reinforcement learning [SB18].

With the increase in computational power, heuristic search algorithms using Monte-Carlo simulations have become more practical for such tasks. These methods estimate the values of states using random simulations. Their generality makes them applicable to a wide variety of domains. Examples include two-player board games such as Go [Gel+12] and Hex [AHH10; Hua+13], real-time domains such as Ms. Pac-Man [PWL14], general video game playing [Per+19], and single-player games [Caz09; Ros11b; Sch+12].

In fact, Monte-Carlo Tree Search (MCTS) is guaranteed to converge to an optimal solution [KS06]. However, the process is often too slow for most applications. For this reason, many possible modifications and enhancements have been introduced [Bro+12], and it is still an active field of research. A recent advancement is to combine MCTS with deep reinforcement learning [ATB17; Sil+18]. These algorithms have achieved state-of-the-art performance in deterministic two-player perfect-information zero-sum games, such as Hex, Chess, Shogi, and Go. They iteratively train neural networks that, given a state of the game, predict both a policy and value. MCTS provides the agent with the ability to look ahead, while the policy and value predictions are used to decrease the width and depth of the search tree, respectively. Additionally, the trained policy networks can be surprisingly strong by themselves. For instance, AlphaZero's Go policy plays at human expert level without the need for forward search.

In this thesis, my goal is to bring the ideas of these algorithms to single-agent optimization tasks. Such problems can be represented as deterministic single-player games, which are a subset of deterministic two-player games; single-player games can be turned into two-player games in which the opponent only has the option to pass the turn. Therefore, it should be possible to successfully apply MCTS and deep reinforcement learning to single-player games as well. But, there are some differences between the two settings that make this task non-trivial.

One major change is that in the games considered by the above work, the rewards seen by the agent is one of $\{-1, 0, 1\}$, for loss, tie, or win, respectively. This is in contrast to single-player games, in which the bound on rewards can be unknown. While it might seem like a minor detail, it brings about two problems that I will address in this thesis.

The first issue is that the convergence guarantees of Upper Confidence Bound for Trees (UCT), which is one of the most common and successful variants of MCTS, assumes that rewards are in the range of $[0, 1]$. One strategy for ensuring that UCT converges to an optimal solution in the single-player setting is to estimate a standard deviation of the value of each action. Alternatively, action values can be normalized, which does not require such estimations; this is the approach taken in this thesis.

Secondly, search algorithms that require frequent predictions from neural networks strongly benefit from parallelization of the search, as it enables batch predictions, which are far more efficient. However, tree parallelization, which is the most successful parallelization strategy for MCTS in zero-sum two-player games, requires *virtual loss*, which in the literature has only been defined for the two-player zero-sum setting. In such domains, the bound of rewards is often known; this knowledge is implicitly used in the definition of the used virtual loss function. Then, to use this parallelization technique in domains with unknown rewards, a new virtual loss function needs to be defined.

Lastly, another difference between the settings is that in single-player games, values found during search are a lower bound on the optimal value. For this reason, the action selection strategy and the policy training target need to be adjusted.

This thesis is organized as follows: Chapter 2 is dedicated to providing background knowledge required to understand this thesis, as well as a review of related work. Chapter 3 is an in-depth introduction of MCTS, along with a review of common modifications to the algorithm, with emphasis on relevant work in the single-player setting. My MCTS algorithm is introduced in Chapter 4, which includes an action value normalization mechanism and a new virtual loss function that enables effective search parallelization. Chapter 5 combines my MCTS algorithm with a policy network, which is iteratively trained from scratch through self-play. In Chapter 6, I present initial work on completely replacing MCTS rollouts with value estimations. In the final chapter, I finish with concluding remarks and a list of possible future works.

Chapter 2

Background and Related Work

2.1 Background

In this section, I introduce some of the necessary background knowledge required to understand the rest of this thesis. As Monte-Carlo Tree Search is a major part of this thesis, it will be explained in detail in Chapter 3.

2.1.1 Single-Agent Optimization

The single-agent optimization problems that I consider are in the intersection of three sub-categories: 1) single-player, 2) deterministic, and 3) perfect-information. Such optimization tasks can be represented as single-player games. Each of these sub-categories are part of a larger category: single-player vs. multi-player (2 or more players), deterministic vs. stochastic, and perfect-information vs. imperfect-information. Every possible intersection leads to different classes of algorithms, each with different strengths and weaknesses. Algorithms often perform poorly when directly used in an intersection of problems they were not intended for, but ideas can be borrowed and adapted to create new algorithms that perform better on those problems. My contribution takes algorithmic ideas from two-player, deterministic, perfect-information games, and adapts them to the single-player case.

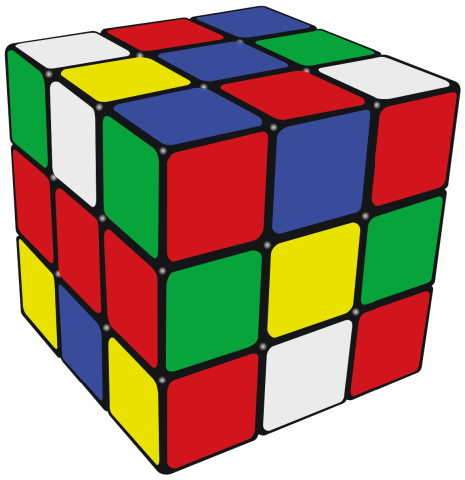
The term *single-player* refers to games that can only be played by one player at a time. The objective in most of these games falls into two categories: 1) find the goal state, or 2) maximize or minimize a numerical value called the *score*. Examples of games in the first category are the Rubik's Cube and 24-

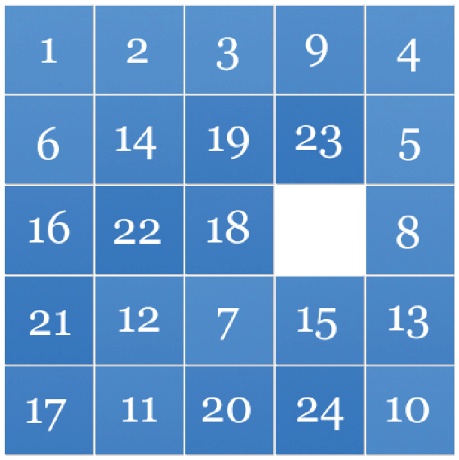
Puzzle. For these games, a custom scoring function is often used to make better comparisons between solutions; for example, the final score can be a linear combination of the total time used to reach a solution and the number of actions in the solution. Therefore, one can claim that the goal in most single-player games is to maximize or minimize a scoring function. In the domains considered in this thesis, the current score can be observed or calculated by the agent at any point. Note that any single-player game can be modeled as a two-player game, where the only action available to the opponent is to *pass* the turn.

In *deterministic* games, there is no randomness in state transitions. That is, taking the same action a in the same state s will always transition the player to the same next state s', for all s, a, and s'. A game such as Connect Four is fully deterministic, as there is no probability of a disc dropping into a column that was not selected. In contrast, the game Candy Crush has some element of stochasticity with the new pieces that fall down to replace the cleared ones.

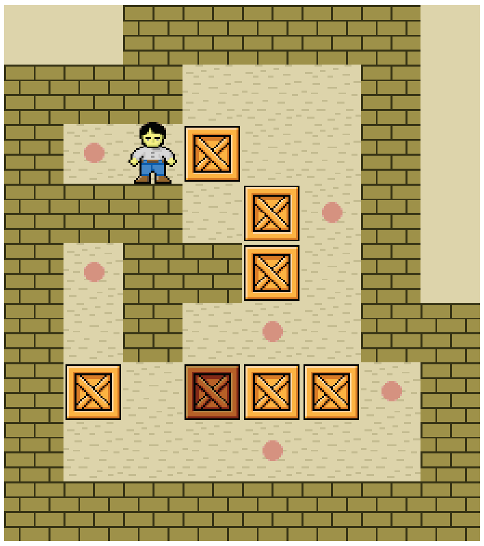
In *perfect-information* games, the complete state of the game is visible to all players; at each point, all players know the current state exactly. Additionally, every player is aware of and can choose between all legal actions at a given state. For the single-player setting considered in this thesis, the player must be completely aware of the current state, all possible actions, and the effects of those actions on the state, i.e. the player has a perfect model of the environment.

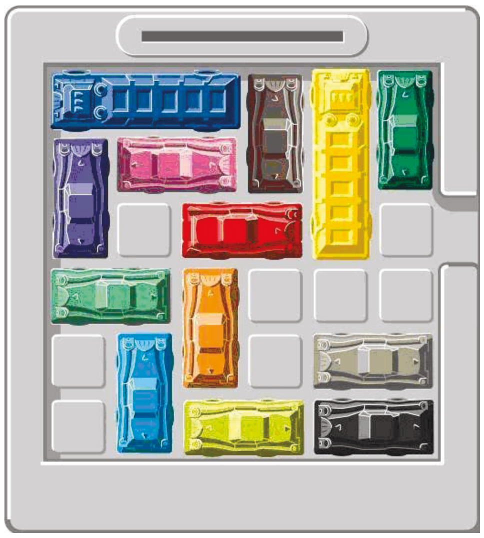
Examples of games that fit these categories include the Rubik's Cube, 24-Puzzle, Sokoban, Rush Hour, and SameGame (Figure 2.1). An interesting problem that also fits this description is the build order planning problem in real-time strategy video games such as StarCraft II. By defining a function that gives each unit a certain value, and modeling the opponent as part of the environment, the problem becomes a single-agent optimization task.


(a) Rubik's Cube



(b) 24-Puzzle


(c) Sokoban


(d) Rush Hour

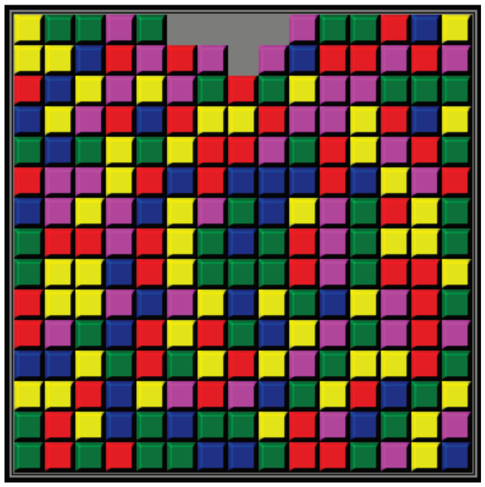

(e) SameGame

Figure 2.1: Examples of deterministic perfect-information single-player games.

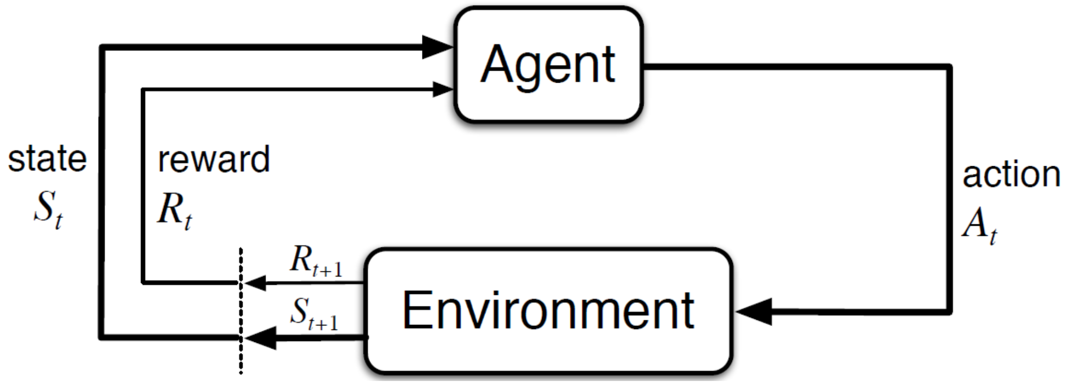


Figure 2.2: The interaction between an agent and the environment in a Markov Decision Process, as used in Reinforcement Learning [SB18].

2.1.2 Reinforcement Learning

Reinforcement learning (RL) problems involve an agent acting inside an environment, with the goal of learning to maximize a reward signal. The agent learns which actions to take by directly interacting with the environment, without any outside instruction. It must learn the effects of actions, including the effect on the reward signal, over extended periods of times. This description covers a broad range of problems and their corresponding potential solutions. Therefore, I only can provide an overview of relevant background information required to understand the work presented in this thesis.

RL problems are formalized as a Markov Decision Process (MDP). At each discrete time step t, the agent receives the state (in domains considered by this work) of the environment $S_t \in \boldsymbol{\mathcal{S}}$, and takes an action $A_t \in \boldsymbol{\mathcal{A}}$, receiving a reward R_{t+1} and transitioning to the next state S_{t+1}. This process is shown in Figure 2.2. The relevant domains for this thesis are all episodic, with each episode finishing once the agent reaches a terminal state. At each timestep, the task of the agent is to maximize the expected return, which is defined as

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T, \tag{2.1}$$

where T is the final time step.

To accomplish its goal, the agent learns a policy $\pi : \boldsymbol{\mathcal{S}} \to [0,1]^{|\boldsymbol{\mathcal{A}}|}$ (Chapter 5), as well as a value function $v_\pi : \boldsymbol{\mathcal{S}} \to \mathbb{R}$ (Chapter 6); both functions will be used to guide the search, as explained in their respective chapters. The policy takes as input a state $s \in \boldsymbol{\mathcal{S}}$, and outputs a vector with $|\boldsymbol{\mathcal{A}}|$ entries—one per possible action in the particular domain. The value of each entry is the probability of taking the corresponding action. The value function also takes as input a state s, and outputs the expected return under the current policy $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$. That is, the expected return at state s, in which actions are taken according to the probabilities given by a fixed policy π.

My agent also makes use of a perfect model of the environment. The model is used for planning: at each time step, the agent first creates a plan, then selects an action. This is known as model-based reinforcement learning. I use Monte-Carlo Tree Search as my planning algorithm, which is described in Chapter 3.

2.1.3 SameGame

SameGame (Figure 2.1) is a tile-matching game with the goal of maximizing the final score. In each move, the player can clear horizontally or vertically connected groups of size two or more of equal colour. The blocks above created holes will always fall down and then move left; When an entire column is cleared, the columns to the right are moved to the left. Each move scores $(\#\mathbf{BlocksCleared} - 2)^2$. The game is over when the player cannot take an action anymore. In addition, if the board is cleared, the player is awarded an additional $1,000$ points. Otherwise, the player receives a penalty. In the literature, there are two penalties that are used—one based on the total number of blocks left, and the other one based on the colours of the blocks left. The latter penalty is used in this thesis, which is calculated as follows: $\sum_c (\#\mathbf{BlocksLeft}_c - 2)^2$.

Deciding whether a general SameGame instance with at least five colours and two columns can be fully cleared has been shown to be at least NP-hard [Sch+08]. In the experiments reported in later chapters I use boards of size 7×7, 10×10, and 15×15, with five different block colours. Due to its difficulty, SameGame is a popular single-player test domain. There exists a set of 20 public test positions that have been used as a benchmark for multiple state-of-the-art search algorithms[1]; some of these algorithms are discussed in the next section. These positions will be used to compare the strength of my algorithm with other search based methods. All other boards used for training and testing are randomly generated; the board is created block by block, with the color of each block chosen uniformly at random.

2.2 Related Work

Work on single-agent optimization tasks span many decades, and includes multiple fields inside of artificial intelligence. I will begin by discussing state-of-the-art Monte-Carlo search algorithms for these domains. Next, pioneer works on combining MCTS and deep reinforcement learning for two-player games are presented. Lastly, I discuss similar work on single-player games.

2.2.1 MCTS in Single-Player Games

Schadd *et al.* introduced Single-Player MCTS (SP-MCTS) [Sch+12], which was one of the first successful application of MCTS to a single-player game with a large state and action space. SP-MCTS uses an additional term in the selection strategy of MCTS. This term accounts for the standard deviation in the rewards of an action; since single-player games are non-adversarial, taking into account the deviation of the action value can lead to a better estimate of the upper confidence bound. The algorithm also evenly splits the compute budget over multiple runs. Other contributions of the work include adding a percentage of the max value found to the average in the selection strategy, creating a tree-per-move rather than tree-per-game by playing the highest valued action, and a few value normalization methods. Several of these ideas are also used in my algorithm.

Cazenave developed a recursive variant of MCTS called Nested Monte-Carlo Search (NMCS) [Caz09]. NMCS estimates the values of states at recursion level k using a recursive call of level $k-1$, where a level 1 recursion is a Monte-Carlo rollout. The best solution found thus far is memorized, which

[1]Available at: http://www.js-games.de/eng/games/samegame

is shown to greatly increase the overall performance of the algorithm. A similar idea is applied in Nested Monte-Carlo Tree Search (NMCTS) [BW12], in which a MCTS, rather than a Monte-Carlo Search (MCS), is performed at each recursive call.

Rosin's Nested Rollout Policy Adaptation (NRPA) combines NMCS with online policy learning [Ros11b]. NRPA's rollouts are guided by a policy, which is slowly adapted towards the moves with the highest return. An efficient encoding of states, which transforms similar states to the same value, is required for the algorithm to perform well. The policy is learned from scratch for each example, and thrown away once the computational budget is reached. Both NMCS and NRPA are computationally expensive, taking several hours per problem instance to run. However, NRPA can be parallelized to mitigate its high runtime cost [Nag19; NC17]. The algorithm has produced record scores in Crossword Puzzle Construction and Morpion Solitaire [Ros11b], and was successfully applied to multiple other domains, such as the Travelling Salesman with Time Windows problem [CT12], Single Vehicle Pickup and Delivery with Time Windows and Capacity Constraints [EG14], and 3D Container Packing [EGR14].

2.2.2 MCTS and Deep Reinforcement Learning

Two-player games

The combination of MCTS and deep reinforcement learning has been successfully applied to two-player games. One work is AlphaZero [Sil+18], which reached superhuman performance in the games of Go, Chess, and Shogi. The algorithm trains the parameters of a single two-headed network, which outputs both a policy and value estimate of a state. The policy is used during the selection step of MCTS, and the simulation phase is replaced by the value estimate. The network is trained from scratch using self-play, without the use of any human generated data. As my algorithm is inspired in part by this work, other details of AlphaZero will be explained throughout the rest of the thesis.

A similar work to AlphaZero is Expert Iteration [ATB17]. Expert Iteration trains a new neural network at each generation, starting with only a policy network, that is, a deep neural network that outputs a policy. The policy network is used in a similar way as AlphaZero, but the training procedure is different. The algorithm trains by playing many quick games by sampling from the current policy, then sampling a single position from each game to evaluate. This method of training is used because moves from the same game are highly correlated, which can reduce the effective size of the training set. Move selection in these positions is then improved by MCTS and the current network by using a large number of random rollouts. Once the generated data is of sufficient quality, the policy network is replaced by a two-headed network, which outputs both a policy and value. The value estimations are then combined with the rollout results using a mixing parameter, further improving the agent.

There also exists work on using linear function approximation, as opposed to deep neural networks, in two-player games. The work of Silver [Sil09] on Go uses linear weighting of features based on local 1×1 to 3×3 patterns; the weights are trained using temporal-difference learning and self-play. The function estimator is then combined with MCTS in various ways, for example, to guide the simulation step.

Single-player games

Laterre *et al.* introduce Ranked-Rewards (R2) [Lat+18], a general algorithm that enables self-play for single-player games. This is accomplished by setting the terminal rewards seen by the agent to a value of either 1 or -1, depending on whether the actual value of the state is higher or lower than a given percentile of previously seen rewards. This pits the agent against itself, forcing it to continuously outperform previous generations. The main drawback of such an approach is that it does not work when games of different difficulties are encountered during training: the agent can perform poorly on a given instance simply because the problem is more difficult than most of the other games encountered by the agent. This can lead to a disproportionately high amount of

-1 rewards in the dataset, i.e. high amount of noise in the training data.

Arfaee *et al.* [AZH11] iteratively learn a heuristic for multiple puzzle games using a neural network and IDA*. In the training procedure, if search is unable to solve a sufficient number of instances in the initial generation, a random walk, starting from the goal state, is used to generate training examples. These states can be solved with the current heuristic, but are just difficult enough to help it improve for the next generation. One downside of the training procedure is that it can create an in-admissible heuristic, which can lead to sub-optimal results.

A work similar to Arfaee *et al.* is [McA+18], in which a two-headed network is trained to solve the Rubik's cube, using a training procedure called Autodidactic Iteration (ADI). ADI is suitable for problems in which there is a single known goal state. The algorithm starts at the goal state and takes k random actions, generating $k-1$ states for training. The value of each state is set to the maximum value found by a one-step look-ahead search, which uses the value output of the network as a heuristic. The action with the highest value is set as the training target for the policy. This training procedure only works for problem domains with reversible actions and a unique, known goal state. This is in contrast to my algorithm, which does not have these requirements.

Chapter 3

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) has become one of the most prominent search techniques in recent years. The algorithm was originally proposed for computer Go [Cou06], and was later used in AlphaGo to beat the world champion Lee Sedol [Sil+16]. Another strength of MCTS is its generality, achieving state-of-the-art performance in general video game playing competitions [BF09b; Per+15; Per+19]. It has also found some success in planning domains [BW12]. In fact, it is a viable choice for any problem that can be described as a Markov Decision Process (MDP), and for which a model exists.

MCTS combines Monte-Carlo simulations and tree search into a single algorithm. Therefore, Monte-Carlo search is introduced first, followed by MCTS. An overview of common enhancements to MCTS is also covered, with emphasis on modifications typically used for single-player games. The chapter is finished by covering several parallelization methods for MCTS.

3.1 Monte-Carlo Search

In Monte-Carlo Search (MCS), the average result of all rollouts starting with action a in state s are used to estimate the state-action value $Q(s, a)$. Rollouts can select actions uniformly at random, or sample from an informed policy. A rollout is completed when a pre-determined number of actions are taken, or a terminal state is reached. If it is cut short, then the value of the final state in the rollout needs to be estimated by a heuristic function $h : \boldsymbol{\mathcal{S}} \rightarrow \mathbb{R}$.

3.2 Algorithm

MCTS has four distinct stages, with a single iteration of the algorithm going through all four stages. Figure 3.1 provides a visual representation of MCTS.

1. **Selection**: starting from root $\boldsymbol{R}$ of the search tree, recursively select the next action based on some strategy. Stop once a node $\boldsymbol{C}$ that is not part of the tree is reached.

2. **Expansion**: add the missing node $\boldsymbol{C}$, or several missing nodes, to the search tree.

3. **Simulation**: starting from $\boldsymbol{C}$, run a simulated playout for some steps, or until a terminal state is reached.

4. **Backpropagation**: update the statistics of the nodes from $\boldsymbol{C}$ (inclusive) to $\boldsymbol{R}$ using the simulation result.

Note that the simulation step of MCS is stage 3 of MCTS, and the value backup is one step of stage 4. The main difference between the two algorithms is that MCTS additionally creates and expands a tree as the search progresses. Additional information about states and actions below the root can be stored in the tree. Therefore, the search budget can be spent in more "promising" sections of the search space, which is defined by the selection strategy.

The addition of a tree adds extra memory and computational costs to the algorithm, as every node and edge in the tree needs to be in memory. The state is typically stored in the nodes, and search statistics are stored in the edges. The size of the state is domain dependent, and can potentially be very large. The traversal of the tree can also become expensive, as the selection value of each outgoing edge at the current node needs to be calculated before an action is chosen. However, for a large number of tasks in a variety of domains, the added benefits of the tree far outweigh the cost; MCTS is used far more often than MCS, even when there is a very limited time and/or memory budget.

MCTS provides a general framework where each part can be specialized and tuned. I will discuss each stage in turn, providing a more detailed explanation.

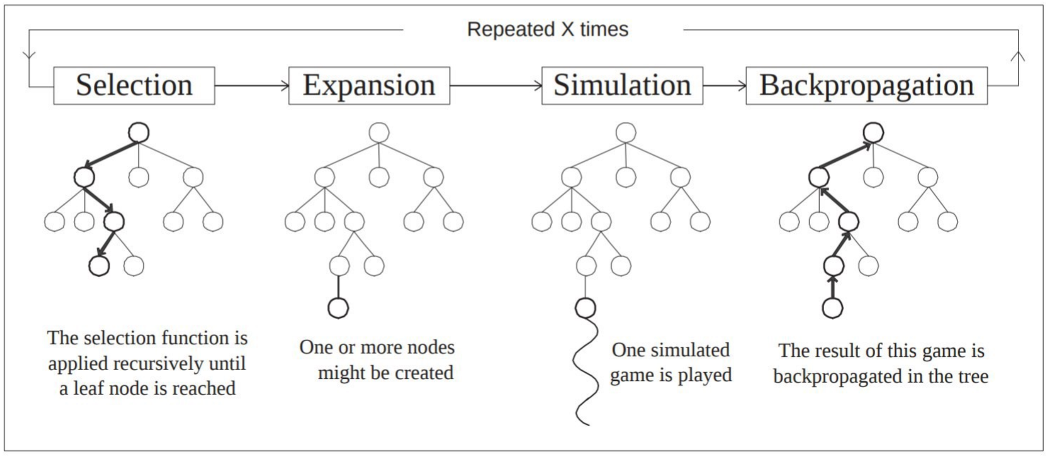

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3.1: A visual representation of the four stages of MCTS [CWD08].

Additionally, I cover some of the related work present in the literature for applying MCTS to single-player games.

3.2.1 Selection

The selection strategy determines how the search tree is traversed, which in turn affects how the tree grows. Any successful selection strategy needs to balance exploration and exploitation. Exploitation refers to taking the currently best action according to the state-action value estimates, whereas exploration is selecting a non-optimal action. A strategy that is too exploratory will not find good solutions in a reasonable amount of time, whereas one that is too exploitative will get stuck in locally optimal solutions. The most commonly used selection strategies are adapted from solutions to the multi-armed bandit problem.

Multi-Armed Bandit Problem

A closely related problem that explores the exploration-exploitation dilemma is the *multi-armed bandit problem*:

> A gambler stands in front of a slot machine with k arms, with the goal of making the most amount of money by pulling the arms. Nothing is known about the arms; he must maximize his return

while finding the arm with the highest expected value. How should he select which arm to pull?

A commonly used performance metric for this problem is the total expected regret, defined for any fixed turn T as [KP14]:

$$R_T = \sum_{t=1}^{T} (\mu_* - \mu_{a_t}), \tag{3.1}$$

where a_t is the arm pulled at time t, μ_{a_t} is the expected reward of pulling the arm corresponding to a_t, and μ_* is the expected reward of the optimal arm. More plainly, it measures the difference, in expectation, between pulling the best arm T times and the arms that were actually selected. Most selection algorithms for the multi-armed bandit problem provide some theoretical guarantees on the growth of expected regret. The lower bound on the growth of regret is proven to be in $\Omega(\log T)$ [LR85]. An algorithm *solves* the multi-armed bandit problem if the growth of its expected regret is in $O(\log T)$.

One approach to the bandit problem is Upper Confidence Bounds (UCB). UCB algorithms are "optimistic in the face of uncertainty", that is, they produce an optimistic guess to the expected payoff of each action; An upper bound $U(a)$ is calculated for each action a, such that $\mathbb{E}[a] \leq \hat{\mathbb{E}}[a] + U(a)$ with *high* probability, where $\hat{\mathbb{E}}[a]$ is an estimate of the expected value of a. The UCB value of an action is decreased as the number of times the action is taken is increased; confidence in the estimate of an action grows as it is taken more times, therefore, the UCB term is decreased.

Upper Confidence Bound 1 (UCB1)

A well known implementation of UCB is UCB1 [ACF02]. The UCB1 value of an arm a is defined as:

$$\text{UCB1}(a) = Q(a) + \sqrt{\frac{2 \log t}{N(a)}}, \tag{3.2}$$

where t is the current time step, and $N(a)$ is the number of times action a has been taken. The addition of $\log t$ ensures that all arms will be continuously explored, i.e. UCB1 will never stop pulling any particular arm. A requirement for this algorithm is that each arm is pulled at least once before using the above formula. Additionally, to simplify the problem, UCB1 works under the assumption that the reward values of each arm are in the range $[0, 1]$; UCB1 can handle other bounds if the value standard deviation of each arm is estimated in some way. The regret of UCB1 grows logarithmically in the number of actions taken [ACF02], i.e. it solves the bandit problem.

Upper Confidence Bound for Trees (UCT)

One of the most popular MCTS algorithms is Upper Confidence Bound for Trees (UCT) [KS06]. UCT treats action selection at each node of the tree as a multi-armed bandit problem, with UCB1 as the selection rule. The UCT value of an action a with parent state s, denoted $\mathrm{UCT}(s, a)$, is defined as:

$$\mathrm{UCT}(s,a) = Q(s,a) + C\sqrt{\frac{\log N(s)}{N(s,a)}}, \tag{3.3}$$

where $N(s)$ is the total visit count for state s, and $N(s, a)$ denotes the number of times action a was taken in state s. At each node in the tree, the edge corresponding to the action with the maximum UCT value is chosen, $a = \mathrm{argmax}_{a'}(UCT(s, a'))$. There is an introduction of a parameter C, which increases exploration or exploitation, with higher and lower values, respectively. Note that $C = \frac{1}{\sqrt{2}}$ reduces the UCT selection strategy to UCB1. Kocsis *et al.* proved that, as t goes to infinity, the root value estimates converge to the optimal values, assuming the same requirements as UCB1.

UCT has been successfully applied to a wide variety of game domains [Bro+12]. These include two-player perfect information board games such as Go [Gel+12] and Hex [AHH10; Hua+13], real-time domains such as Ms. Pac-Man [PWL14] and strategy game combat [BF09a], imperfect information card games such as Skat [SBH08] and Hearts [Stu08], single-player games such as SameGame [Sch+12], and general video game playing [Per+15]. There are more successful applications than are listed here, with new ones published each year. It is important to note that all of these works combine variants of UCT with other improvements to MCTS, sometimes making changes to all four stages of the algorithm.

Predictor + Upper Confidence Bound for Trees (PUCT)

A recently introduced variant of UCT is Predictor + Upper Confidence Bound for Trees (PUCT). PUCT is used in AlphaGo [Sil+16], which defeated the Go world champion Lee Sedol. It is also used in proceeding works of AlphaGo Zero [Sil+17] and AlphaZero [Sil+18], and is the selection strategy used in all of my algorithms. The PUCT algorithm is derived from Predictor + Upper Confidence Bound (PUCB) [Ros11a].

PUCB is another solution to the multi-armed bandit problem. It works under the assumption that a predictor is available before the first arm is pulled. The predictor guides the gambler by placing a higher weight on certain arms than others. Through successive episodes, the predictor is improved by using information from previous runs, leading to better results.

The PUCT value of an edge corresponding to action a with parent state s, which is denoted $\mathrm{PUCT}(s, a)$, is given by:

$$\mathrm{PUCT}(s, a) = Q(s, a) + CP(s, a)\frac{\sqrt{N(s)}}{1 + N(s, a)}. \tag{3.4}$$

The $\sqrt{N(s)}$ term serves a similar purpose as the $\log N(s)$ term in UCT: to ensure all actions are continuously explored. In PUCT, there is the addition of the predictor's value $P(s, a)$, which has a large influence on the UCB value of an action. Furthermore, the requirement that every action must be tried at least once is removed. Thus, the search is focused on a smaller subset of actions; initially, most of the search budget is spent on actions with high prior probability and low visit count, but, asymptotically, actions with high action-value are preferred instead.

Any function $P : \boldsymbol{\mathcal{S}} \to [0, 1]^{|\boldsymbol{\mathcal{A}}|}$ can serve as the predictor. Note that the optimal predictor will place a weight of 1 on the optimal action, and 0 everywhere else. Given that the predictor ideally becomes more confident about stronger actions after each episode, it is natural to use a machine learning method. AlphaZero and similar work use a neural network that, given a state as input, outputs move probabilities. The network is trained over thousands of iterations, becoming progressively stronger, with the final trained network surpassing human skill in several games.

Other Modifications

The general approach to modifying the selection strategy is to start with UCT and introduce additional terms. There are various modifications present in literature [Bro+12]. I describe several modifications that have been applied to single-player domains next.

The main idea behind the selection strategy of Single-Player Monte-Carlo Tree Search (SP-MCTS) [Sch+12] is to also consider a standard deviation of the value of an action. In deterministic single-player domains with rewards outside of $[0, 1]$ range, taking a possible standard deviations into account can give a better estimate of the upper confidence bound, which helps preserve the optimistic nature of the algorithm. The selection strategy is further modified by adding a fraction of the maximum value of an action. Then, the selection strategy picks the action with the highest UCT-SP value, given by:

$$\text{UCT-SP}(s,a) = Q^W(s,a) + C\sqrt{\frac{\log N(s)}{N(s,a)}} + \hat{\text{SD}}(s,a). \tag{3.5}$$

The standard deviation estimate, denoted $\hat{\text{SD}}(s,a)$, is as follows:

$$\hat{\text{SD}}(s,a) = \sqrt{\frac{\sum_{i=0}^{N(s,a)} R_i(s,a)^2 - N(s,a) \times Q(s,a)^2 + D}{N(s,a)}}, \tag{3.6}$$

where $R_i(s,a)$ is the ith reward propagated to the edge corresponding to action a. The extra parameter D is introduced so that nodes with low visit counts are still considered promising. Additionally, the first term is changed from the average value estimate to the sum of the average value and a percentage of the highest reward that has been propagated to the edge:

$$Q^W(s,a) = Q(s,a) + W \cdot Q_{max}(s,a). \tag{3.7}$$

The idea of using the max along with the average appears in several other works as well. An example includes playing Mario with MCTS [JGT14], in which a mixing parameter λ is used to combine the two values

$$Q^\lambda(s,a) = \lambda Q_{max}(s,a) + (1-\lambda)Q(s,a). \tag{3.8}$$

By increasing λ, they observed that the agent became more courageous, and decreasing it lead to more risk averse behaviour.

3.2.2 Expansion

The expansion strategy determines the number of nodes added to the tree in every iteration of the algorithm. One approach is to add the missing node $\boldsymbol{C}$ every iteration. A downside of such a method is that it can quickly consume large amounts of memory. A common remedy to this problem is to only expand a node once it has been visited k times, i.e. a node is expanded only when the algorithm is relatively confident about the success of the move leading to it. However, when using a policy or a two-headed network, any new node that is explored by the search is considered promising. Therefore, in such scenarios, a node is often expanded on the first visit.

In single-player domains, when a new best solution is discovered, an effective expansion strategy is to expand a node for each visited state in the solution. [HI15; ŚMO15]. This corresponds to memorizing the current best sequence of actions. This is an effective strategy because, in such domains, all values found during search are a lower bound on the optimal value.

3.2.3 Simulation

The simulation policy can have a large impact on the effectiveness of MCTS. One possible policy is to assign equal probabilities to all valid actions in a state, i.e. no bias is introduced in the selection of actions. This policy is often not practical however, as the amount of search time is limited. Instead, most successful applications of MCTS use an informed policy to guide the rollouts [Bro+12]. A policy can be handcrafted by a domain expert, learned by MCTS while searching (see NRPA in Section 2.2), or learned from a labeled dataset using supervised learning [Sil+16], which is the approach taken in this thesis (see Chapter 5).

Hand-tuned policies are still prevalent for most applications. This is because people that work on a certain application often have useful domain knowledge they can pass onto the system. For example, the application of

SP-MCTS to SameGame uses two novel hand-crafted policies to guide the rollouts. These were presumably developed as the authors became more comfortable with the domain while working on their research. The main downside of using handcrafted policies is that they are often crude and biased approximations.

An informed policy can be used to guide the rollouts by selecting the greedy action with respect to the policy output in each state, i.e. always choosing the action with the highest probability. A more robust approach is to sample from the policy instead, which increases the diversity of the rollouts, often leading to better results.

In addition to a rollout policy, heuristics can be used to estimate values of states encountered during the rollout. A heuristic allows the rollouts to be cut short, for example, after reaching a certain depth, which can drastically speed up the simulation stage of the algorithm. Such approaches leverage the fact that a heuristic is often more accurate when used in states that are deeper in the search space, since they will be closer to terminal states. But, as a heuristic becomes more accurate, the number of actions in a rollout can be further reduced. In fact, given a strong enough heuristic, rollouts can be entirely replaced, as done in AlphaZero.

3.2.4 Backpropagation

The statistics propagated back to the root are ones that are required by the other stages of the algorithm. In UCT, the statistics stored in the edges are updated using the rollout result R, as follows:

$$\begin{aligned} N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1, \\ Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \frac{R - Q(s_t, a_t)}{N(s_t, a_t)}, \end{aligned} \tag{3.9}$$

where s_t and a_t denote the state, and the action taken at time t in the selection stage, respectively. When more information is required, additional updates may be necessary. For example, in SP-MCTS, Q_{max} needs to be tracked as well, so an additional update is performed:

$$Q_{max}(s_t, a_t) \leftarrow \max\left(Q_{max}\left(s_t, a_t\right), R\right). \tag{3.10}$$

3.3 Parallelization

A natural way to improve any search algorithm is to implement a parallelized version, which helps the algorithm scale with the available computational resources. Ideally, a parallel version using k threads and given 1 second of computation time incurs no loss when compared to the single-threaded version given k seconds of computation time, i.e. performance is not sacrificed when using additional threads. This cannot be achieved by any known parallelization technique for MCTS, as there is always a trade-off between speed gained and information lost. Several parallelization strategies are discussed next, with a visual representation of these methods provided in Figure 3.2.

3.3.1 Leaf Parallelization

The simplest stage of MCTS to parallelize is the simulation step. The simulation stage starts with the thread selecting an action at the leaf node. A simple parallelization strategy then, is to assign a legal action to each available thread and perform all rollouts in parallel. This is called *leaf parallelization* [CJ07]. In this scheme, all other stages of MCTS are done by a single thread.

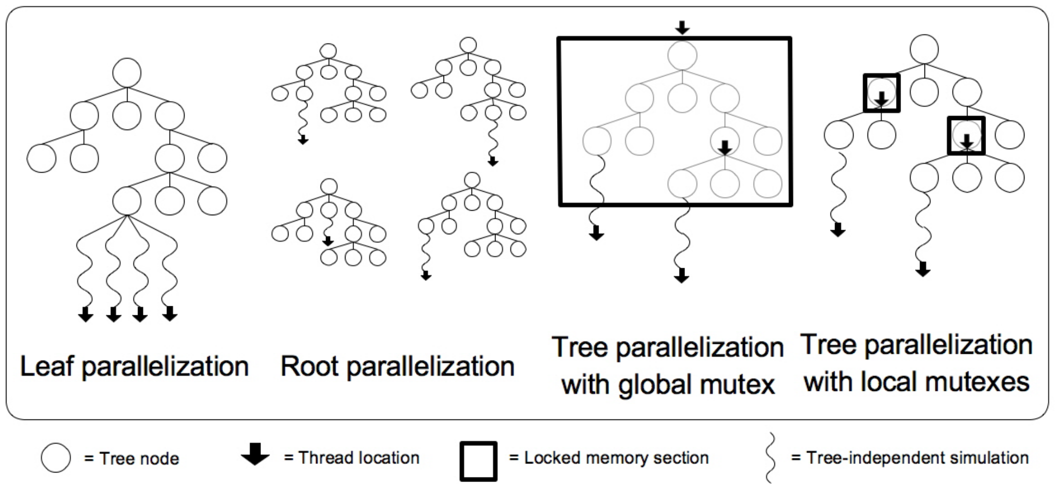

Leaf parallelization
Root parallelization
Tree parallelization with global mutex
Tree parallelization with local mutexes
= Tree node
= Thread location
= Locked memory section
= Tree-independent simulation

Figure 3.2: A visual representation of different parallelization strategies for MCTS [CWD08].

This method is simple to implement, but it has two main drawbacks.

Since the backpropagation stage is done by a single thread, the algorithm has to wait until all the rollouts are finished. The amount of time taken per rollout can vary widely, even if all the rollouts are started at the same state, especially if they are random. Therefore, multiple threads can finish quickly, while others can take a long time. The main weakness of this strategy is that it cannot proceed to the backpropagation step until the slowest thread is finished. An improvement that reduces the overall wait time is to stop all rollouts once a given number of rollouts produce near identical results. For example, when half the rollouts return the exact same value.

The second weakness of this strategy is the fact that it only allows for the parallelization of the simulation stage. Therefore, the effectiveness of leaf parallelization is bounded by the time spent on the other three stages of MCTS.

3.3.2 Root Parallelization

In *root parallelization* [CJ07], each thread runs an independent instance of MCTS in parallel. Once the time limit is reached, all trees are combined into a single tree. Clones of nodes and edges are merged, with the new tree containing the statistics of all the clones. Then, the best action is selected in the merged tree.

A major strength of this parallelization scheme is that no communication is required between separate threads until the merging process. The lack of communication also encourages exploration, with the ratio of exploration to exploitation increasing with the number of threads. This is not necessarily good, however, as exploitation is often required when given a limited time budget. One drawback of root parallelization is that, since only one thread works on a tree, the depths of the search trees will be similar to that of the single-threaded version.

3.3.3 Tree Parallelization

A limitation of both leaf and root parallelization is that neither have more than a single thread working on the search tree at a time. In contrast, in *tree* *parallelization* [CWD08], all threads cycle through the four stages of MCTS using the same search tree. Since memory is shared between all threads, the same memory location can be read and written on by multiple threads at the same time. Therefore, parallel access to a particular section of memory needs to have defined behaviour. One solution to this issue is to use locks, which are often referred to as *mutexes*. The two types of mutex implementations introduced in [CWD08] are a *global mutex* and *local mutexes*.

A global mutex limits access to the tree to a single thread at a time. When a thread enters any stage that requires the tree (Stages 1, 2, and 4), it has to wait until it can acquire the lock. Once the simulation stage is started, it will release the lock, allowing another thread to enter the tree. Such a strategy ensures that no data corruption occurs, but it does so at a large cost; this strategy works best in domains in which rollouts take up the majority of the search time. A global mutex is not suitable when a high number of threads can be used, as most of the threads will spend the majority of the time trying to acquire the tree lock.

A more scalable approach to tree parallelization is to use local mutexes at each node and/or edge. This strategy allows for a large number of threads to concurrently traverse the search tree. But it pays a higher overhead price of creating these mutexes. Every thread also has to lock and unlock a mutex for each step of the selection and backpropagation stages. Given a large number of threads, when combined with virtual loss, this strategy has been experimentally shown to perform the best in two-player settings [Seg10]. An alternative approach to local mutexes is a lock-free implementation which uses atomic operations to change statistics [EM09]; the majority of high-performance implementations of tree parallelization are lock-free.

3.3.4 Virtual Loss

When using tree parallelization, one can expect most threads to take the same path down the tree. This is because UCT and related selection strategies have no stochasticity. The diversity of paths can be increased by incrementing visit counts as actions are selected, rather than during the backpropagation step; since UCB values will change while threads are traversing the tree, each thread might go down a different route. However, this modification is not enough when using a high number of threads.

Virtual loss is one solution to this problem [CWD08]. As each thread traverses the tree, it increments the visit counts and also adds a temporary *loss* to the score of each selected edge. This discourages other threads from taking the same route in the tree. During the backpropagation step, the loss is removed. Using virtual loss also encourages exploration, as threads will be discouraged from taking actions that they might have otherwise chosen.

To provide a better explanation, I explain the process through the four stages of MCTS next. A thread enters and traverses the tree using the selection strategy until it reaches a leaf node. Every time an edge is selected, the thread will increase both the visit count and the virtual loss. Once a leaf node is reached, a new node is expanded and added to the tree, and the simulation stage is started. During the backpropagation step, the virtual loss of all visited edges is decreased.

The virtual loss value becomes a parameter of the algorithm, with higher values increasing the rate of exploration. In two-player zero-sum games, the value corresponds to the number of virtual rollouts that resulted in a loss. This approach will not work in a single-player setting with unknown rewards however, as a "loss" is undefined; I will introduce a virtual loss function suitable for such domains in the following chapter.

Chapter 4

MCTS for Single-Agent Optimization

In this chapter I describe my MCTS algorithm in detail. To create an effective MCTS algorithm for single-agent optimization, I introduce a value normalization technique, as well as define a virtual loss function, which enables effective tree parallelization. It is important to note that the purpose of my MCTS algorithm is to be combined with a policy network (Chapter 5) and a two-headed network that outputs both a policy and value estimate (Chapter 6). However, since both the value normalization mechanism and the virtual loss function introduced in this chapter are applicable to any domain with unknown rewards, I study them in isolation without using any networks.

4.1 Value Normalization

One of the most common variant of MCTS is UCT (and its variants). Given infinite time, UCT has been proven to converge to an optimal solution, but it requires all rewards to be scaled to a bound of $[0, 1]$ [KS06]. This requirement is often not a problem in adversarial zero-sum two-player games—the domain in which UCT is most widely used—as the bound on rewards is often known, and, therefore, can be used to normalize the values into the required range. However, in the majority of single-agent optimization tasks, the range of rewards is often different, and unknown. Therefore, a robust normalization strategy is required for UCT and its variants, such as PUCT, to converge to an optimal solution.

4.1.1 Related Work

Several methods for dealing with unknown rewards have been introduced in previous work. In SP-MCTS, the values are not normalized. Instead, larger parameter values are used in the selection strategy (Eq. 3.5), increasing the UCB values of actions. The main limitation of such an approach is that it is domain specific, as it requires either extensive domain knowledge or experiments to track the standard deviations of average values to pick suitable values for C and D.

Klein [Kle15] uses a constant term to normalize the values. This method is called *static normalization*, which is given by the following equation

$$Q_{norm}(s,a) = \frac{Q(s,a)}{M}, \tag{4.1}$$

where $Q(s,a)$ is the average of encountered values and M is the normalization constant; Klein uses $M = 5000$ for his experiments on SameGame. This normalization strategy leads to early exploration and late exploitation, as the UCB term will dominate until higher values are discovered, which can be a desirable property. However, there are multiple limitations to this normalization strategy. The first is as described above for SP-MCTS. The second limitation is that the selected value needs to be good on average, meaning that it may not perform well on a potentially large number of instances.

Another normalization mechanism, called *space-local value normalization*, is introduced by Vodopivec *et al.* [VSS17]. In this method, the highest and lowest action values encountered in state s are used to normalize the values locally at the node level, as follows:

$$Q_{norm}(s,a) \ = \ \frac{Q(s,a) - \min_{a'} Q_{min}(s,a')}{\max_{a'} Q_{max}(s,a') - \min_{a'} Q_{min}(s,a')}, \tag{4.2}$$

Where $Q_{min}(s,a)$ and $Q_{max}(s,a)$ denote the minimum and maximum values found by taking action a from state s. When the maximum and minimum values are equal, i.e. the denominator is 0, the global maximum and minimum values are used instead. If the global bounds are also undefined, then normalization cannot be performed, and the un-normalized value is returned. This linear transformation will normalize the state-action values to an interval of $[0, 1]$, which keeps the convergence guarantees of UCT. Such an approach is analogous to dynamically changing the exploration parameter C of UCT at each node. Since the average state-action value $Q(s, a)$ is normalized by the maximum and minimum value encountered at each node, the normalized values will rarely be exactly 0 or 1.

4.1.2 Max-Min Normalization

My normalization strategy, called *max-min normalization*, scales values to a range of $[0, 1]$, and is applied locally at the node level as follows

$$Q_{norm}(s, a) \;=\; \frac{Q(s, a) - \min_{a'} Q(s, a')}{\max_{a'} Q(s, a') - \min_{a'} Q(s, a')}. \tag{4.3}$$

When max and min values are not yet defined, or they are equal, $Q_{norm}(s, a)$ is set to 1. This is being "optimistic", as a value of 1 means that the action has a higher value (or is tied) than all other possible actions from state s. The highest and lowest Q-values can be stored in a node directly, or calculated by looping over all edges.

In contrast to static normalization, this method does not require any knowledge about the highest and lowest achievable values in the domain. Furthermore, the tracking of standard deviations to relate average values and UCB exploration terms is no longer required. Lastly, since state-action values are defined locally, action values in one node do not have an impact on the normalized values in other nodes.

Note that max-min normalization maximizes the spread of mapped action values: there will always be one action with a value of 1, and after several simulations, at least one action with a value of 0. Assuming a scenario in which the maximum and minimum values are very similar, and the reward bounds are known, max-min normalization can lead to a higher level of exploitation than standard UCT [KS06]. One method for reducing the spread of mapped action values is to use the actual maximum and minimum instead, as done in space-local value normalization. Since both methods normalize values to range $[0, 1]$, UCB-based selection strategies will converge to an optimal solution.

A drawback of max-min normalization strategy is that it is more expensive to compute, especially when combined with tree parallelization. This is because during the selection phase, values of upcoming edges could be changed by other search threads, leading to new upper and/or lower bound values. When this occurs, the selection phase has to be restarted at the node.

Max-min normalization is different to space-local normalization in two ways. The first is that space-local value normalization uses the maximum and minimum values found, in contrast to using the average values. My experiments suggest that the difference between these two approaches is minimal, assuming the exploration parameter C is set appropriately (Subsection4.4.1). Secondly, when the local bound is undefined, max-min normalization returns a normalized value of one for all actions, i.e. the next action is selected according to the UCB value. Compared to using the global bound, this may result in better performance when an informed policy is used in the calculation of the UCB value.

4.2 Tree Parallelization and Virtual Loss

In the next two chapters, I combine MCTS with deep neural networks. The network will need to make many predictions during search, which can lead to a large decrease in the speed of the algorithm. The slowdown caused by the network can be reduced by predicting in batches; a single prediction on a batch of size k takes far less time than k predictions of size 1. There are multiple ways to combine search with batch predictions. The simplest solution is to execute many independent runs at the same time. A downside of this approach is that the algorithm will still be slow when executing a single run. This can be remedied by parallelizing MCTS using one of the strategies discussed in Section 3.3.

My MCTS algorithm uses tree parallelization combined with virtual loss, which is also the parallelization strategy used in AlphaZero [Sil+18]. In domains in which virtual loss is typically applied, such as the game of Go, virtual loss corresponds to virtual rollouts that resulted in a loss. However, in this setting, the bounds on the score are unknown, so a definition of "loss" is required. I define virtual loss as

$$L(s, a) = wW(s, a)Q_{norm}(s, a), \tag{4.4}$$

where $W(s, a)$ is the virtual loss count stored in the edges of the tree, and w is the global virtual loss weight, which is subject to optimization; I explore the impact of different values for w in Section 4.4. In the above definition, the loss is relative to the current normalized state-action value. This induces a higher amount of exploration than using a constant loss.

The main benefit of this approach is that it requires no knowledge about the bound of rewards, and thus, is more general than the virtual loss function used in zero-sum two-player games; it can also be used in that setting, but will likely not perform as well due to the adversarial nature of such games.

4.3 Stages

The four stages of my MCTS algorithm are as follows.

1. Selection. At each node, the action a with the highest sum of average value and upper confidence bound, corrected for virtual loss, is selected:

$$a = \underset{a'}{\operatorname{argmax}} \left(Q_{norm}(s, a') - L(s, a') + U(s, a') \right), \tag{4.5}$$

where the upper bound $U(s, a)$ is given by PUCT, which is calculated as follows:

$$U(s, a) = C\pi_{uniform}(s, a) \frac{\sqrt{N(s)}}{1 + N(s, a)}. \tag{4.6}$$

Even though the plain MCTS does not use a policy network, I still use the PUCT selection strategy with a policy that is uniform over valid actions. This is analogous to using UCT with C conditioned on the number of actions. A uniform policy encourages exploration in states in which a low number of actions are possible, and conversely, encourages exploitation in states with a high number of legal actions. An alternative approach is to assign a value of 1 to each action. This would require the optimization of two different C values however, as the UCB values (excluding C) will be different, potentially by a large amount, when a policy is used.

When actions with equal value are encountered, the standard approach is to break ties uniformly at random. However, I discovered through experiments that random tie-breaking is not as effective in SameGame as biasing the selection towards the most left, and lower group of blocks, respectively. This bias helps focus the search, which allows for the creation of deeper trees, often leading to better results. Note that this is a relatively small detail as ties are a rare occurrence when an informed policy is used.

Once an edge has been selected, the visit count is updated. This is to ensure that each thread has the most current information while applying the selection strategy. In addition to the visit count, each edge also stores a virtual loss count $W(s, a)$, which is also updated during the selection stage:

$$W(s, a) \leftarrow W(s, a) + 1,$$
$$N(s, a) \leftarrow N(s, a) + 1.$$

2. Expansion. A leaf node n_L is expanded on the first visit and added to the tree. All child edges, one per legal action a, are created and initialized as follows:

$$\{N(s_L, a) = W(s_L, a) = Q(s_L, a) = Q_{total}(s_L, a) = 0\}, \tag{4.7}$$

where s_L is the state of node n_L, and $Q_{total}(s_L, a)$ is the sum total of all rewards seen by taking action a in state s_L. The prior probability is also stored in the edge, which is set to uniform over all valid actions.

3. Simulation. Multiple simulation strategies are possible; I use the following alternative. The first action in a rollout is selected uniformly at random from the newly expanded node n_L. Each subsequent action is also selected uniformly at random among all valid actions. The rollout is finished once a terminal state is reached.

After the first rollout is finished, the initial value of all unvisited edges is set to the result of the rollout R: $Q(s_L, a) = R$ for all legal actions a from state s_L. This gives the other edges the opportunity to be selected independently of the success of the first simulation. Other initialization strategies, such as optimistic initialization by setting the normalized value of unvisited edges to 1, might work better for plain MCTS. However, I keep the previously described method to be consistent with the MCTS algorithm used in the next two chapters.

If a simulation finds a sequence of actions with a higher score than the current best, the solution is kept in memory so it can be reported once the search is finished. Memorization of the best solution has been shown to greatly improves the performance of MCTS algorithms in single-player domains [Caz09].

4. Backpropagation. The result of the rollout R is propagated to the root, and the statistics of each edge along the way is updated as follows:

$$\begin{aligned} Q_{total}(s,a) &\leftarrow Q_{total}(s,a) + R, \\ Q(s,a) &\leftarrow \frac{Q_{total}(s,a)}{N(s,a)}, \\ W(s,a) &\leftarrow W(s,a) - 1. \end{aligned}$$

4.3.1 Implementation Details

There are multiple ways to implement MCTS with tree parallelization and virtual loss. While virtual loss discourages threads from traversing the same path in the tree, it does not prevent it. Thus, the implementation needs to have defined behaviour when multiple threads select the same path. I use mutexes to avoid memory corruption.

All nodes and edges have a unique mutex. A node is locked when creating its edges (expansion) and checked if it is locked before starting the selection phase. This prevents a thread from starting the selection process at a node that is not yet fully expanded. An edge is locked when calculating the action value (selection), updating the visit and virtual loss count (selection), adding a child node (expansion), and updating the statistics (backpropagation). If multiple threads choose the same action from a leaf node, only one thread expands and starts its simulation from that node; the other threads restart their selection stage at the newly expanded node.

It is possible that while a thread is in the selection stage at a particular node, another thread updates the value of an edge such that it becomes the new maximum or minimum. If the value of this edge has not been calculated yet, then the selection stage is restarted at the node. Otherwise, the update has no impact on the thread.

4.4 Experiments

In this section, I will provide empirical results justifying parameter choices in my algorithm. All experiments use a test set of 500 randomly generated 15×15 boards. 5 runs are done per board, for a total of 2500 runs, with each run given 1 second. Multiple runs are executed on a board because of the variance in achievable scores across different boards. 1 second of search time was discovered to be enough time to be able to differentiate between different parameter values. All p values for statistical significance were acquired using paired-samples, two-sided t-tests.

4.4.1 Normalization with Average vs. Real Values

I will experimentally evaluate the difference between max-min normalization using average values and actual values seen during search, as done in space-local value normalization. I wish to explore if the increased exploitation when using average values has a significant impact on the results. Five different values of $\{0.1, 0.5, 1, 2, 3\}$ are tried for the exploration parameter C for both normalization strategies. The CPU used for these experiments is an Intel i7-7700K CPU @ 4.20GHz. The result is provided in Figure 4.1.

The results provide some evidence that using average values is more exploitative than using real values: when using real values to normalize, lower values of C perform better. In contrast, higher C values are more effective when normalizing using average values. A conclusion can be made that both normalization methods can work, but each require a different value for the exploration parameter C.

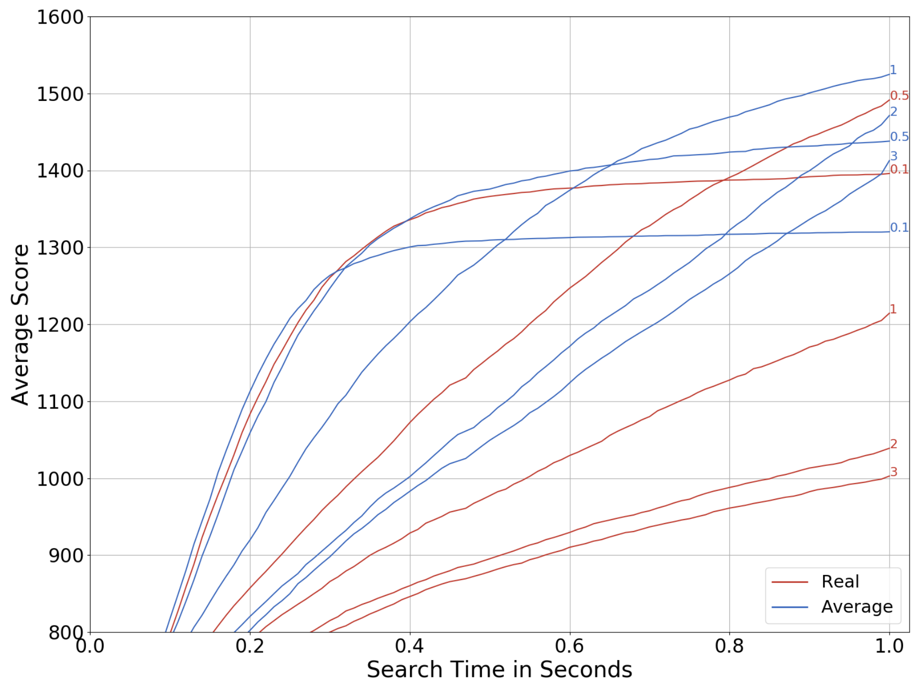


Figure 4.1: Comparing the performance of MCTS with max-min normalization using average or real values.

4.4.2 Virtual Loss

The virtual loss function introduced in Eq. (4.4) has a loss weight parameter w. Lower values for w will cause threads to take similar paths down the search tree. Increasing w will have the opposite effect, with threads taking relatively more explorative paths. The same can be said of the number of threads, with higher number of threads causing more exploration.

In the following experiment, I will empirically optimize the parameter w. This hyperparameter is highly correlated with the number of threads. Therefore, I test each value of w with k threads, where $k \in \{2, 4, 8, 16\}$. Additionally, I run the single-threaded version of MCTS, and give it k seconds; multi-threaded MCTS with k threads and 1 second of computation time is compared against single-threaded MCTS with k seconds of computation time, for $w \in \{0.005, 0.01, 0.015, 0.02, 0.025, 0.03\}$. Based on the results from Figure 4.1, parameter C was set to 2, since performance was still improving. Eight

Table 4.1: Comparing the performance of MCTS with k threads and 1 second of computation time against MCTS with 1 thread and k seconds of computation time, for different values of virtual loss weight w. The error values are the 99% confidence interval.

	Threads			
w	2	4	8	16
0.000	0.930 ± 0.018	**0.964 ± 0.014**	0.979 ± 0.013	0.970 ± 0.012
0.005	**0.961 ± 0.019**	**0.964 ± 0.015**	1.001 ± 0.013	**0.981 ± 0.013**
0.010	0.951 ± 0.018	0.945 ± 0.015	**1.010 ± 0.013**	0.953 ± 0.013
0.015	0.916 ± 0.018	0.905 ± 0.015	0.971 ± 0.014	0.924 ± 0.014
0.020	0.926 ± 0.018	0.918 ± 0.016	0.974 ± 0.014	0.904 ± 0.014
0.025	0.934 ± 0.018	0.909 ± 0.015	0.977 ± 0.014	0.870 ± 0.014
0.030	0.919 ± 0.018	0.845 ± 0.016	0.909 ± 0.015	0.860 ± 0.014

cores of an Intel Xeon 6148 @ 2.4 GHz were used for all of the runs; these models do not support hyper-threading. It is important to note that experiments based on real-time are heavily influenced by the computation devices used: the best parameter found by the following runs are for the above CPU only, and will likely change if a different CPU is used. For this reason, all multi-threaded experiments in the rest of the thesis are performed on the same computation devices. The experimental results are summarized in Table 4.1.

There are a few observations that can be made from the results. The first is that using virtual loss improves performance, with $w = 0$ outperformed in all instances; there is a statistically significant difference when using two and eight threads, with $p < 0.01$, and $p < 0.0001$, respectively, but not when using four and sixteen threads. Secondly, using threads equal to the number of cores provides the best performance. This is the case for all but the highest value of w, which performs best given only two threads. There is also no clear winner, with p=0.214 when comparing eight threads with $w = 0.005$ and $w = 0.01$. However, eight threads with $w = 0.01$ outperforms all other values, with $p < 0.0001$ when compared to the third highest value (16 threads with $w = 0.005$).

Multi-threaded MCTS with eight threads and $w \in \{0.005, 0.010\}$ is able to outperform the single-threaded version. This can be due to the increase in exploration caused by using virtual loss. It can also be a fluke, as the confidence interval suggests that they can perform worse as well.

Table 4.2: The ratios of simulations, node expansions, node expansions to simulations, and average value, between multi-threaded MCTS given 1 second of computation time with $w = 0.01$ and k threads, and single-threaded MCTS given k seconds of computation time.

Threads	Simulations Ratio	Node Expansions Ratio	Node Expansions: Simulations	Average Value Ratio
2	0.90	0.93	1.03	0.95
4	0.75	0.92	1.22	0.95
8	0.65	1.02	1.56	1.01
16	0.28	0.77	2.76	0.95

In Table 4.2, I provide further data comparing the performance of multi-threaded MCTS given 1 second of computation time with $w = 0.01$ and k threads and single-threaded MCTS given k seconds of computation time. All ratios are calculated by dividing the multi-threaded by the single-threaded count. The simulation count also contains simulations starting (and ending) at terminal leaf nodes, i.e., terminal states that are part of the search tree.

The data suggests that, as expected, increasing the number of threads also increases exploration, with the ratio of node expansions to simulations increasing with the number of threads. Another observation is that as the thread count increases, the ratio of simulations goes down. This indicates that most threads are taking similar paths in the tree, which creates longer wait times for acquiring the mutexes. The number of simulations can be increased by using a lock-free method with atomic and C++ volatile variables, as done in [EM09]. My initial implementation used mutexes as I found it easier to debug, and due to lack of time, I did not change to a lock-free implementation.

The average value ratios demonstrate that the parallelization method only incurs a loss of 0.05, even with a much lower simulation count. One possible explanation is the increased exploration caused by the virtual loss. The ratio can potentially be decreased by optimizing a different C value for single-threaded and multi-threaded versions. Overall, the results demonstrate that a CPU with k cores can use k threads to decrease the computation time (wall-clock time) by a factor of k, and incur a minimal amount of loss in the process.

Table 4.3: The ratio of the average values achieved by multi-threaded and single-threaded MCTS, with $w = 0.01$ and k threads, with both given 1 second of computation time.

Threads	Average Value Ratio
2	1.27
4	1.56
8	1.74
16	1.70

Lastly, I compare the performance of single and multi-threaded MCTS, with both given only 1 second of computation time, and $w = 0.01$. The results (Table 4.3) suggest that using eight threads provides the highest performance boost. Given the previous results, a conclusion can be reached that eight threads with $w = 0.01$ provides the best performance.

Chapter 5

Policy-Guided MCTS for Single-Agent Optimization

In the game of Chess, there is an average of about 35 legal moves per turn, many of which are weak. To play the game at a high skill level, an agent needs to have an *intuition* about the strength of each move, which allows the narrowing of good moves. This gives the agent more time to look ahead, to predict actions of both the agent and the opponent for several turns into the future.

The agent's intuition can be modeled as a policy $\pi : \boldsymbol{\mathcal{S}} \rightarrow [0, 1]^{|\boldsymbol{\mathcal{A}}|}$, which, given a state, outputs a probability distribution over all actions; actions with higher probability can be interpreted to be more promising. Using a policy in conjunction with a search algorithm allows the search to focus on the few promising moves. That is, the policy reduces the branching factor (width) of the search tree. When combined with MCTS, the policy helps create deep narrow trees, and it can additionally be used to guide the rollouts.

Training such a policy can be a difficult task in complex domains. One approach is to use supervised learning on a labeled dataset of human games. This has two main drawbacks: 1) it requires the existence of such a dataset, and 2) the distribution of states in the dataset can be very different from the distribution of states actually experienced by the agent. Both issues can be addressed by using reinforcement learning: the agent plays various instances of the game to collect and store data, which will be used to train the next policy. Then, the newly trained policy is used by the agent to gather more, higher quality data. Since the dataset is created by the previous agent(s), the distribution of states in the dataset will be similar to the one seen by the current agent.

Using the above training process, a recent development is to combine the policy with a search algorithm to find stronger actions during self-play; this technique is used in AlphaZero [Sil+18] and Expert Iteration (ExIt) [ATB17]. In this process, there is an apprentice and an expert. The job of the apprentice is to generalize the policy of the expert, and the expert is tasked with using the apprentice to find better actions than either could on their own. Given the success of deep neural networks in other complex domains such as Go and Hex, I use a policy network for the apprentice, as well as MCTS for the expert.

The main contribution of my work is introduced in this chapter. The presented algorithm is general and can be applied to most single-player games, assuming the following two conditions: 1) a forward model of the environment exists (required by MCTS), and 2) the state can be used for learning (required by the policy network).

5.1 Integrating Policies into MCTS

In this section, I describe how a policy can be used in conjunction with the MCTS algorithm described in the previous chapter, to create a new MCTS algorithm called *Policy-MCTS*. As Section 4.3 in the previous chapter was dedicated to explaining the base MCTS algorithm, I only go into detail about changes made to each stage.

1. Selection. The policy used in the upper bound $U(s, a)$ calculation is changed from a uniform policy to the output of the policy network:

$$U(s, a) = C\pi_\theta(s, a)\frac{\sqrt{N(s)}}{1 + N(s, a)}. \tag{5.1}$$

2. Expansion. The prior probability $\pi_\theta(s, a)$ is also stored in each edge, which is the re-normalized output of the network after filtering out illegal actions. When a node is expanded, the node is locked and the thread put to sleep until the state is evaluated by the network.

3. Simulation. The edge with the highest prior probability is selected from the newly expanded node n_L as the first action in the simulation. Therefore, the initialization strategy described in the previous chapter becomes "optimistic" under the current policy.

The rest of the simulation can follow any simulation policy; I use the following two alternatives. The first selects an action uniformly at random among all valid actions, which is called *random* rollouts. The second approach, which I call *informed* rollouts, selects an action in state s by sampling from the policy $\pi_\theta(s)$. Such an approach leads to far more informed rollouts, but is much slower than the random counterpart, since a network prediction is required at each step of the simulation; one option for increasing the speed of informed rollouts is to use a faster approximate policy, such as a smaller network trained on the same dataset [Sil+16].

4. Backpropagation. The backpropagation step does not change.

5.1.1 GPU Queue Implementation

The real-time performance of parallel Policy-MCTS is heavily influenced by the implementation of the prediction queue. My queue class works as follows: there are N separate queues, all of which are of size L. The first thread to add to an empty queue starts a timeout timer M. If the queue is filled before the timeout expires, the content of the queue are sent for prediction. Once the thread is timedout, it checks if there are at least N/F items in the queue. If so, it will send the queue over for prediction. Otherwise, it will repeat the process P times, after which it will simply send the queue for prediction, regardless of how many items are in the queue.

The size of the queues L is highly correlated with the number of threads used in Policy-MCTS, as well as the CPU(s) and GPU(s) of the machine; experiments demonstrating the relationship between the queue size L and the number of threads is provided in Section 5.4.3. Some effort was put into optimizing the other parameters to maximize the performance of the GPU: for all multi-threaded experiments, $M = 2ms$, $F = 4$, $N = 8$, $L = 48$, and $P = 1$ were chosen; L is set to 1 when only a single thread is used.

5.2 Training Policies

5.2.1 Policy Training Target

In two-player adversarial games, assuming a limited search time budget for MCTS, the edge with the highest visit count—as opposed to the highest valued edge—is often selected as the action to play (e.g., AlphaZero and ExIt). Actions with higher visit counts are considered more robust, as they guard against the case in which a newly analyzed move with higher value, but much fewer simulations, is over-confidently chosen.

Therefore, it is logical to use a training target for the policy that is based on visit counts, as done in both previously cited works:

$$\pi_{two-player}(s_t, a) = \frac{N(s_t, a)}{\sum_{a'} N(s_t, a')}, \tag{5.2}$$

where s_t is the state of the environment seen by the agent at time $t \in \{1, 2, \ldots, T\}$, and a is an action. That is, the training value of each action is proportional to the number of times that action was selected. By contrast, in single-player games, which are non-adversarial, the values of simulations starting in a state are a lower bound on the maximum achievable value. This means that the action with the highest simulation value is currently the best action, regardless of how often other actions were attempted. In the limit, the action with the highest visit count will also have the highest value [KS06]. However, this might not be the case given a limited search budget.

Considering the above observation, I set the target for the policy as follows, with ties between equal valued actions broken randomly:

$$\pi(s_t, a) = \begin{cases} 1, & a = a_t, \\ 0, & otherwise. \end{cases} \tag{5.3}$$

That is, the training target for state s_t is the action selected at time t. This target aggressively pushes the policy towards the best action found, and thus, is more exploitative than a target based on visit counts (Eq. 5.2).

Using the above policy target, training reduces to a supervised learning task that seeks to minimize the average cross-entropy loss between the target policy π and its approximation π_θ, for all training samples. That is, for a mini-batch of size B, its loss L is given by:

$$L = -\frac{1}{B}\sum_{i=1}^{B} \pi(s_i)^{\mathsf{T}} \log \pi_\theta(s_i) \tag{5.4}$$

5.2.2 Data Generation

In many single-player games, the initial actions have a large impact on the final score the agent can obtain. An intuitive choice, then, is to let the agent spend the majority, if not the entire MCTS budget, on determining the best initial move. This would allow the agent to come up with the best first move it possibly can. However, once the search budget is spent, the entire sequence, and not just the first move, is reported. Since the entire search budget has been spent, there is no opportunity to optimize the rest of these moves further.

It has been shown in [Sch+12], as well as confirmed by my own experiments, that committing to an action after a fraction of the search budget is spent, thereby allowing the search to optimize the remaining move sequence, can produce better results. Committing to actions corresponds to playing a game; once a decision has been made, it cannot be reversed. This allows MCTS to spend more time in deeper sections of the search tree. Therefore, actions near the middle and end of the game receive more of the search budget than they would otherwise, making the resulting action sequence better.

Inspired by this observation, my agent interleaves planning and playing. In each state, the agent receives a constant planning budget of k MCTS simulations. Once the planning budget is spent, the best action, which is the one with the highest value, is taken. Although this forces the agent to commit to earlier actions, it opens up the opportunity to better optimize subsequent moves. This process is repeated until a terminal state is reached. Then, all state-action pairs taken to reach the final state are returned, to be used as training data.

One downside of assigning k simulations to every move is that it assumes the same decision complexity in all stages of the game. In reality, this is rarely the case. For example, the last several moves in SameGame are often trivial. Thus, depending on the domain, choosing a better simulation budget allocation is likely possible, but this is outside of the scope of this thesis, and is not explored further.

Once a move is taken, the search tree is reset. I experimentally tested between keeping and removing the section of the tree that can still be used, and discovered that resetting the tree performed better. One possible explanation is that resetting the tree induces additional exploration, as it removes the bias from the previous search. Note that the action sequence leading to the highest score is memorized, and is not reset.

While training, Dirichlet noise is added to the prior probability of all actions a at the root of the tree. The main purpose of the noise is to give all legal actions a non-zero probability of being selected. It can also increase the UCB value of actions the current policy believes to be bad by a non-trivial amount, which encourages exploration. Noise is added to the root exclusively because only the action taken at the root is stored for training. When noise is added, the prior probability stored in each edge at the root of the tree becomes:

$$\pi_\theta(s_{root}, a) \leftarrow (1-\epsilon)\pi_\theta(s_{root}, a) + \epsilon \mathrm{Dir}(\alpha, a). \tag{5.5}$$

$\mathrm{Dir}(\alpha)$ is a random vector with L_1-norm of 1. Using a low α value will add a high amount of noise to a few moves, whereas a higher value will add a more uniform amount of noise to a larger number of moves. The value of α is dependent on the average number of legal moves per game.

5.2.3 Training Procedure

The policy is trained in generations, with the data from previous generations used to train the next generation's policy network; the training of each generation is synchronous. The first policy is trained using data generated by MCTS using the uniform random policy. Subsequent generations combine MCTS and the current iteration of the policy to generate data. See Figure 5.1 for a visual illustration of this process. To jump-start the learning process, the first generation uses more simulations per step; since no policy network is used, there

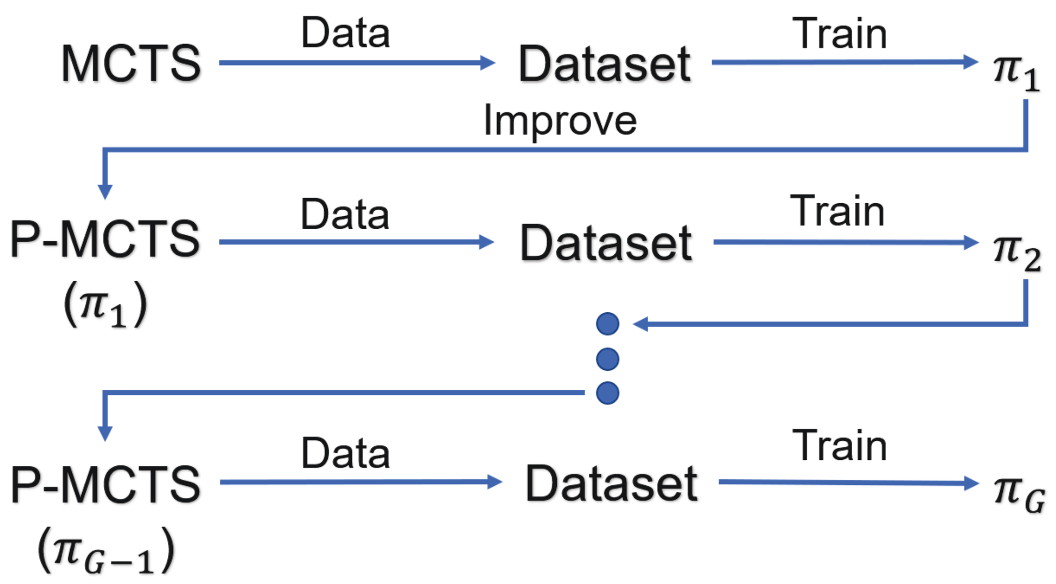


Figure 5.1: An illustration of the training procedure.

is not a large run-time cost to this. I explore the effect of jump-starting the learning process in Section 5.4.3.

The training procedure (Algorithm 1) works as follows: I use two queues, $B_{training}$ and $B_{validation}$, for training, which store state-action pairs (s_t, a_t). As each run finishes (line 9), the resulting pairs are stored in a temporary buffer, which only contains data produced from the current generation (line 10). Once all runs in a generation are finished, the pairs stored in the temporary buffer are shuffled and split (e.g., 90-to-10), and appended to both buffers, respectively (lines 12-15). This ensures that both buffers will contain data from multiple generations. Note that using a single buffer and randomly splitting it before training is not equivalent to this procedure.

The sizes of the buffers determine the amount of data that is kept from older generations; given more data, one can expect the policy to be better [ATB17], but the training time will be longer. The size of the buffers is also directly related to the amount of data generated per run. An alternative approach that will make these two parameters independent is to limit the buffer sizes by the number of games, rather than data points.

The validation buffer is used to avoid overfitting. I use early stopping for regularization. When using early stopping, the network is trained to the point

Algorithm 1: Policy training procedure

```
1  **Input**: #generations $G$, #runs $N$, training/validation buffer lengths
     $l_t, l_v$, training-validation split percentage $\lambda$
2  **Output**: Trained policy network
3  $\pi_0 \sim$ Uniform-Over-Valid-Actions()
4  $B_{training}$ = queue($len$=$l_t$), $B_{validation}$ = queue($len$=$l_v$)
5  for $g = 1, ..., G$ do
6      $B = []$
7      for $r = 1, ..., N$ do
8          $s \leftarrow$ Generate-Random-State()
9          $[(s_1, a_1), ..., (s_T, a_T)] \leftarrow$ Policy-MCTS($s, \pi_{g-1}$)
10         $B \leftarrow B + [(s_1, a_1), ..., (s_T, a_T)]$
11     end
12     Shuffle($B$)
13     $T, V \leftarrow$ Split($B$, $\lambda$)
14     $B_{training} \leftarrow B_{training} + T$
15     $B_{validation} \leftarrow B_{validation} + V$
16     $\pi_g \leftarrow \pi_{\theta=\text{random}}$
17     Train($\pi_g, B_{training}, B_{validation}$)
18 end
19 **Return**: $\pi_G$
```

where another epoch will cause overfitting to the training set, i.e. the validation loss will increase. Therefore, a new network with re-initialized weights is used after every generation. If a new network is not used, only a fraction of the data is new to the network since previous data is also stored in the buffer, which will cause overfitting. The severity of the overfit is proportional to the ratio of new to old data: a larger buffer and a smaller number of runs will cause a higher amount of overfitting. This effect can also be seen while observing the training process, with the validation error increasing after the first epoch of training. Using early stopping and training a new neural network at each iteration is also done in ExIt [ATB17].

An alternative approach is to use weight regularization, such as L_2-norm. When using this regularization technique, the weights of a single network can be continuously trained, as done in AlphaGo Zero [Sil+17] and AlphaZero. My experiments on whether this approach is better than using early stopping were inconclusive.

5.3 Comparison to AlphaZero

My training procedure (Algorithm 1) combines elements from both AlphaZero and ExIt, but is more similar to the former. Therefore, I only provide some of the differences between my algorithm and AlphaZero, which are as follows:

1. The training procedure of AlphaZero is asynchronous, with data generation and training occurring at the same time. While it is not a strict requirement, my algorithm is synchronous.
2. AlphaZero uses a two-headed network which outputs both a policy and value estimate; my training procedure uses only a policy network, and performs full rollouts without any value estimation. Initial work on using value estimation is provided in 6.
3. AlphaZero uses L_2 regularization. Additionally, the network parameters are continuously updated using all of the generated data. In contrast, my algorithm uses early stopping for regularization, which requires a validation buffer. Furthermore, network weights are re-initialized before training on new data.
4. AlphaZero uses a dynamic C value, which is calculated based on visit counts and two constants, whereas my MCTS algorithm uses a static C value.

5.4 Experiments

5.4.1 Network Architecture

The architecture of the network is similar to the one used by [ATB17]. In particular, the input to the network is an "image" of size $d \times d \times (c + 1)$, where d is the dimension of the board and c is the number of block colours, and 1 is added to encode empty tiles. That is, the board is represented as $c + 1$ binary layers to one-hot-encode the tile state. The input is padded by 1 on all four sides, increasing the dimension to $(d + 2) \times (d + 2) \times (c + 1)$.

This ensures that the information at the edges of the board is not lost during convolutions. The padded input is passed to 13 convolution layers, all of which have 64 filters, stride of 1, and ELU activation [CUH15]. Layers 11 and 13 have a kernel size of 1×1, and all others have size 3×3. The dimension of the input is kept (by padding) for layers 1-8 and 12, and reduced by the convolutions in layers 9-11 and 13. The output of the final convolution layer is flattened and passed to a linear layer with $d \times d$ units and softmax activation, which represents the policy. The network has a total of 459,569, 790,820, and 2,814,945 trainable parameters, for 7×7, 10×10, and 15×15 boards, respectively. A simple illustration of the network is given in Figure 5.2. The performance of networks with different sizes is explored in the next section.

Other architectures were also considered, including residual networks, as well as fully connected networks. The performance of residual networks of similar size was comparable to the above network. In contrast, fully connected networks performed poorly.

Adam [KB14] is used as the optimizer, with a learning rate of 5e−4. For

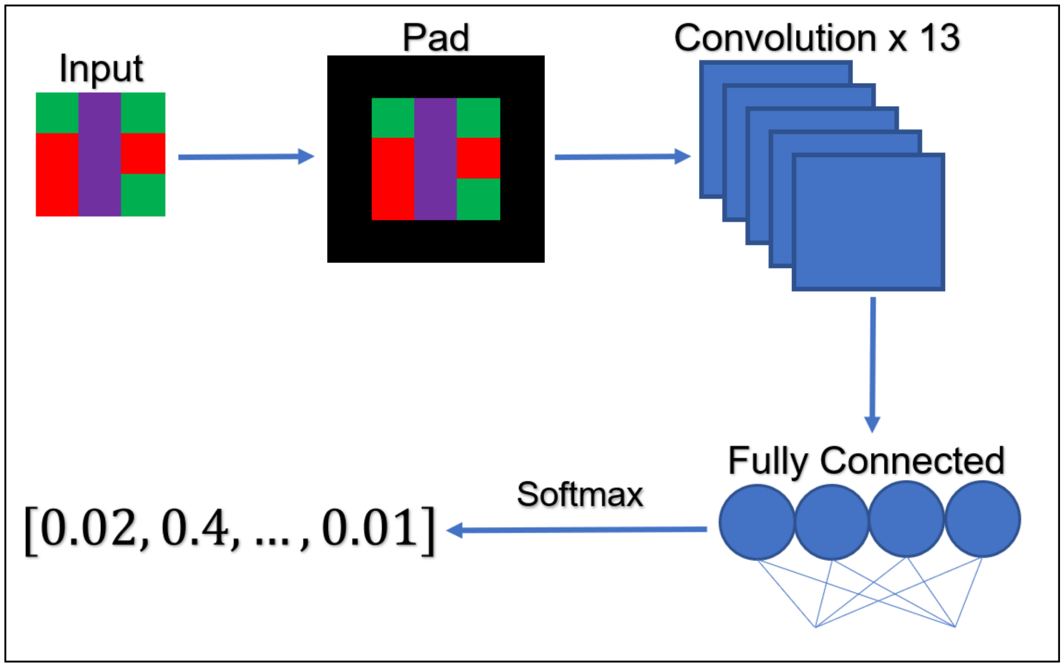


Figure 5.2: A simple illustration of the policy network. For simplicity, a 3×3 SameGame board is used as input.

regularization, early stopping on the validation loss is used. The early stopping point is the first epoch after which the validation loss does not decrease for 2 consecutive epochs. I also experimented with different optimizers, as well as learning rates and early stopping points, and found that the above choices provided a good balance between training speed and stable results across a wide set of experiments.

5.4.2 SameGame Specifics

In SameGame, the player can click on any of the blocks in a group to clear the group. To simplify the search, I only consider the most left, bottom block in a group, respectively. However, forcing the network to learn from this representation, that is, probability of 1 to that block and 0 everywhere else, makes the learning task more difficult. Instead, uniform probability is given to all of the actions that clear the group. Then, the probability given to the unique search action that clears a group is the sum of the probabilities of all actions that clear that group.

A unique property of SameGame is that the colours in a given state can be permutated without affecting the outcome. This knowledge can be exploited by creating a permutation of each data point, which can lead to less correlated data, and thus, potentially increase the generalizability of the network. Since my goal is to showcase the strength of my algorithm, which is general and applicable to a wide range of domains, I explicitly don't exploit this domain knowledge.

5.4.3 Parameter Study

Policy-MCTS has several parameters that can be optimized. The standard approach for finding reasonable parameter values for an algorithm is through grid-search, which was the method I used to optimize the number of threads based on different values for the virtual loss weight parameter w in Section 4.4. However, a grid search is not feasible in this scenario, as each experiment can take anywhere from several hours to multiple days. It is also not enough to

Table 5.1: Default parameters used for studying the effects of different parameters on training.

Parameter	Value
Generations	20
Runs/generation	1,000
Simulations	100
C	2
Threads/run	1
Runs in parallel	50
Dirichlet noise α	0.25
Training buffer size l_t	120,000
Training-validation split λ	0.10
Batch size	32
Rollout type	Random
Jump-Start	None (Section 5.2.3)
Neural Network	See Section 5.4.1
CPU	Xeon 6148 10 cores @ 2.4 GHz
GPU	Nvidia V100/16 GB memory

optimize each parameter in isolation, and then combine the optimized values together, as many of the parameters are strongly correlated with others. Therefore, the goal of this section is not to thoroughly tune every parameter, but to provide some intuition about the effects of certain parameters on the training procedure.

The default parameters used for each experiment are provided in Table 5.1; only 20 generations are performed per experiment to keep a low overall runtime. All experiments are done on 15×15 SameGame, with all boards generated randomly. Whenever possible, the results from the same experiments are used in multiple places, i.e. the same experiment with the exact same parameter values is not repeated.

In some sections, reference is made to a *test set*, which is a collection of 500 randomly generated 15×15 SameGame boards. All experiments that make use of this test set execute 5 runs per board, for a total of 2,500 runs, with 5 seconds of computation time per run; 5 seconds was determined to be long enough to make fair comparisons between the different parameter choices.

Due to the computational expense of these experiments, each experiment is

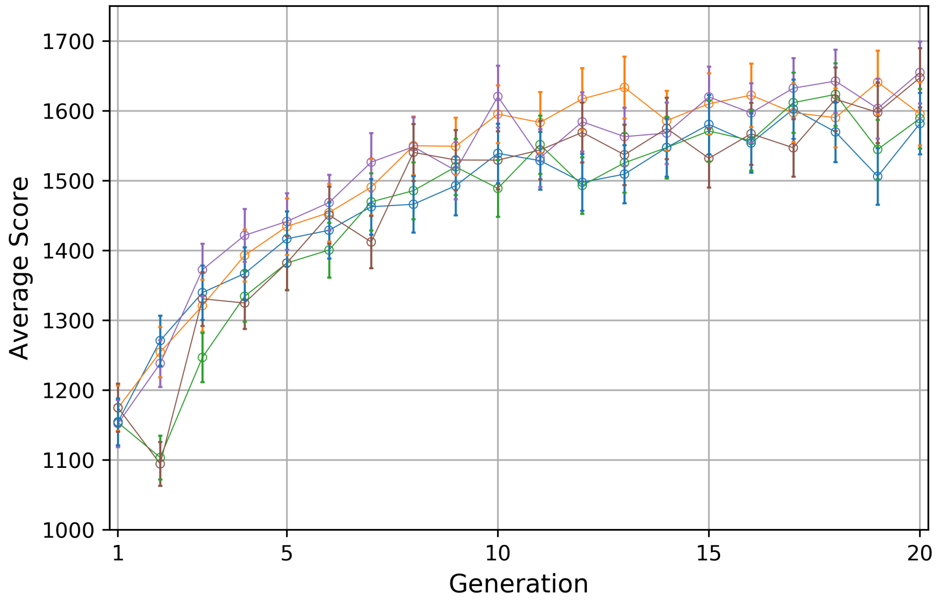


Figure 5.3: Average score of 5 different training runs, all of which use the default parameters provided in Table 5.1.

only performed once. This is not a major issue however, as most runs behave very similarly (Figure 5.3). Note that in all learning curve graphs, the y-axis depicts the average score obtained by the algorithm on the randomly generated boards used during training, i.e. the average of scores obtained in lines 7-11 of Algorithm 1, and not the variance over repeated runs; the error bars indicate the 99% confidence interval of the score.

Simulation Count

The number of simulations per move is correlated with the strength of the MCTS algorithm. By increasing the number of simulations, MCTS is expected to perform better, and thus, act as a stronger policy improvement operator. The simulation count needs to be high enough such that MCTS is able to continuously improve upon the current policy, because otherwise, the policy can get stuck in local optima. However, as the simulation count is increased, training time is also increased. The effect of the simulation count on the training performance is shown in Figure 5.4.

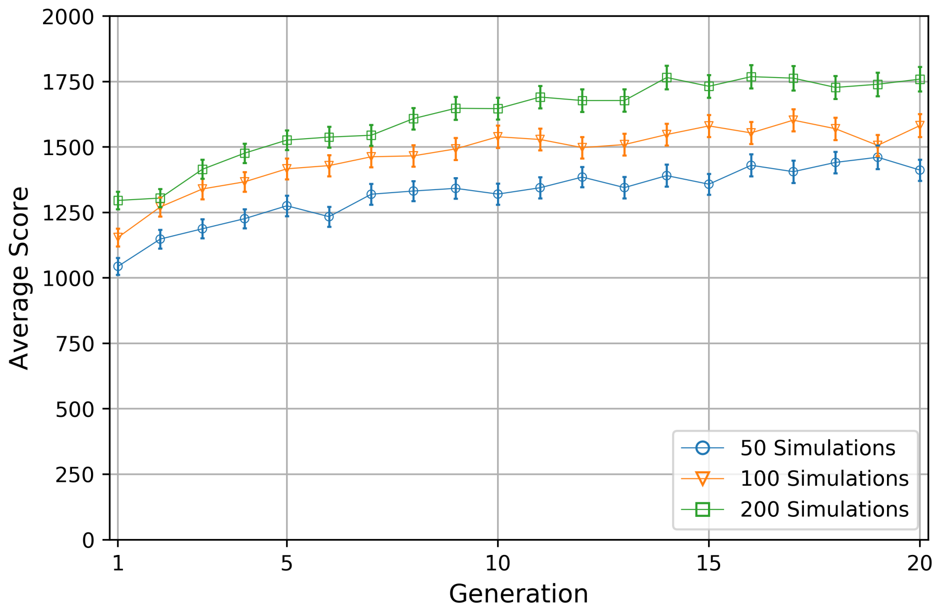


Figure 5.4: Average score of policies during training, for 20 generations, with varying number of simulations per move.

There is a clear difference in performance during training when using only 50 simulations compared with using 200 simulations per move. However, this comes at the cost of increased training time, as shown in Table 5.2. However, the difference in performance can be entirely due to the increased simulation count, and not a stronger policy: to make a fair comparison, the final policy from each experiment is used to run Policy-MCTS on the test set. The results shown in Figure 5.5 provide some evidence that using a higher number of simulations will produce a stronger policy.

Table 5.2: Training time per generation of Policy-MCTS with 1,000 runs per generation, and using 50, 100, and 200 simulations per move. Each run corresponds to solving a single 15×15 board.

Simulations	Minutes Per Generation
50	~7.1
100	~10.3
200	~16.9

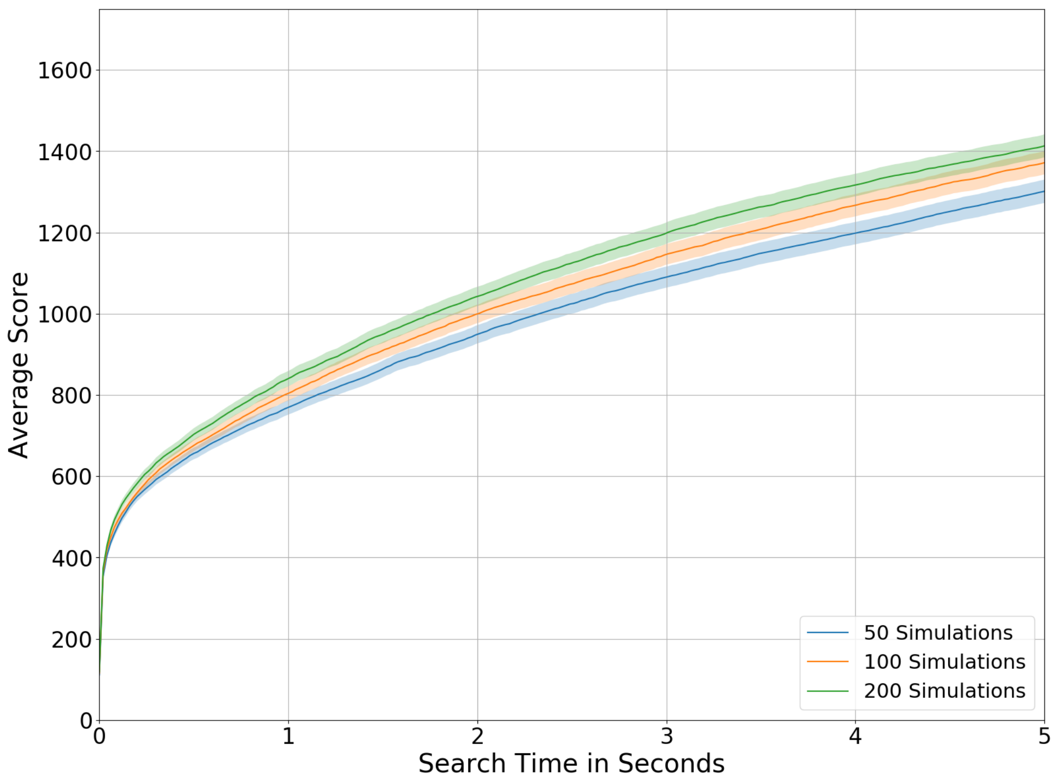


Figure 5.5: Comparing the performance of Policy-MCTS on the test set using policies trained for 20 generations, with varying number of simulations per move. The y-axis depicts the average score attained on 500 boards, with 5 runs on each, for a total of 2,500 runs. Each run is given up to 5 seconds of computation time per run, indicated by the x-axis. The shaded areas indicate the 99% confidence bands.

Comparing the performances in Figures 5.4 and 5.5, the average scores obtained during training is higher. There are two possible explanations for this. The first is that during training, the simulation budget is spent uniformly across moves, which can lead to better performance (see Section 5.2.2). Secondly, as there is no time limit while training, the overall number of simulations can be higher than the amount performed in five seconds, which can lead to better performance.

A conclusion can be made that if one is willing to spend more time on training, increasing the simulation count will lead to better performance. Note that there is a point of diminishing returns, for example, having to double the simulation count for a slight boost in performance; this does not seem to be the case in the above experiments. Lastly, since the number of generations

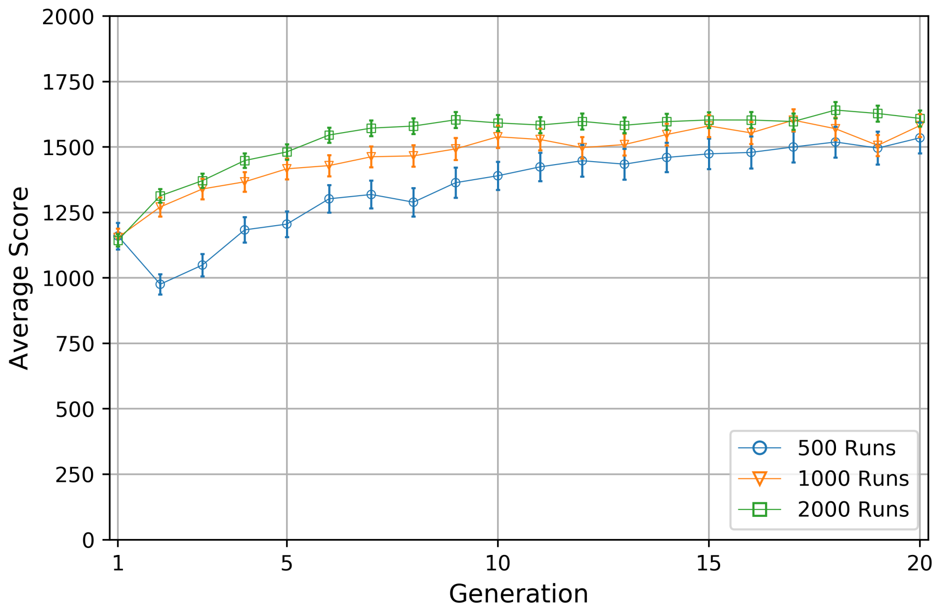


Figure 5.6: Average score of policies during training, for 20 generations, with varying number of runs per generation.

was limited to 20, the effect of the number of simulations on the asymptotic performance of the algorithm cannot be determined from these experiments.

Runs Per Generation

Neural networks often require large amounts of data to generalize to states not present in the dataset. In my algorithm, the number of data points for training is limited by the buffer size l_t, but the amount of new, higher quality data added per generation—since the most current network is used—is determined by the number of runs. However, as the number of runs is increased, so, too, does the training time. Therefore, a balance needs to be found.

The effect of using different number of runs per generation is shown in Figure 5.6. Several interesting observations can be made from these results. The first is that the performance in the second generation decreases when using only 500 runs, whereas it increases for 1,000 and 2,000 runs. This could simply be an artifact from performing only a single experiment, or another explanation is that, due to lack of data, the network is unable to generalize as

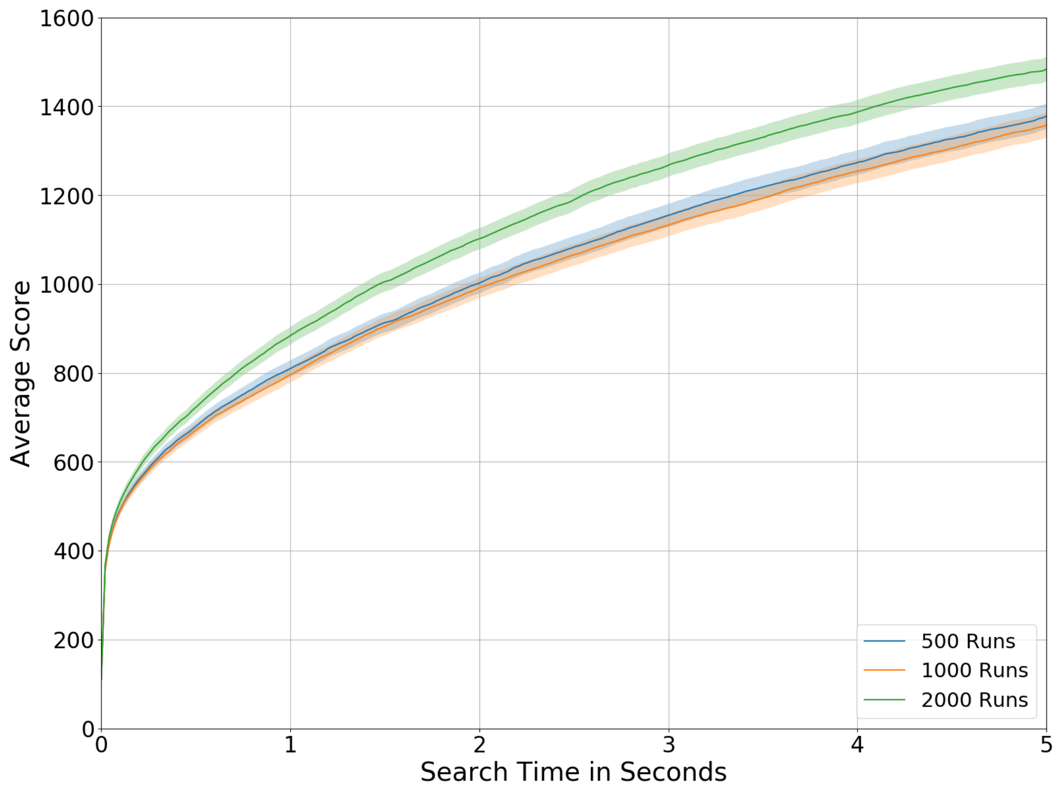


Figure 5.7: Comparing the performance of Policy-MCTS on the test set using policies trained for 20 generations, with varying number of runs per generation. The y-axis depicts the average score attained on 500 boards, with 5 runs on each, for a total of 2,500 runs. Each run is given up to 5 seconds of computation time per run, indicated by the x-axis. The shaded areas indicate the 99% confidence bands.

well. Furthermore, while the trajectories all look different, all policies perform similarly in the last generation. This potentially implies that the number of training points, which is controlled by the training buffer size, is more important to the performance of the algorithm than replacing older data with higher quality data generated by the most recent policy. But, overall, executing more runs seems to provide the best and most stable performance. This gain comes at the cost of training speed however, with training times similar to ones provided in Table 5.2.

To make a better comparison between the strengths of the policies, I run Policy-MCTS, using the final trained policy of each experiment, on the test set. The result is presented in Figure 5.7. The policy trained with 2,000 runs

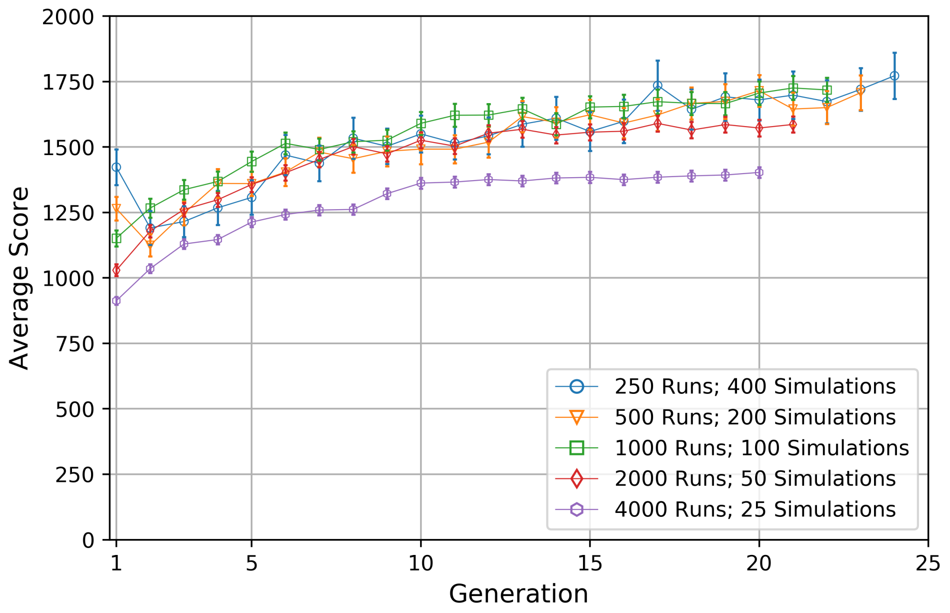


Figure 5.8: Average score of policies during training, given a total time limit of 5 hours, with varying number of runs per generation and simulations per move.

per generation clearly outperforms both other policies. It is interesting that the policy trained with 1,000 runs is marginally outperformed by the policy trained with only 500 runs; note that their confidence intervals have a large overlap. This can simply be due to the states seen during training, or the fact that the difference in the number of runs is not significant enough to make an impact on the final performance of the policy.

More Runs vs. More Simulations

A natural question that arises when observing the effects of simulations per move and runs per generation is: given a limited time budget, is it better to increase the number of runs or the simulation count? The answer is not immediately clear, as increasing the simulation count will make the policy improvement step better, whereas increasing the number of runs creates more data per generation, which improves the policy evaluation step.

To investigate the above question, I run five different experiments, each with a time limit of 5 hours, using a different number of runs per generation

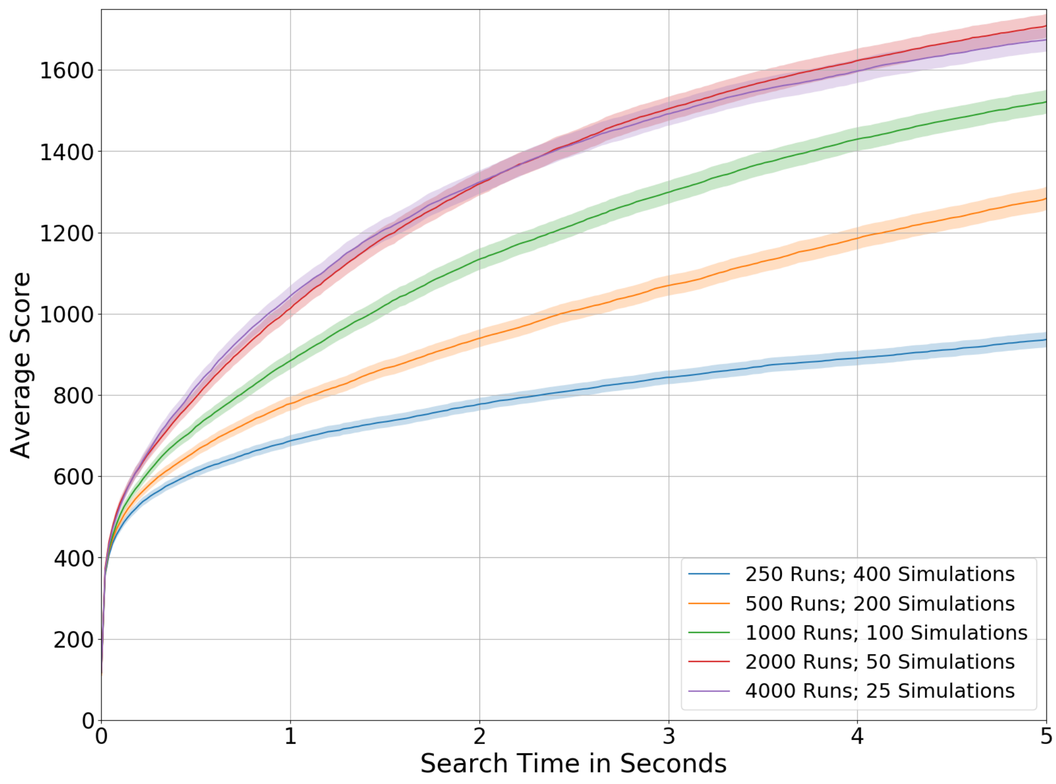


Figure 5.9: Comparing the performance of Policy-MCTS on the test set using policies trained for 5 hours, with varying number of runs per generation and simulations per move. The y-axis depicts the average score attained on 500 boards, with 5 runs on each, for a total of 2,500 runs. Each run is given up to 5 seconds of computation time per run, indicated by the x-axis. The shaded areas indicate the 99% confidence bands.

and simulations per step. The training buffer size is increased to 250,000 data points, which is roughly the amount of data created by 4000 runs, as to not throw away any generated data before it can be used for training. The performance during training for each experiment is provided in Figure 5.8. While the training performance between the different policies can be compared, it is difficult to determine the contribution of the apprentice policy network versus that of the MCTS expert. Therefore, I take the final policy from each experiment, and run Policy-MCTS on the test set. The results are given in Figure 5.9.

Contrary to the training graphs, the policies trained with more runs and lower sims perform better on the test set. One possible explanation for this is that training on more states allows the network to generalize better, leading to better predictions on the test set, even if the quality of the data generated during training is possibly worse. Additionally, since 4000 runs nearly fills up the entire training buffer, all of the data from previous generations is replaced by data generated from the strongest policy, which can compensate for the smaller number of simulations. Lastly, this implies that it is not the apprentice policy network, but the expert MCTS with a higher number of simulations, that is responsible for the better performance during training. As a final note, the policy trained with 2000 runs and 50 simulations outperforms, on the test set, the policy from the previous section which used 2000 runs and 100 simulations ($p < 0.0001$). One reason for the difference in performance is the bigger training and validation buffer sizes. Overall, I can conclude that given a limited training budget, executing a higher number of runs with a lower number of simulations per move can lead to a stronger policy than executing a lower number of runs with a higher simulation count.

Policy Training Target

To confirm that the new policy target outperforms the two-player variant (Eq. 5.2), I train two separate policies using the same parameters except for the training target. The results presented in Figure 5.10 clearly show that the new target performs better, given that the two-player variant's performance slightly declines during training, which implies that the policy is unable to guide the search. One possible explanation for the poor performance is the low ratio of simulation count to branching factor, which is around 5:1. Even with this low ratio, the single-player variant is still able to improve, making it the superior choice.

Policy-Guided Rollouts

It has been shown that using an informed policy to guide the rollouts of MCTS can greatly increase performance [Bro+12]. Therefore, it might also be beneficial to use the policy network to guide the rollouts during training. However, since the network needs to make a prediction at every step of the rollout, the training time is increased by a large amount; the added cost is correlated with

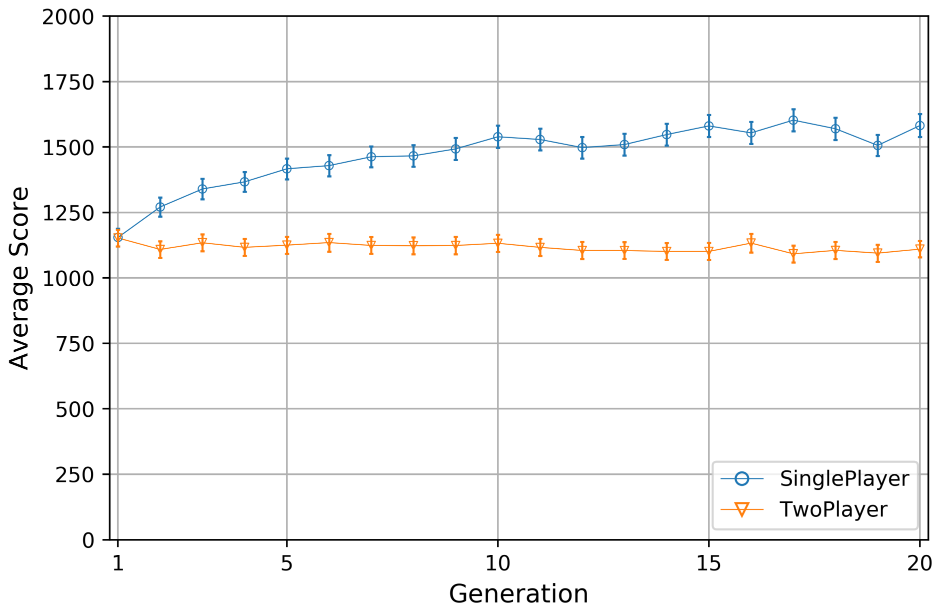


Figure 5.10: Average score of policies during training, for 20 generations, with a single-player or two-player training target.

the average number of moves in a rollout.

To compensate for the increase in training time, the simulation count can be lowered. Figure 5.11 shows the training performance when using 5 guided or 200 random simulations, with both taking roughly 18 minutes per complete generation (data generation and training). Since the difference in performance can be due to different simulation counts, the final policy from each experiment is used to run Policy-MCTS with random rollouts on the test set (Figure 5.12).

The results might seem counter-intuitive, as the network trained with guided rollouts performs best, whereas during training, the one trained with random rollouts achieves a better average score by 375 points. This implies that it is easier for the policy network to learn and generalize the data created while using guided rollouts. One explanation is that the state distribution in the dataset created using guided rollouts more closely resembles the state distribution actually experienced by the agent.

These results demonstrate that, when the simulation strategies are different, analyzing only the training performance can lead to false conclusions

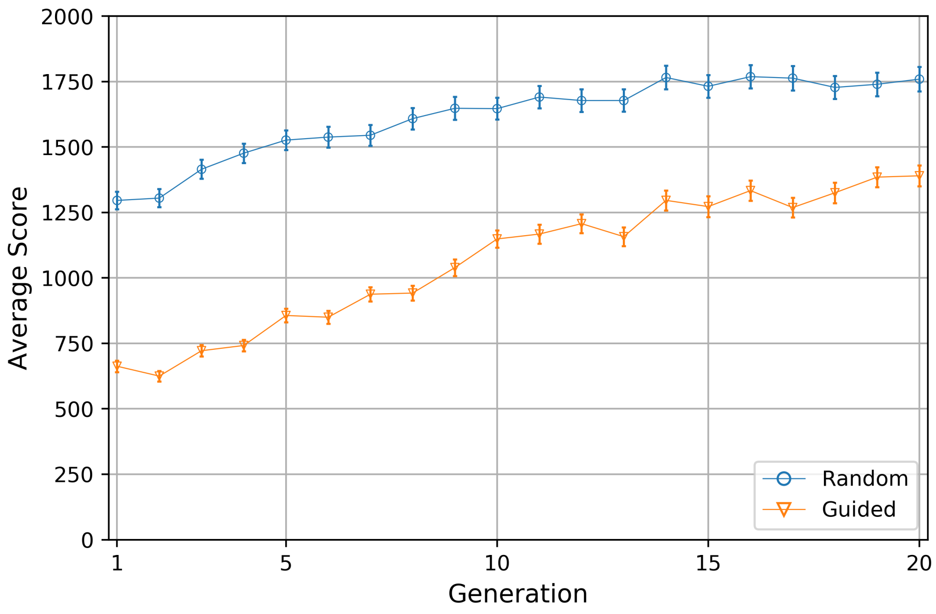


Figure 5.11: Average score of policies during training, for 20 generations, with 5 guided or 200 random rollouts.

on the strength of the policy, i.e. better performance during training does not necessarily translate to a stronger policy. Note that, since the number of generations was limited to 20, the effect of the simulation strategy on the asymptotic performance of the algorithm cannot be determined from these experiments. However, given a limited training budget of 20 generations, using informed rollouts seems to be the better choice.

Jump-Starting

The first generation during training uses plain MCTS, which is far faster than policy MCTS, but also less informed. Therefore, using the same simulation count per move in the first generation might not be the ideal choice. Instead, a much higher number of simulations can be used, with little effect on the total training time. I call this “jump-starting” the training process. By making the first generation stronger, one can expect the training performance to be better in earlier generations, which might also have a positive impact on the later generations as well.

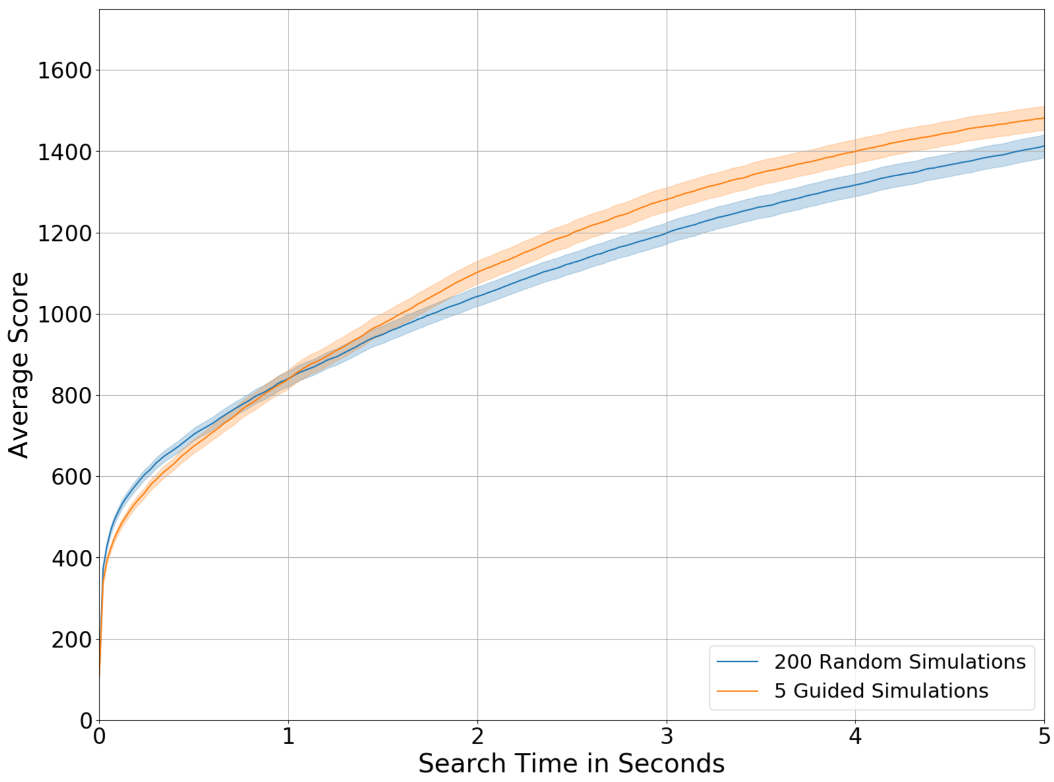


Figure 5.12: Comparing the performance of Policy-MCTS on the test set using policies trained for 20 generations, with random, or guided rollouts, respectively. The y-axis depicts the average score attained on 500 boards, with 5 runs on each, for a total of 2,500 runs. Each run is given up to 5 seconds of computation time per run, indicated by the x-axis. The shaded areas indicate the 99% confidence bands.

To test the above hypothesis, I run four different experiments, with the only difference being the number of simulations used in the first generation. The results presented in Figure 5.13 show that jump-starting has a relatively large impact on the first few generations: with no jump-starting, the second generation averages a score of 1,271, whereas when jump-started using 10,000 simulations, an average score of 1,386 is achieved. Jump-starting seems to have little effect on the later generations, with all policies achieving similar scores. This implies that the positive effect on the first several generations is not enough to boost the overall performance.

The training times for the above policies are presented in Table 5.3. For comparison, a single generation with Policy-MCTS, using 1,000 runs and 100 simulations, takes around 7 minutes. While using a large number of simu-

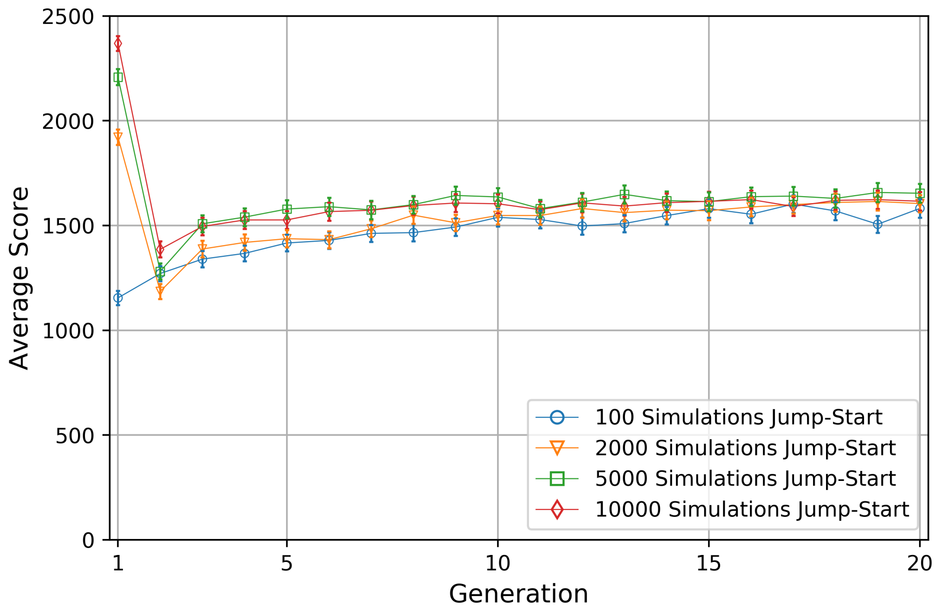


Figure 5.13: Average score of policies during training, for 20 generations, when jump-starting is used. Each policy is trained using a different number of simulations per move in the initial generation.

lations to jump-start the training process seems to provide the best initial performance, it comes with a relatively high training time cost; the increase will be less pronounced when a higher number of simulations are used in Policy-MCTS.

It can be concluded that, at the cost of increased training time, jump-starting the process does have a positive impact on the initial generations. It

Table 5.3: Comparison of training times when using 100, 2,000, 5,000, and 10,000 simulations per move for plain MCTS for jump-starting the training process. There are 1,000 runs per generation, and 100 simulations per move for Policy-MCTS.

Simulations	First Generation Time in Minutes	Total Training Time in Minutes
100	~1	~202
2,000	~14	~214
5,000	~30	~232
10,000	~64	~260

is unclear whether jump-starting has a large impact on the later generations.

Network Size

The size (and depth) of the network determines the number of trainable parameters. Naturally, a bigger network has more trainable parameters, and therefore, is able to learn more. However, having too many parameters can lead to overfitting, which limits the ability of the policy to generalize to unseen states. Since all states are randomly generated, one can assume that no state will be seen again, making generalization important. Does there exist a trade-off, or is simply using the largest network possible—given my computational budget—the best choice?

To answer the above question, I train four networks of different sizes. The small network uses 2 convolution layers with 64 filters, kernel size of 3×3, stride of 1, and ELU activation, and a fully connected layer with 64 units. The medium network uses 4, 128, and the large network uses 6, 256, convolution layers and fully connected units, respectively. The "default" model is as described in Section 5.4.1, which uses 13 convolution layers and no fully connected layer.

Training performance of all four networks is shown in Figure 5.14. As expected, the network size has a large impact on the overall training performance: the small and medium networks are unable to outperform plain MCTS (generation 1), whereas the large and default networks are able to continuously improve; the default network performs the best. This suggests that an even bigger network could be better, but using such a network is expensive, as both training and prediction times will increase.

It is a straight-forward conclusion that the strength of the policy network is very important to the training process, as the network needs to be able to effectively generalize the policy of the expert MCTS. I would hypothesize that using a much larger network will perform even better; AlphaZero uses a network with 19 residual blocks, which is 40 convolution layers, each with 256 filters. Such a network has over a hundred times more trainable parameters than the default network I use. However, using such a large network is out of

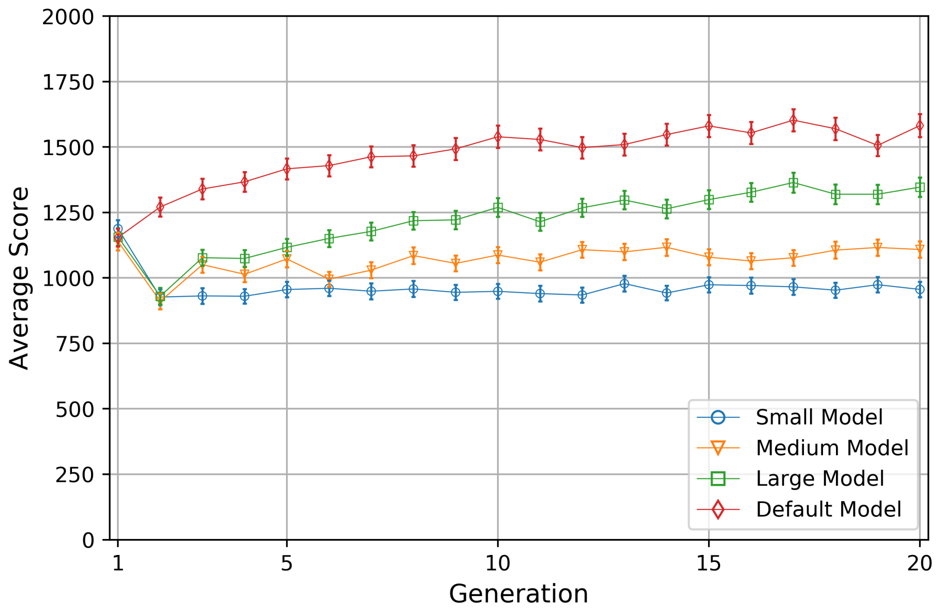


Figure 5.14: Average score of policies during training, for 20 generations, with networks of different sizes.

my computational budget.

Queue Size vs. Number of Threads

I will experimentally optimize the number of threads and queue size L of Policy-MCTS with random rollouts, using $w = 0.01$ and the same CPU and GPU as provided in Table 5.1, on the test set. The results are provided in Figure 5.15.

The results demonstrate that multi-threaded Policy-MCTS can outperform the single-threaded version by a large margin, assuming that a proper balance between the number of threads and queue size is found. Using 60 threads with a higher queue size of 32, 40, or 48 performs the best. Even though the CPU has only 10 cores, spawning several factors more threads is beneficial as each thread is put to sleep while waiting for the prediction result. When the queue size is small however, using too many threads will lead to worse performance, as most of the computation time will be wasted on waiting for predictions, as well as threads competing with each other for CPU resources. Given the

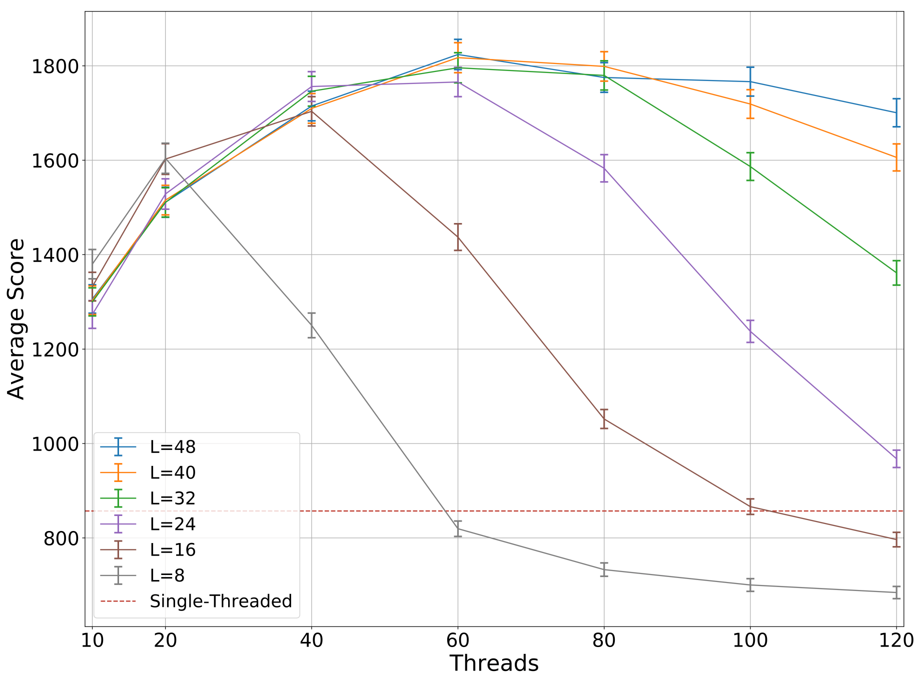


Figure 5.15: The performance of Policy-MCTS, given 1 second of computation time, with different number of threads and queue sizes.

results, I can conclude that using 60 threads with a queue size of 48 is a reasonable choice for parallel Policy-MCTS with random rollouts, assuming the above computation devices are used.

5.4.4 Comparing with Plain MCTS

In this section, I compare the performance of Policy-MCTS against plain MCTS on SameGame boards of size 7×7, 10×10, and 15×15. To demonstrate the effectiveness of the parallelization strategy introduced in Section 4.2, I also compare the performance of single and multi-threaded variants of both algorithms.

Two policy networks are trained per board size, using either random or guided rollouts. The 7×7 board is small and relatively simple. The other two sizes are increasingly more complex, with 15×15 SameGame often used as a benchmark problem in the literature. The average solution length of

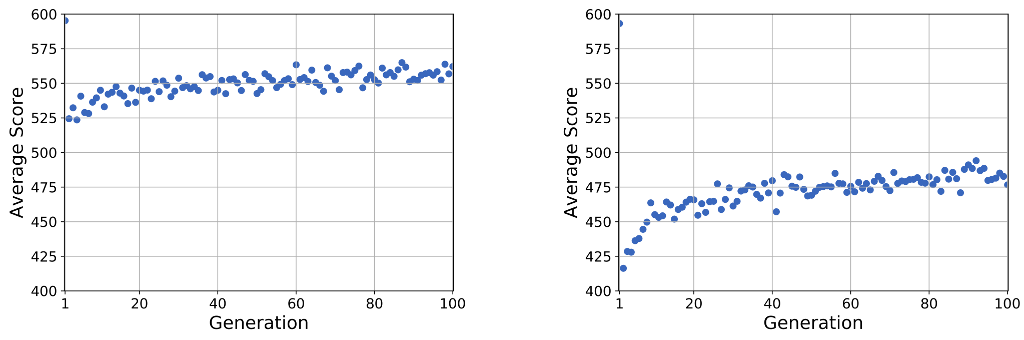
Table 5.4: Parameters used for training policies for 7×7, 10×10, and 15×15 SameGame. Two networks are trained per board size, using either random or guided rollouts.

Parameter	7×7	10×10	15×15
Generations	100	100	100
Runs/generation	10,000	10,000	5,000
Random simulations	1,500	600	450
Guided simulations	100	25	15
C	30	4	2
Threads/run	1	1	1
Dirichlet noise	0.75	0.40	0.25
Jump-start	10,000	10,000	10,000
Training buffer size	1,500,000	1,500,000	1,500,000
Training-validation split	0.10	0.10	0.10
Batch size	256	256	256
Training time (days)	~6	~7	~9

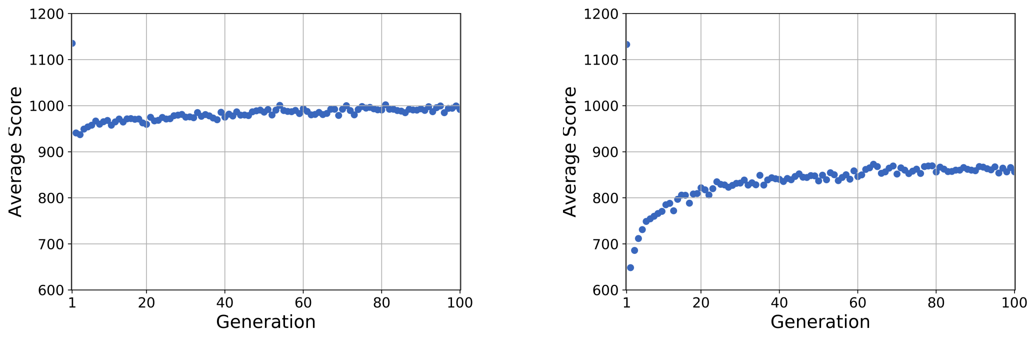
10×10 SameGame is roughly twice that of 7×7, and half that of 15×15. To keep training time manageable, the number of runs and/or simulations are decreased as the board size increases. The same computation devices are used to train all the policies: 2 x Intel Xeon 6148 20 Cores @ 2.4 GHz CPU and 4 x Nvidia V100 with 16GB memory GPU.

To train stronger policies, the values of several parameters are increased from the default values used in the previous section; parameter values are provided in Table 5.4. All networks are trained for the same number of generations, with several factors more runs per generation than simulations per move, given the results of Figure 5.9. The time required to perform the listed number of random and guided simulations is roughly equal. The training buffer contains data for approximately 11, 6, and 5 generations, for 7×7, 10×10, and 15×15 boards, respectively. Note that while some effort was put into optimizing the parameters, they can be further improved given more time and computational resources. Performance during training for all six policies is presented in Figure 5.16.

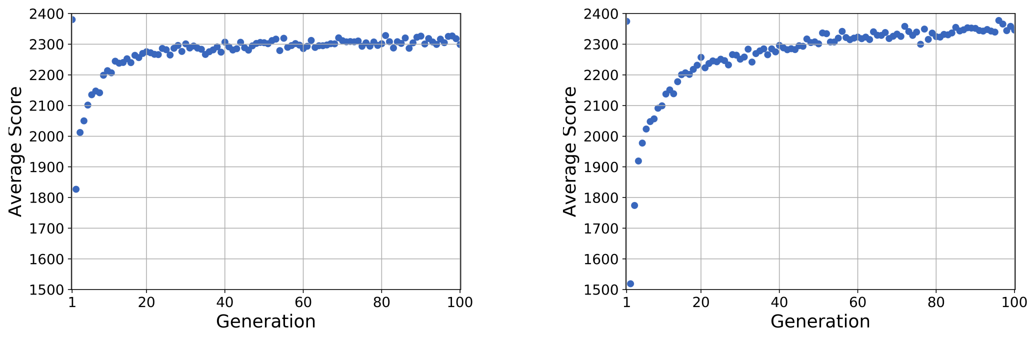
Given the same node budget, Policy-MCTS is expected to outperform plain MCTS. However, using a neural network incurs a large cost on the speed of



(a) 7×7



(b) 10×10



(c) 15×15

Figure 5.16: Average score of policies during training, for 100 generations. The plots on the left are policies that were trained using random rollouts, and the ones on the right used informed rollouts.

the search. Therefore, to make a more fair comparison, wall-clock time is used for all of the experiments. However, using wall-clock time is not ideal either, as now both the computation device and the implementation matter. The former is mitigated by using the same computer to run all the experiments.

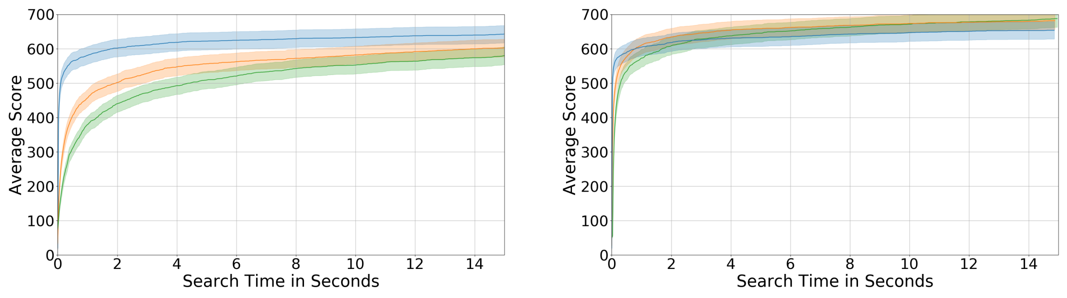
For the latter, I described my implementation of the prediction queue as well as parallel MCTS to make these experiments reproducible.

Since wall-clock time is used, I also present the averages of simulation count, terminal node expansions, and node expansions for the experiments on the 15×15 boards. The simulation count also contains simulations starting (and ending) at terminal leaf nodes, that is, terminal states that are part of the search tree. The number of terminal node expansions is not included in the node expansions count.

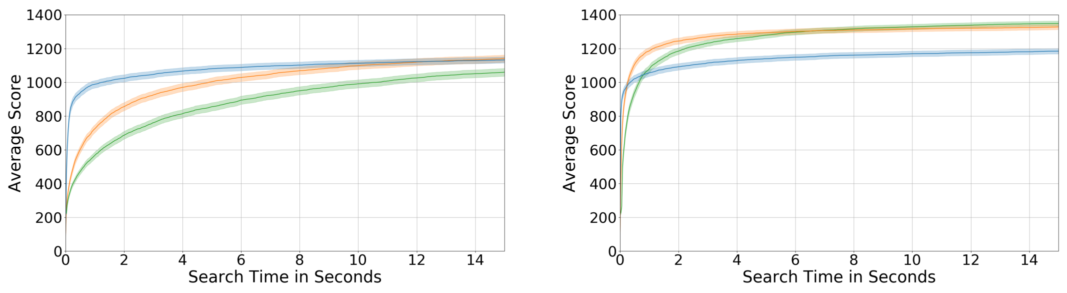
The same parameters are used for both the single-threaded and multi-threaded experiments. Additionally, the C value used during training is kept for all experiments. The number of threads used by parallel plain MCTS is 10, which is the number of available cores, with virtual loss weight $w = 0.01$. Parallel Policy-MCTS uses 60 threads with a queue size of 48 and $w = 0.01$, which were the best performing parameters (Figure 5.15). Note that these values were optimized using random rollouts, and not informed rollouts; I did not optimize the number of threads and queue size for Policy-MCTS with informed rollouts, as it was computationally expensive to optimize the parameters with random rollouts.

A test set of 500 randomly generated boards of each size is used to compare Policy-MCTS with plain MCTS. For each experiment, 5 runs are performed per board, for a total of 2,500 runs. Results are shown for plain MCTS, and Policy-MCTS with random, and policy guided rollouts, respectively. The same computer is used for all experiments: 10 computation cores (Intel Xeon 6148 @ 2.4 GHz) and a single GPU (Nvidia V100 with 16GB memory).

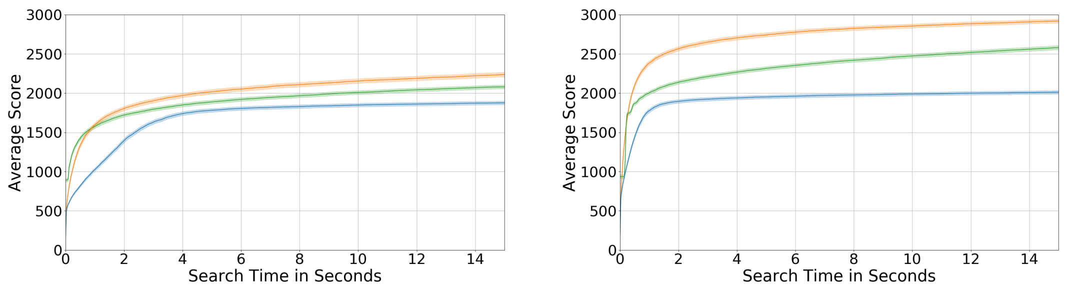
The experimental results shown in Figure 5.17 and 5.18 demonstrate that the multi-threaded versions of all three algorithms outperform their single-threaded counter-parts. The performance of Policy-MCTS is greatly improved when the algorithm is parallelized: for the 15×15 board, the average score of Policy-MCTS with random rollouts, using a policy trained with random rollouts, rises from 2,238 to 2,919, an increase of nearly 30%. This is expected, as using many threads allows for batch prediction, which utilizes the GPU more effectively.



(a) 7 × 7



(b) 10 × 10

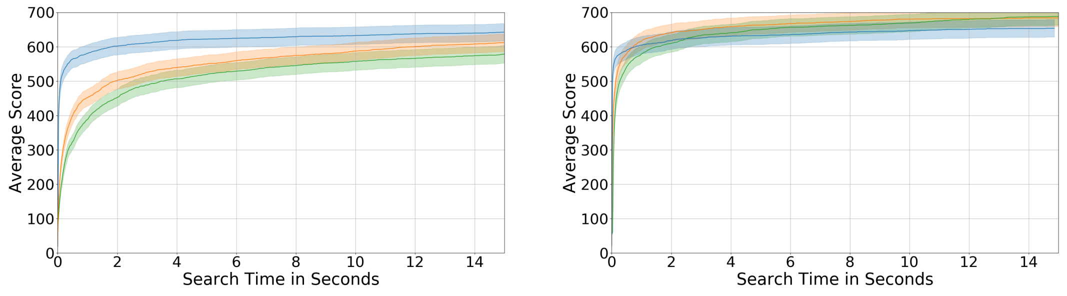


(c) 15 × 15

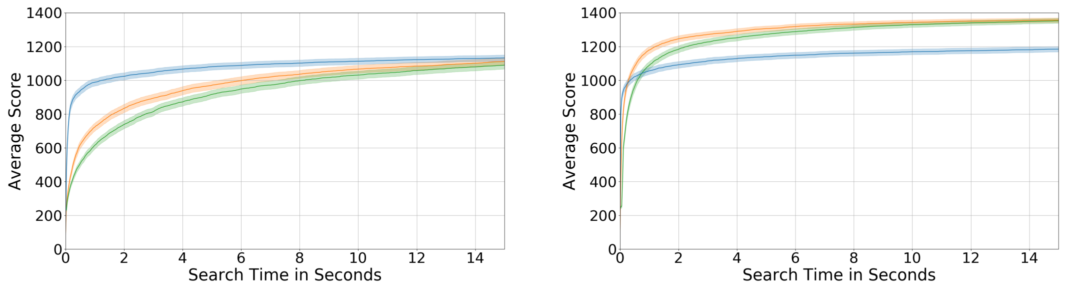
— MCTS — Policy-MCTS Random — Policy-MCTS Informed

Figure 5.17: Average score of single-threaded (left) and multi-threaded (right) MCTS, Policy-MCTS with random rollouts, and Policy-MCTS with informed rollouts, as a function of search time in seconds. All policies are trained using random rollouts. Each experiment is performed on a test set of 500 randomly generated boards, with 5 runs per board. The shaded areas indicate the 99% confidence bands.

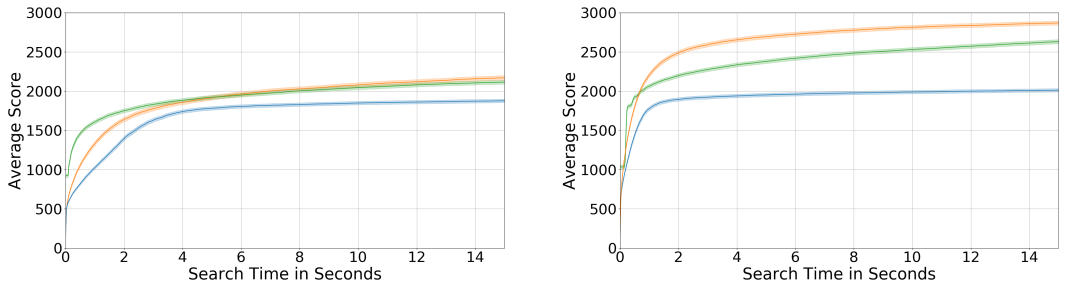
For plain MCTS, parallelization seems to have a large initial impact, but the final solution sees little improvement. This implies that the search becomes stuck in local optima. This is a common problem with UCB-based selection strategies in single-player domains; one solution to this problem is random



(a) 7×7



(b) 10×10



(c) 15×15

Figure 5.18: Average score of single-threaded (left) and multi-threaded (right) MCTS, Policy-MCTS with random rollouts, and Policy-MCTS with informed rollouts, as a function of search time in seconds. All policies are trained using informed rollouts. Each experiment is performed on a test set of 500 randomly generated boards, with 5 runs per board. The shaded areas indicate the 99% confidence bands.

restarts [Sch+12]. Lastly, the results show the downside of using a neural network on simpler problems when given a short time limit, with single-threaded MCTS outperforming single-threaded Policy-MCTS in both 7×7 and 10×10 SameGame.

Table 5.5: Number of simulations, node expansions, and leaf node expansions, after 15 seconds of search, for single-threaded and multi-threaded experiments on 15 $\times$ 15 SameGame. The policy used in Policy-MCTS is trained using random rollouts.

Single-Threaded			
Statistic	MCTS	Policy-MCTS Random	Policy-MCTS Informed
Simulations	184,019	20,725	328
Expansions	54,042	14,077	328
Leaf Expansions	1,380	241	0
Multi-Threaded			
Statistic	MCTS	Policy-MCTS Random	Policy-MCTS Informed
Simulations	840,939	499,132	4,972
Expansions	147,009	211,249	4,972
Leaf Expansions	5,588	5,019	0

Another important observation that can be made is that as the decision complexity increases, the strength difference between plain MCTS and Policy-MCTS also increases. Given a simpler problem and a limited time budget, it seems more beneficial to use plain MCTS. However, as the complexity of the problem increases, so, too, does the benefit of using a policy. This is clearly demonstrated in the 15 $\times$ 15 graphs, with Policy-MCTS far outperforming plain MCTS in both single and multi-threaded experiments.

Detailed performance data for experiments on 15 $\times$ 15 SameGame is provided in Table 5.5. Given the similar performance of the two policies, only data for the policy trained using random rollouts is given. Even with multi-threading, only 4972 policy guided rollouts were finished, compared to nearly 841K for plain MCTS. Note that 328, and 4972 informed rollouts outperform 184K, and 841K random rollouts from plain MCTS, respectively. Additionally, the single-threaded version of Policy-MCTS with random rollouts expands less than a third of the nodes of plain MCTS, but has a higher score by 361 points. This provides evidence that the training procedure has produced a competent policy. The data also suggests that multi-threaded Policy-MCTS with random

Table 5.6: Parameters used for runs on the benchmark set of 20 test positions.

Parameter	Policy-MCTS	Parallel-MCTS
Runs	3	1
C	5	10
Threads/Run	120	40
Time/Run	2 hours	2 hours
CPU	2 x Intel Xeon 6148 20 Cores 2.4 GHz	
GPU	NVidia V100 16GB	None

rollouts creates a wider tree than plain MCTS. The multi-threaded version of MCTS could potentially benefit from further exploration—by increasing C—as only one in five simulations expands a new node.

5.4.5 Comparison to State-of-the-Art Search Algorithms

In this section, I compare the performance of my parallel MCTS and Policy-MCTS algorithms against published state-of-the-art search methods on a benchmark set of 20 test positions; some of these algorithms were discussed in Section 2.2. Note that the performance of each algorithm on the test set is taken directly from the respective publication. The two 15×15 networks trained from the previous section are used in Policy-MCTS: the network trained with random rollouts is used in Policy-MCTS with random rollouts, and the network trained with informed rollouts is used in Policy-MCTS with informed rollouts, as each network performed best when tested with the same type of rollout it was trained with. To gain a better perspective of the results obtained in the last section, I additionally run the parallel version of plain MCTS. The search parameters are provided in Table 5.6, and the results in Table 5.7.

I give Policy-MCTS and plain MCTS two hours total per position, which is similar to SP-MCTS [Sch+12] and NMCS(4). The results of Dist-NRPA(5) [NC17] are also achieved in two hours, but while using 160 CPUs. Algorithms Sel-NRPA(4) and Div-NRPA(4) [EC16] take about half a day per position; the number in the brackets represents the nesting level used by the algorithm. Most of these algorithms, with the exception of Div-NRPA(4) and Dist-NRPA(5), also make use of hand-crafted heuristics to guide the rollouts.

Table 5.7: Performance of various algorithms on the benchmark set of 20 test positions.

Position	SP-MCTS	NMCS(4)	Sel-NRPA(4)	Div-NRPA(4)	Dist-NRPA(5)	Parallel-MCTS	Policy-MCTS Random	Policy-MCTS Informed
1	2,919	3,121	3,179	3,145	**3,185**	1,559	2,777	2,923
2	3,797	3,813	**3,985**	**3,985**	**3,985**	2,251	3,769	3,885
3	3,243	3,085	3,635	**3,937**	3,747	2,361	3,031	3,189
4	3,687	3,697	3,913	3,879	**3,925**	2,241	3,125	3,181
5	4,067	4,055	4,309	4,319	**4,335**	2,337	3,865	3,747
6	4,269	4,459	**4,809**	4,697	**4,809**	3,137	4,491	4,557
7	**2,949**	**2,949**	2,651	2,795	2,923	2,195	2,617	2,725
8	4,043	3,999	3,879	3,967	4,061	3,363	3,973	**4,199**
9	4,769	4,695	4,807	4,813	**4,829**	2,865	4,765	4,741
10	3,245	3,223	2,831	3,219	3,193	2,245	3,083	**3,261**
11	3,259	3,147	3,317	3,395	**3,455**	2,067	2,969	3,133
12	3,245	3,201	3,315	3,559	**3,567**	2,819	3,045	3,435
13	3,211	3,197	3,399	3,159	**3,591**	2,523	2,783	3,307
14	2,937	2,799	3,097	3,107	**3,135**	2,497	2,487	2,751
15	3,343	3,677	3,559	3,761	**3,885**	2,719	2,917	3,117
16	5,117	4,979	5,025	5,307	**5,375**	4,625	4,823	5,093
17	4,959	4,919	5,043	4,983	**5,067**	2,631	4,049	4,639
18	5,151	5,201	5,407	5,429	**5,481**	3,723	4,765	5,119
19	4,803	4,883	5,065	5,163	**5,299**	3,987	4,845	4,981
20	4,999	4,835	4,805	5,087	**5,203**	2,717	4,815	4,823
Σ	78,012	77,934	80,030	81,706	83,050	54,862	72,994	76,806

To account for variance in performance, both versions of Policy-MCTS are run three times on each position. It would not be a fair comparison if the maximum score across runs is taken. Therefore, I only report the run with the highest total score. Plain MCTS is only run once, since the performance is far weaker than any other competitor.

While this iteration of the policy is unable to beat the state-of-the-art, it still produces competitive results. Even though there is a large difference of 6,244 points between the highest total score and Policy-MCTS with informed rollouts, two new high scores were produced. Note that the training curve (Figure 5.16) suggests that training had not yet converged.

There is a large performance gap between Policy-MCTS with random, and guided rollouts, respectively. This implies that as the time budget is increased, the benefits of informed rollouts can outweigh their cost: when given 15 seconds of search time, random rollouts provided better results than informed rollouts, but with two hours of search time, informed rollouts performed the best.

The scores can potentially be further increased in multiple ways. The time budget can be evenly distributed over the average number of moves, as done in SP-MCTS. Alternatively, an exponential budget distribution in which initial moves receive more computation time than later moves can be used. Additionally, an implementation capable of using multiple GPUs could potentially achieve much higher scores with the same policy. Using a transposition table to store predictions will also increase the speed of the system. Smaller gains could be obtained by fine-tuning the search parameters; I did not tune any of the parameters to these positions. Lastly, the simulation count of MCTS can be increased during training. This greatly increases the training time, but it can also lead to a better policy.

Chapter 6

Replacing Rollouts with Value Estimation

Calculating the true value function $V(s)$ in a complex domain is a challenging task. Monte-Carlo methods create an estimate $\widetilde{V}(s)$ by using the average value of all rollouts that start in state s. MCTS creates an estimate by averaging over all rollout values that visited state s in the tree.

Using rollouts has several drawbacks. The depth of terminal nodes, i.e. the number of actions required to reach a terminal state, is domain dependent, and can require hundreds, if not thousands, of actions to reach. Therefore, even with an efficient implementation, rollouts can require a long time to finish. Using a policy can make the rollouts even slower, especially when using a neural network, as demonstrated in the previous chapter.

For this reason, in many domains, rollouts are cut short and the value of the final state in the rollout is estimated by a heuristic. Since heuristics are often more accurate when applied deeper in the search tree, the rollouts can be stopped after a certain number of actions. Alternatively, rollouts can be stopped when the heuristic evaluation becomes extreme, for example, when the predicted outcome of the game is a win (or loss) with high probability.

Pursuing the above idea, a natural question arises: assuming a strong heuristic, say a neural network with millions of parameters, is it necessary to use rollouts at all? That is, can rollouts be entirely replaced with a heuristic? One major contribution of AlphaGo Zero was the complete replacement of rollouts with value estimations [Sil+17], and still managing to surpass all previous Go programs, including those that bested world champions. Since single-player games are a subset of two-player games, a reasonable hypothesis is that rollouts can also be replaced with value estimation in such domains.

The final goal of my work is to completely replace the rollouts in the simulation phase of MCTS with value estimations. To accomplish this task, the policy network from Chapter 5 is replaced by a two-headed network which outputs both a policy and value estimate. The MCTS algorithm from the previous chapter is kept, albeit with small changes, with the main difference being the replacement of rollouts with the network value estimations.

6.1 Training a Two-Headed Network

A network that only outputs a value estimation can be trained concurrently, but separately, from the policy network. Then, each state requires a separate evaluation from the policy and the value networks. Such an approach is highly inefficient, both in terms of training and search. Instead, a single network with two heads can be trained, that given an input state, outputs both a policy and value. Intuitively, the body of the network learns a representation of the game that is shared between both heads. Then, each head uses this representation to make a prediction. That is, given a state s,

$$f_\theta(s) = (p, v), \tag{6.1}$$

where f represents a two-headed network parameterized by θ, p is the policy output, and v represents the value that can be achieved from the given state, i.e. it does not include the current score.

6.1.1 Value Training Target

I will discuss two possible choices for the value training target for a state s, denoted $y(s)$, which are the 1) return value, and 2) the MCTS Q-values $Q(s, a)$.

Return Value

The final game value z is the score of the terminal state reached by the agent. For each state s_i in the sequence of states seen by the agent $(s_0, ..., s_T)$, the return value training target becomes

$$y(s_i) = z - score(s_i), \tag{6.2}$$

where $score(s_i)$ is the current score at state s_i, i.e. rewards collected prior to reaching s_i. Therefore, the learning target for each state is the value of a game started in that state.

One benefit of using the return value as the target is that it is derived from the real value of a final state. That is, it corresponds to on-policy learning, which has been shown to be a far more stable method of learning. One drawback of on-policy learning is that explorative actions taken by the agent are also included, which can cause potentially large changes in the learning targets. This can become a problem in SameGame, as a single action can drastically change the final score.

MCTS Q-Values

A target can also be created from the MCTS state-action values that are calculated while searching. That is,

$$y(s_i) = \max_a Q(s_i, a) - score(s_i), \tag{6.3}$$

which is analogous to Q-learning [WD92], an off-policy learning algorithm. One benefit of off-policy methods is their sample efficiency. Therefore, the MCTS Q-values target might require less data than the return value target to achieve the same performance. However, the reward signal is no longer based on real values, but averages of estimates; the training target is created by averaging over network estimates. This can potentially cause divergence in the learning, as is the case with many off-policy methods that employ function approximation [SB18]. However, since the value estimates are not directly used as the target, as done in Deep Q-Networks [Mni+13], but are based on averages of estimates, learning can be more stable.

6.1.2 Loss

Multiple loss functions can be used for training the value head of the network; I use mean squared-error (MSE). When training a two-headed network, the loss of both heads is added together. This can become an issue when there is a large difference between the losses, as the network will disproportionately optimize the output with the larger loss. Since the range of values in single-player games can be large, the loss of the value head can dominate the total loss. To avoid this problem altogether, a separate value network can be trained, but this introduces issues discussed previously. Another approach is to weigh the two loss components differently. Then, the loss L, given a mini-batch B, is as follows:

$$\begin{aligned} L &= u \cdot \mathrm{MSE}(y, v) + \mathrm{CE}(\pi, p) && (6.4) \\ &= \frac{1}{B} \sum_{i=1}^{B} \left(u \cdot (y(s_i) - v(s_i))^2 - \pi(s_i)^{\mathsf{T}} \log p(s_i) \right), && (6.5) \end{aligned}$$

where I set $u = 16/v_{max}^2$: v_{max} is the highest absolute value in the training or validation set. The numerator for u was derived using manual tuning. The cross-entropy loss is given a weight of one. An alternative approach would be to weigh the value loss by a constant. I experimented with both weighing schemes on 7×7 SameGame, using constant factors of 0.01 and 0.001, and found that setting the value of u based on v_{max} performed better empirically.

6.1.3 Policy Training Changes

MCTS

The same MCTS algorithm as in Chapter 5 is used, with a few changes; I call this algorithm PolicyValue-MCTS (PV-MCTS). The value propagated back to the root, assuming a rollout started in start s, is $v + \mathit{score}(s)$. This is because the search requires the accumulated rewards from the initial state to state s, whereas the current score is not included in the value estimation. The second change is that the tree is no longer reset after taking a move; better average scores were often obtained during training when keeping the search tree. One possible explanation is that, since values of states are estimated by the same network, the increase in exploration caused by resetting the tree does not outweigh the information lost.

Network

The network described in Section 5.4.1 is the body of the two-headed network, with the flatten layer replaced with a global average pooling layer. The value head has an additional fully connected layer with 64 units and ELU activation, and a fully connected layer with 1 unit. The policy head has a fully connected layer with 32 units and ELU activation, which outputs to a fully connected layer with units equal to the number of possible actions and softmax activation.

Training Procedure

Instead of re-initializing the network before training on new data, the same parameters are continuously trained. This was changed as the latter performed better empirically. To avoid the issues discussed in Section 5.2.3, early stopping is replaced with L_2 regularization with a weight of 0.0001. Alternatively, early stopping can be used in conjunction with weight regularization; I did not perform any experiments to determine which approach is superior. The parameters are trained for five epochs on the training buffer every generation.

6.1.4 Results

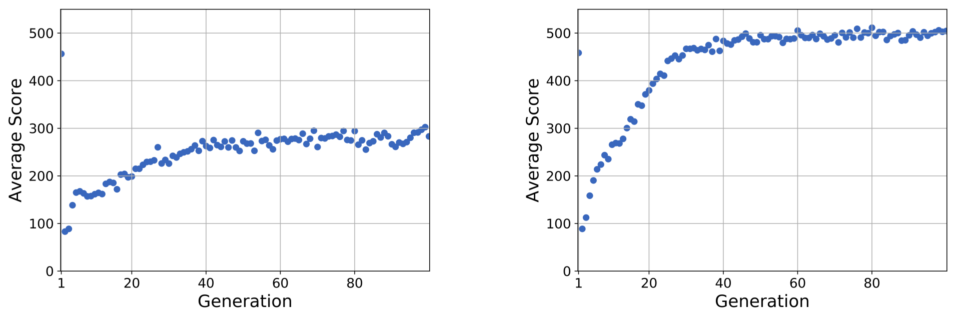
Two networks were trained on 7×7 SameGame, each using one of the above value targets. The parameters are provided in Table 6.1.

The results shown in Figure 6.1 provide some evidence that using MCTS Q-values leads to better performance on 7×7 SameGame. One possible explanation is that, due to the short length of the solutions, which is on average around 14 moves, MCTS Q-values are accurate estimates of the state value. The poor performance of the return value target could be attributed to the fact that states prior to exploratory moves are not excluded from the dataset; since a single wrong move can make the board impossible to clear, the variance of the obtained scores can be large, especially on the smaller 7×7 board.

Table 6.1: Parameters used for training two-headed policy and value networks on 7×7 SameGame.

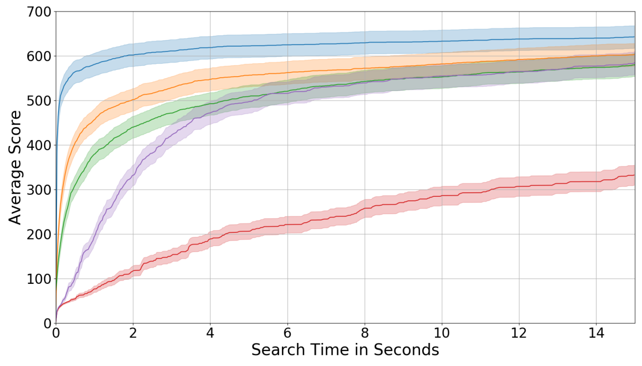
Parameter	Value
Generations	100
Runs/generation	5,000
Simulations	200
C	5
Threads/run	1
Dirichlet noise	0.75
Jump-Start	5,000
Training buffer size	1,500,000
Training-validation split	0.10
Batch size	256

To compare the performance with policy training, both of the above networks are used to run PV-MCTS on the testset of 500 7×7 SameGame boards: 5 runs are executed per board, for a total of 2,500 runs, with 5 seconds of computation time per run. The results shown in Figure 6.2 demonstrate that, as expected, the network trained with the return value target has the worst performance. In contrast, the network trained with the MCTS Q-values target is able to compete with the policies, albeit it requires more time to reach the same performance. This can, in part, be explained by the fact that a rollout will provide a complete solution upon completion, which is not the case when

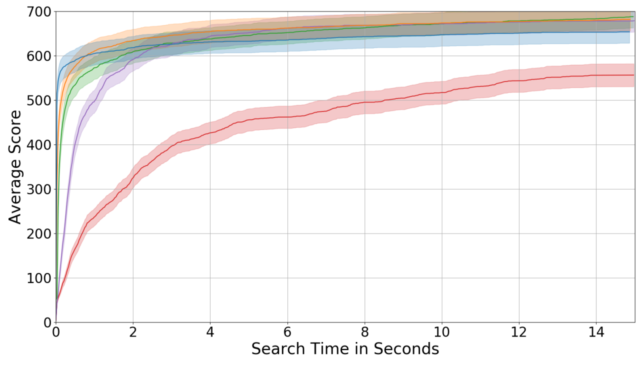


(a) Return Value (b) MCTS Q-values

Figure 6.1: Average score of two-headed policy and value networks during training for 100 generations, using the return value or MCTS Q-values as the training target.



(a) Single-Threaded



(b) Multi-Threaded

— MCTS	— Policy-MCTS Random	— Policy-MCTS Informed
— PV-MCTS; Return Value	— PV-MCTS; MCTS Q-Values	

Figure 6.2: Average score of single-threaded and multi-threaded MCTS, Policy-MCTS with random, and informed rollouts, respectively, and PV-MCTS trained with the return value, and MCTS Q-value training targets, respectively, as a function of search time in seconds. The policy is trained using random rollouts. Each experiment is performed on a test set of 500 randomly generated boards, with 5 runs per board. The shaded areas indicate the 99% confidence bands.

using value estimation. It is important to note that the training time for the two-headed networks is around one day, whereas on the same computer, training a policy network with the parameters provided in Table 5.4 takes around six days; these results are promising, as the two-headed network is able to achieve competitive performance using roughly a sixth of the computational resources.

6.2 Asymmetric Loss

A network that regularly overestimates values of states can become stuck until all the computational budget is spent, whereas one that regularly underestimates might never find a better solution. Therefore, the results from the previous section can possibly be further improved by training the network to be more optimistic or conservative with its value estimations. This can be accomplished by changing the loss function such that the network receives a higher loss for overestimation, which can lead to more conservative estimations, or a higher loss for underestimation, which can lead to more optimistic estimations.

6.2.1 Overestimation vs. Underestimation

An argument could be made that overestimations are worse than underestimations, as they will lead the agent towards "specious pitfalls"; states that "look" nice, but are actually bad, as they will not lead to any terminal states with a high score. Since neighbouring states of a pitfall state are similar in appearance, the network could overestimate on these states as well, which can cause the search to become stuck until all of the search budget is spent. The agent is able to train and fix some of the wrong estimations after a generation is finished, but most of the states with overestimated values will likely not be in the dataset. Therefore, the network might not generalize to those states, which can cause specious pitfalls to appear in the next generation as well.

If the agent suffers from underestimation, then it will never find a better solution unless it is forced to explore sections of the search space which it believes to be bad. In MCTS, random or guided rollouts can stumble upon good solutions in such spaces, but when value estimation is used, the agent might never commit to these sections of the search space. For this to occur, an external force of exploration, such as epsilon greedy action selection, is required, which is not included in my algorithm. Therefore, one can argue that underestimations are worse than overestimations, and the network should be trained in a way that discourages underestimations.

Discouraging one type of estimation more than the other can be achieved by modifying the squared loss as follows:

$$\mathrm{SE}_{asymmetric}(a, b) = \begin{cases} w_{under}(a-b)^2, & a \geq b, \\ w_{over}(a-b)^2, & otherwise. \end{cases} \tag{6.6}$$

Then, $\mathrm{MSE}_{asymmetric}$ is used instead of MSE in Eq. 6.4, which is given by:

$$\mathrm{MSE}_{asymmetric}(y, v) = \frac{1}{B}\sum_{i=1}^{B} \mathrm{SE}_{asymmetric}(y(s_i), v(s_i)). \tag{6.7}$$

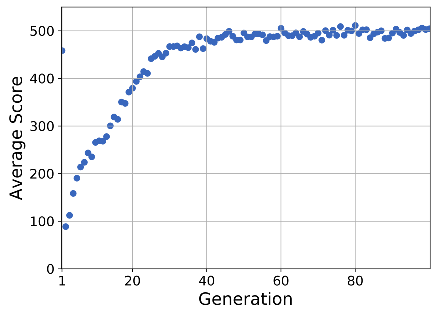
w_{under}, w_{over} are the weights of the loss for underestimation, and overestimation, respectively.

6.2.2 Results

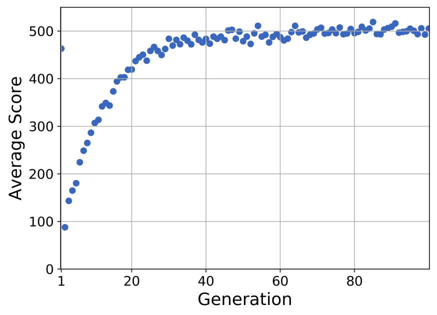
The same parameters provided in Table 6.1 are used to train two networks on 7×7 SameGame, one which prefers overestimations over underestimations, and vice versa. To discourage the network from overestimating, I set $\boldsymbol{w}_{over} = 2$ and $\boldsymbol{w}_{under} = 1/2$, and vice versa for underestimating; these values were not optimized, and were selected to observe the effects of using an asymmetric loss function. Both networks use MCTS Q-values as the training target. Performance during training is provided in Figure 6.3.

The results demonstrate that the network which prefers underestimations performs better initially. In contrast, the network which prefers overestimations not only learns slower, but it also reaches a lower average score in the final generation. One possible explanation for the difference in performance is that PUCT is already optimistic, and therefore, conservative estimations can still lead to better scores, whereas overestimations can cause the search to frequently land in specious pitfalls.

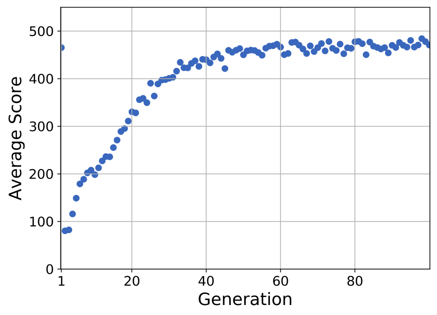
In this chapter, I introduced two value learning targets for training a two-headed network which outputs both a policy and value estimate. Additionally, I suggested the use of an asymmetric loss function to induce more optimistic or conservative estimates. The performance of the two-headed network was competitive with a policy network trained using six times more computational resources. These results are promising, which leads me to hypothesize that



(a) No preference



(b) Underestimations preferred



(c) Overestimations preferred

Figure 6.3: Average score of two-headed policy and value networks during training, for 100 generations. The bottom graphs are policy networks that prefer underestimations or overestimations, respectively, by a factor of 4.

the initial work presented in this chapter may lead to better solutions than policy-only learning.

Chapter 7

Conclusion and Future Work

In this thesis, I have shown that a strong policy can be learned from scratch for single-agent optimization tasks. The training procedure iteratively trains a policy network, using a modified variant of MCTS, in an algorithm called Policy-MCTS.

I introduced two enhancements for MCTS for single-player games. The first was a general value normalization mechanism that is applied at the node level. My second enhancement was a novel virtual loss function, which enabled effective search parallelization. The parallelized version of MCTS was shown to be as effective as the single-threaded version given the same computational budget. Additionally, parallelized Policy-MCTS drastically outperformed the single-threaded version when given a limited time budget.

Using SameGame as a test domain, I demonstrated that Policy-MCTS is more effective than plain MCTS. Furthermore, the performance of my algorithm was shown to be competitive with state-of-the-art search based algorithms on a benchmark set of 20 positions. These results are promising, as this work is the first successful application of reinforcement learning to SameGame.

Lastly, I conducted initial work on adding a value head to the network. The output of this head was used to completely replace the rollouts with value estimation. Two training targets were tested, as well as a novel idea of using an asymmetric loss function to differentiate overestimations and underestimations. The performance of the two-headed network was shown to be competitive with a policy network trained using six times more computational resources. This leads me to hypothesize that using value estimation may lead to better solutions than policy-only learning.

7.1 Future Work

The work presented in this thesis leads to many possible avenues for future work:

- The presented Policy-MCTS algorithm can be improved in multiple ways. For instance, a transposition table can be added to store network predictions, which can drastically reduce the number of network evaluations [Sil+17]. Also, Policy-MCTS with informed rollouts could be further improved by using a smaller, faster network for the simulation stage. Additionally, while some time was spent on understanding and optimizing parameters, the ones used for the larger training runs are likely not optimal. Other optimization methods such as Bayesian Optimization [Che+18] may help in this regard.

- The training procedure can become more efficient by making it asynchronous, with data generation and training occurring at the same time. Using such a process may benefit from training one set of parameters continuously, rather than re-initializing network parameters every generation. More experiments are required to determine if training a single network is better than the approach taken in this thesis.

- When using a two-headed network, value estimates do not necessarily need to replace the rollouts. Instead, they can be mixed with rollout results, which has also been shown to improve performance [ATB17; Sil+16]. The work presented in Chapter 6 can be directly applied to this scenario by simply introducing a mixing parameter. When using value estimates, it might be beneficial to train the network with simple games first, and progressively increase the difficulty as training progresses; this process is known as curriculum learning [Ben+09]. For example, in SameGame, training can be started with small boards first, and once the network learns to solve the smaller boards, the size can be increased.

- While I have claimed that the presented training procedure, MCTS, and Policy-MCTS are general, all experiments in this thesis were conducted in a single domain only. Note that the only SameGame specific modification to my algorithm was the simplification of actions during search. Testing these algorithms in other domains would further strengthen my claims. The most domain-dependent part of this work is the neural network architecture, which was hand-crafted and tuned to the specific domain. The rest, including value normalization, tree parallelization, and the training procedure can be used directly without any changes, but will likely require optimization of parameters. An in-depth study of my parallelization strategy, by comparing it with leaf and root parallelization in multiple domains, is also required to truly test the effectiveness of the method.

- Another direction for research is to use a different expert; MCTS might not be the ideal choice for the single-player setting. Any search algorithm that acts as an effective policy improvement operator could be used. This implies that the algorithm must be able to use a policy to guide the search in some way. For example, since Nested Monte-Carlo Search can be combined with policy learning, as done in Nested Rollout Policy Adaptation, it could also be used in my algorithm.

References

[ATB17] Thomas Anthony, Zheng Tian, and David Barber. “Thinking fast and slow with deep learning and tree search.” In: *Advances in Neural Information Processing Systems*. 2017, pp. 5360–5370. 2, 12, 40, 45–47, 86

[AZH11] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. “Learning heuristic functions for large state spaces.” In: *Artificial Intelligence* 175.16-17 (2011), pp. 2075–2098. 13

[AHH10] Broderick Arneson, Ryan B Hayward, and Philip Henderson. “Monte Carlo tree search in Hex.” In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 251–258. 2, 18

[ACF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem.” In: *Machine learning* 47.2-3 (2002), pp. 235–256. 17, 18

[BW12] Hendrik Baier and Mark HM Winands. “Nested Monte-Carlo Tree Search for Online Planning in Large MDPs.” In: *ECAI*. Vol. 242. 2012, pp. 109–114. 11, 14

[BF09a] Radha-Krishna Balla and Alan Fern. “UCT for tactical assault planning in real-time strategy games.” In: *Twenty-First International Joint Conference on Artificial Intelligence*. 2009. 18

[Ben+09] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. “Curriculum learning.” In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 41–48. 86

[BF09b] Yngvi Bjornsson and Hilmar Finnsson. “Cadiaplayer: A simulation-based general game player.” In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.1 (2009), pp. 4–15. 14

[Bro+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. “A survey of Monte Carlo tree search methods.” In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43. 2, 18, 20, 21, 58

[Caz09] Tristan Cazenave. “Nested Monte-Carlo search.” In: *Twenty-First International Joint Conference on Artificial Intelligence*. 2009. 2, 10, 33

[CJ07] Tristan Cazenave and Nicolas Jouandeau. "On the parallelization of UCT." In: *proceedings of the Computer Games Workshop*. Citeseer. 2007, pp. 93–101. 23, 24

[CT12] Tristan Cazenave and Fabien Teytaud. "Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows." In: *International Conference on Learning and Intelligent Optimization*. Springer. 2012, pp. 42–54. 11

[CWD08] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. "Parallel Monte-Carlo tree search." In: *International Conference on Computers and Games*. Springer. 2008, pp. 60–71. 16, 23, 25, 26

[Che+18] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. "Bayesian optimization in alphago." In: *arXiv preprint arXiv:1812.06855* (2018). 86

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." In: *arXiv preprint arXiv:1511.07289* (2015). 48

[Cou06] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search." In: *International conference on computers and games*. Springer. 2006, pp. 72–83. 14

[DLR07] René De Koster, Tho Le-Duc, and Kees Jan Roodbergen. "Design and control of warehouse order picking: A literature review." In: *European journal of operational research* 182.2 (2007), pp. 481–501. 1

[EC16] Stefan Edelkamp and Tristan Cazenave. "Improved diversity in nested rollout policy adaptation." In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2016, pp. 43–55. 72

[EG14] Stefan Edelkamp and Max Gath. "Solving Single Vehicle Pickup and Delivery Problems with Time Windows and Capacity Constraints using Nested Monte-Carlo Search." In: *ICAART (1)*. 2014, pp. 22–33. 11

[EGR14] Stefan Edelkamp, Max Gath, and Moritz Rohde. "Monte-Carlo tree search for 3D packing with object orientation." In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2014, pp. 285–296. 11

[EM09] Markus Enzenberger and Martin Müller. "A lock-free multithreaded Monte-Carlo tree search algorithm." In: *Advances in Computer Games*. Springer. 2009, pp. 14–20. 25, 37

[Gel+12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. "The grand challenge of computer Go: Monte Carlo tree search and extensions." In: *Communications of the ACM* 55.3 (2012), pp. 106–113. 2, 18

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. 2

[HI15] Daniel Hennes and Dario Izzo. "Interplanetary trajectory planning with Monte Carlo tree search." In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015. 21

[Hol92] John H Holland. "Genetic algorithms." In: *Scientific american* 267.1 (1992), pp. 66–73. 2

[Hor+06] Gregory Hornby, Al Globus, Derek Linden, and Jason Lohn. "Automated antenna design with evolutionary algorithms." In: *Space 2006*. 2006, p. 7242. 1

[Hua+13] Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. "MoHex 2.0: a pattern-based MCTS Hex player." In: *International Conference on Computers and Games*. Springer. 2013, pp. 60–71. 2, 18

[JGT14] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. "Monte Mario: platforming with MCTS." In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM. 2014, pp. 293–300. 20

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014). 49

[Kle15] Simon Klein. *Attacking SameGame using Monte-Carlo tree search: using randomness as guidance in puzzles*. 2015. 28

[KS06] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo planning." In: *European conference on machine learning*. Springer. 2006, pp. 282–293. 2, 18, 27, 29, 42

[Kor85] Richard E Korf. "Depth-first iterative-deepening: An optimal admissible tree search." In: *Artificial intelligence* 27.1 (1985), pp. 97–109. 2

[KP14] Volodymyr Kuleshov and Doina Precup. "Algorithms for multi-armed bandit problems." In: *arXiv preprint arXiv:1402.6028* (2014). 17

[LR85] Tze Leung Lai and Herbert Robbins. "Asymptotically efficient adaptive allocation rules." In: *Advances in applied mathematics* 6.1 (1985), pp. 4–22. 17

[Lat+18] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S Dahl, Amine Kerkeni, and Karim Beguir. "Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization." In: *arXiv preprint arXiv:1807.01672* (2018). 12

[McA+18] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. "Solving the Rubik's Cube Without Human Knowledge." In: *arXiv preprint arXiv:1805.07470* (2018). 13

[Mni+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013). 77

[Nag19] Andrzej Nagórko. "Parallel Nested Rollout Policy Adaptation." In: *2019 IEEE Conference on Games (CoG)*. IEEE. 2019, pp. 1–7. 11

[NC17] Benjamin Negrevergne and Tristan Cazenave. "Distributed Nested Rollout Policy for SameGame." In: *Workshop on Computer Games*. Springer. 2017, pp. 108–120. 11, 72

[PWL14] Tom Pepels, Mark HM Winands, and Marc Lanctot. "Real-time Monte Carlo tree search in Ms Pac-Man." In: *IEEE Transactions on Computational Intelligence and AI in games* 6.3 (2014), pp. 245–257. 2, 18

[Per+15] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. "The 2014 general video game playing competition." In: *IEEE Transactions on Computational Intelligence and AI in Games* 8.3 (2015), pp. 229–243. 14, 18

[Per+19] D Perez, Jialin Liu, A Abdel Samea Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. "General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms." In: *IEEE Transactions on Games* (2019). 2, 14

[Ros11a] Christopher D Rosin. "Multi-armed bandits with episode context." In: *Annals of Mathematics and Artificial Intelligence* 61.3 (2011), pp. 203–230. 19

[Ros11b] Christopher D Rosin. "Nested rollout policy adaptation for Monte Carlo tree search." In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011. 2, 11

[Sch+12] Maarten PD Schadd, Mark HM Winands, Mandy JW Tak, and Jos WHM Uiterwijk. "Single-player Monte-Carlo tree search for SameGame." In: *Knowledge-Based Systems* 34 (2012), pp. 3–11. 2, 10, 18, 20, 43, 70, 72

[Sch+08] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, and Huib Aldewereld. “Addressing NP-complete puzzles with Monte-Carlo methods.” In: *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning.* Vol. 9. 2008, pp. 55–61. 9

[SBH08] Jan Schäfer, Michael Buro, and Knut Hartmann. “The UCT algorithm applied to games with imperfect information.” In: *Diploma, Otto-Von-Guericke Univ. Magdeburg, Magdeburg, Germany* (2008). 18

[Seg10] Richard B Segal. “On the scalability of parallel UCT.” In: *International Conference on Computers and Games.* Springer. 2010, pp. 36–47. 25

[Sil09] David Silver. “Reinforcement learning and simulation-based search in computer Go.” In: (2009). 12

[Sil+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search.” In: *nature* 529.7587 (2016), p. 484. 14, 19, 21, 41, 86

[Sil+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. “A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play.” In: *Science* 362.6419 (2018), pp. 1140–1144. 2, 11, 19, 30, 40

[Sil+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. “Mastering the game of go without human knowledge.” In: *Nature* 550.7676 (2017), p. 354. 19, 46, 75, 86

[Stu08] Nathan Sturtevant. “An analysis of UCT in multi-player games.” In: *ICGA Journal* 31.4 (2008), pp. 195–208. 18

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2018. 2, 8, 77

[ŚMO15] Maciej Świechowski, Jacek Mańdziuk, and Yew Soon Ong. “Specialization of a UCT-based general game playing program to single-player games.” In: *IEEE Transactions on Computational Intelligence and AI in Games* 8.3 (2015), pp. 218–228. 21

[The+10] Christophe Theys, Olli Bräysy, Wout Dullaert, and Birger Raa. “Using a TSP heuristic for routing order pickers in warehouses.” In: *European Journal of Operational Research* 200.3 (2010), pp. 755–763. 2

[VSS17] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. “On monte carlo tree search and reinforcement learning.” In: *Journal of Artificial Intelligence Research* 60 (2017), pp. 881–936. 28

[WD92] Christopher JCH Watkins and Peter Dayan. “Q-learning.” In: *Machine learning* 8.3-4 (1992), pp. 279–292. 77