Universiteit
Leiden
The Netherlands

Searching by learning: Exploring artificial general intelligence on small board games by deep reinforcement learning

Wang, H.

Citation

Wang, H. (2021, September 7). *Searching by learning: Exploring artificial general intelligence on small board games by deep reinforcement learning*. Retrieved from https://hdl.handle.net/1887/3209232

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3209232

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle https://hdl.handle.net/1887/3209232 holds various files of this Leiden University dissertation.

Author: Wang, H.
Title: Searching by learning: Exploring artificial general intelligence on small board games by deep reinforcement learning
Issue Date: 2021-09-07

Searching by Learning: Exploring Artificial General Intelligence on Small Board Games by Deep Reinforcement Learning

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 7 september 2021
klokke 16.15 uur

door

Hui Wang

geboren te Anhui, China

in 1992

Promotiecommissie

Promotors:	Dr. Michael Emmerich	
	Prof. Dr. Aske Plaat	
Co-promotor:	Dr. Mike Preuss	
Overige leden:	Prof. Dr. Thomas Bäck	
	Prof. Dr. Marcello Bonsangue	
	Prof. Dr. Joost Batenburg	
	Prof. Dr. Mark Winands	Maastricht University
	Dr. Mitra Baratchi	
	Dr. Thomas Moerland	
	Dr. Ingo Schwab	Karlsruhe University of Applied Science

Copyright © 2021 Hui Wang All Rights Reserved

ISBN: 978-94-6419-253-7

Het onderzoek beschreven in dit proefschrift is uitgevoerd aan het Leiden Institute of Advanced Computer Science (LIACS, Universiteit Leiden).

This Research is financially supported by the China Scholarship Council (CSC), CSC No. 201706990015.

Contents

Chapter 1

Introduction

1.1 Background

Modern Artificial Intelligence (AI) research began in the mid-1950s [1]. Mainstream AI scientists focused on the narrow AI research which usually aims to solve a single sub-problem (specific task). In 1990s, some researchers started to try to develop artificial general intelligence (AGI), also known as strong AI [2, 3], by combining the programs that solve various sub-problems (different tasks). AGI is the hypothetical intelligence of a computer program that has the capacity to understand or learn any intellectual task that a human being can [4]. Although current AI techniques have achieved impressive performance in mastering specific tasks, AGI it is still speculated to be decades away [5]. Therefore, it is useful to further study how far the current techniques (especially in reinforcement learning landscape [6]) can bring us to AGI, which I will do in this thesis. I will now list a few challenges in AGI to frame the contributions in this thesis.

Reinforcement Learning

There are many of AI techniques, such as searching, reasoning, pattern recognition and learning that achieve impressive successes [7, 8, 9, 10]. In reinforcement learning, as this thesis will emphasize, searching and learning are both important. Well-known techniques are Monte Carlo Search (MCS) [11], Monte Carlo Tree Search (MCTS) [7, 12] and table based (or neural network based) Q-learning [13, 14, 15]. These techniques have shown impressive capability in

mastering practical problems, especially the recent success of training the AlphaGo series of programs playing Go, Chess and Shogi [10, 16, 17]. The AlphaGo programs use an architecture that combines MCTS and neural network training, which has become a highly successful paradigm of deep reinforcement learning for high-dimensional problems. Before neural networks were used, table-based Q-learning has been employed in combination with MCTS, resulting to the same structure combining online search and offline learning [18]. It is interesting to study this approach.

General Game Playing

General game playing (GGP) is a well-known testbed for AGI. The goal of GGP is to play previously unknown board games, where the rules are not known in advance. The program must play the game without human intervention. The games are described using a standard game description language; legal moves can be automatically generated. Thus, in writing the program, the GGP-author can use search and learning techniques. For example, MCTS and its variants achieve quite good performance on the GGP system [19, 20, 21, 22]. However, there are few works that apply deep reinforcement learning to play GGP games. Therefore, a table based Q-learning should be investigated for GGP to enter the deep reinforcement learning era.

AlphaZero

AlphaZero [10] can also be regarded as an AGI framework. AlphaZero provides a general framework to play Go, Chess and Shogi. In fact, it is implemented to play a class of two-player zero sum games. Therefore, it is a promising testbed for AGI with deep reinforcement learning. The landmark achievements of the AlphaGo series of programs have created a large research interest into self-play in reinforcement learning. In self-play, MCTS is used to train a deep neural network, that is then used in tree searches. Training itself is governed by many hyper-parameters. There has been surprisingly little research on design choices for hyper-parameter values and loss functions, presumably because of the prohibitive computational cost to explore the parameter space.

Expert Data

In addition, we note that the creators of AlphaGo use data from expert games for AlphaGo, but not for AlphaGo Zero nor for AlphaZero, which are so-called tabula rasa approaches. However, a further program, in this series, AlphaStar,

for StarCraft, does use expert data again [23]. There is little research into the necessity of expert data. Well studied MCTS enhancements, such as Rapid Action Value Estimation (RAVE) [24, 25, 26] can improve the performance of table-based Q-learning, but there is no research reported on applying such enhancements to the AlphaZero framework.

Single Agent Combinatorial Optimization

Last but not the least, AlphaZero-like deep reinforcement learning is initially developed for two-player games, where it is highly successful. Therefore, it is interesting to see how it could be transferred to deal with single agent combinatorial optimization games and to benefit the solution of combinatorial problems in computer science.

Overview

Overall, in this dissertation, we focus on GGP and the AlphaZero framework to test promising and novel ideas to explore AGI. The computational demands of these approaches are high. By using small board games we are able to perform many experiments while retaining many aspects of larger games.

Specifically, in Chapter 2, we assess the potential of classical Q-learning in GGP, together with dynamic ϵ and MCS enhancements. Then, in Chapter 3, we investigate 12 hyper-parameters in an AlphaZero-like self-play algorithm and evaluate how these parameters contribute to training. Next, Chapter 4 evaluates the alternative loss functions of AlphaZero-like self-play to study the importance of policy function and value function. Subsequently, we propose a warm-start search enhancement method to boost training at the start phase of self-play training in Chapter 5, which evidences the necessity of expert data and we show how these data can be generated by MCTS enhancements. In the following Chapter 6, we further propose an adaptive warm-start method to dynamically control the warm-start length during training. In Chapter 7, we embed a ranked reward algorithm within AlphaZero-like self-play to challenge a well-studied single player combinatorial game, Morpion Solitaire, and obtain a near human level result as a first attempt.

In Fig 1.1, the structure of this dissertation is depicted as a diagram. The diagram shows that the main work of our thesis is based on two frameworks (GGP and AlphaZero-like self-play). For each framework, different learning and searching techniques are investigated in different chapters.

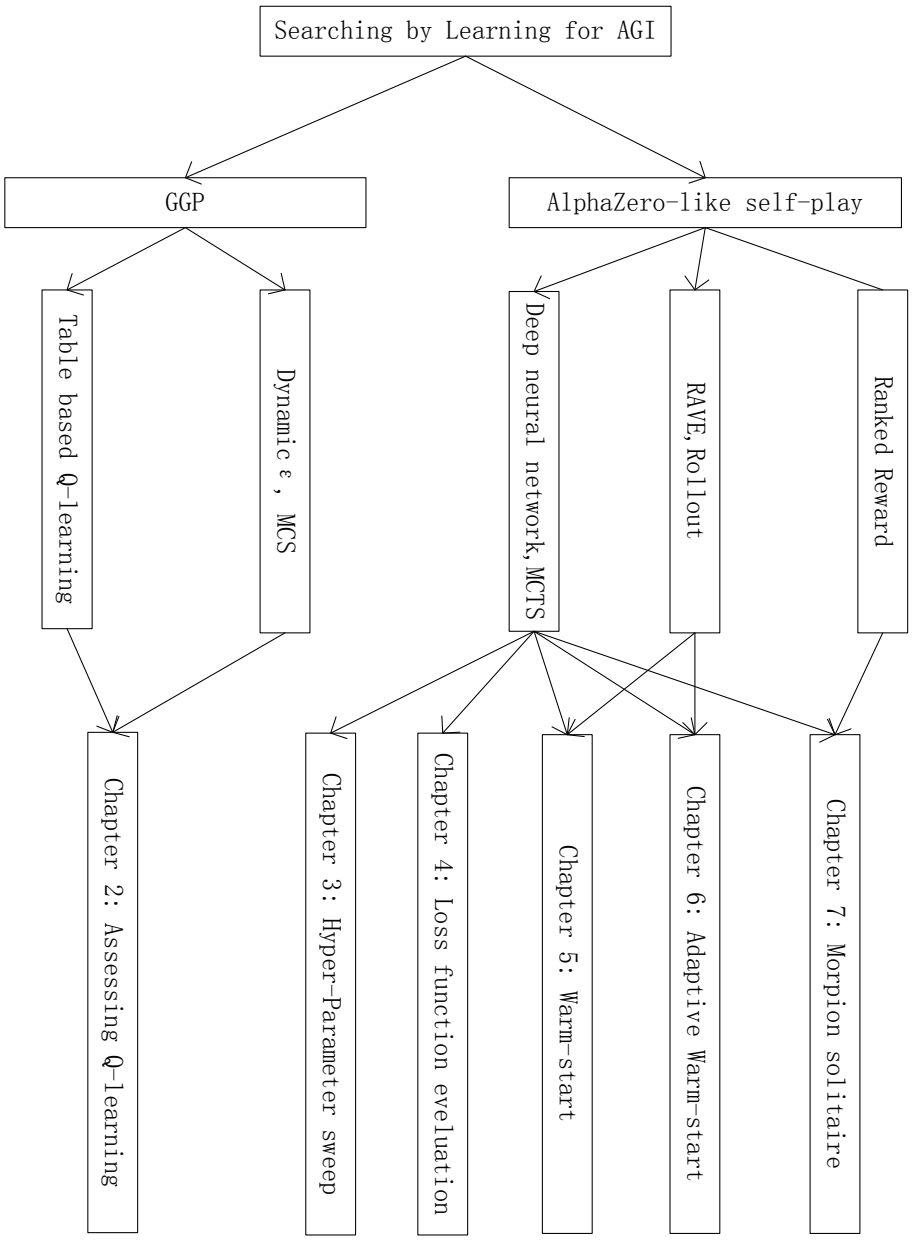


Figure 1.1: A general structure of this dissertation

1.2 Research Questions

In the following, we list the research questions of this thesis and sketch the approaches used to find answers to them.

RQ1 (Chapter 2) How to assess the potential of classical Q-learning in GGP? In order to build a bridge between GGP and deep reinforcement learning, the initial step is to assess the potential of table-based Q-learning on GGP system. We test different values for fixed ϵ and then propose a dynamic ϵ enhancement, where the ϵ is used to balance the exploration and exploitation of Q-learning. Furthermore, we introduce another enhancement with MCS which is to generate better training examples at the beginning when the Q-table has no record of the state. All experiments are tested on small board games, such as Tic-Tac-Toe, 3×3 Hex and 4×4 Connect Four. In order to assess convergence with different board size, 3×3, 4×4 and 5×5 Tic-Tac-Toe are also investigated.

RQ2 (Chapter 3) How hyper-parameters contribute to AlphaZero-like self-play? A hyper-parameter sweep is computationally expensive; Little is known on how to set these hyper-parameters, there is a need to provide insight into how the hyper-parameters contribute to training efficiency. AlphaZero-like self-play can be divided into three stages, namely *self-play*, *neural network training* and *arena comparison*. We identify 12 main hyper-parameters for this framework. The explanation of each hyper-parameter is given, and a light hyper-parameter sweep is performed. This sweep provides an overview of the hyper-parameter contributions. Besides, a further study of the interaction between important hyper-parameters is also performed in this dissertation, which helps to understand the trade off between searching and learning.

RQ3 (Chapter 4) How alternative loss functions work in AlphaZero-like self-play? Players in AlphaZero consist of a combination of MCTS and a deep neural network, that is trained using self-play. A unified deep neural network is used, which has a policy-head and a value-head. During training, the optimizer minimizes the **sum** of policy loss and value loss. However, it is not clear if and under which circumstances other formulations of the loss function are better. Therefore, we perform experiments with different combinations of these two minimization targets. In contrast to many recent papers who adopt single run experiments and use the whole history Elo ratings from self-play, we propose to use repeated runs. The results show that this method can describe the training performance quite

well within each training run. Because of a high self-play bias a final best player Elo rating is adopted to evaluate the playing strength in a direct competition between the evolved players, inspired by the approach reported by the AlphaZero team.

RQ4 (Chapter 5) Can MCTS enhancements be used to replace human experts to improve the AlphaZero-like self-play? AlphaZero's design is purely based on self-play and makes no use of labeled expert data or domain specific enhancements; it is designed to learn from scratch. We propose a novel approach to deal with this cold-start problem by employing simple search enhancements at the beginning phase of self-play training. We use Rollout, RAVE and dynamically weighted combinations of these with the neural network, and Rolling Horizon Evolutionary Algorithms (RHEA).

RQ5 (Chapter 6) How to control the warm-start length? While tuning warm-start length for different enhancements, the results show that it is costly and usually unstable. Therefore we propose an adaptive method to control the warm-start length of using MCTS enhancements by employing an arena to determine whether the enhancement is not better any more. The experimental results show that our approach works better than the fixed I', especially for deep, tactical, games (Othello and Connect Four). We conjecture that the adaptive value for I' is also influenced by the size of the game, and that on average I' will increase with game size. We conclude that AlphaZero-like deep reinforcement learning benefits from adaptive rollout based warm-start, as RAVE did for rollout-based reinforcement learning 15 years ago.

RQ6 (Chapter 7) Can AlphaZero-like self-play be used to master complex single player combinatorial optimization games? Morpion Solitaire is a popular single player complex combinatorial optimization game, performed with paper and pencil [27, 28]. Due to its large state space (on the order of the game of Go) traditional search algorithms, such as MCTS, have not been able to find good solutions. A new algorithm, Nested Rollout Policy Adaptation, was able to find a new record of 82 steps, albeit with large computational resources [29]. Morpion Solitaire has never been studied in a deep reinforcement learning framework. A challenge of Morpion Solitaire is that the state space is sparse, there are few win/loss signals. Therefore, we use an approach known as ranked reward to create a reinforcement learning self-play framework for Morpion Solitaire. This enables us to find medium-quality solutions with reasonable computational effort.

Our record is a 67 steps solution, which is very close to the human best (68) without any other adaptation to the problem than using ranked reward.

1.3 Dissertation Outline

The dissertation outline is described in this section. For each main chapter of this dissertation, there is at least one publication by the author. A brief outline of this work is presented as follows.

★ Chapter 2 introduces the GGP system and the definition of classical Q-learning. In addition, two enhancements (dynamic ϵ and QM-learning) of classical Q-learning are proposed. The classical Q-learning and proposed enhancements are assessed in a GGP system to play several different games (different size of Tic-Tac-Toe, 3×3 Hex and 4×4 ConnectFour). The contents of this chapter are published in a preprint [30] and a conference paper [31] (**best regular paper award**).

> **Wang H.**, Emmerich M., Plaat A. (2018) Monte Carlo Q-learning for General Game Playing. arXiv preprint 1802.05944.
>
> **Wang H.**, Emmerich M., Plaat A. (2019) Assessing the Potential of Classical Q-learning in General Game Playing. In: Atzmueller M., Duivesteijn W. (eds) Artificial Intelligence. BNAIC 2018. Communications in Computer and Information Science, vol 1021. Springer.

★ Chapter 3 introduces the AlphaZero-like self-play framework with three stages in a single iterative loop, and identities 12 potentially important hyper-parameters. A hyper-parameter sweep is performed for every hyper-parameter and moreover, further experiments of four selected more interesting hyper-parameters are also performed and evaluated. Parts of this chapter are published in preprints [32, 33].

> **Wang H.**, Emmerich M., Preuss M., Plaat A. (2019) Hyper-Parameter Sweep on AlphaZero General. arXiv preprint 1903.08129.
>
> **Wang H.**, Emmerich M., Preuss M., Plaat A. (2020) Analysis of Hyper-Parameters for Small Games: Iterations or Epochs in Self-Play?. arXiv preprint 2003.05988, submitted to journal.

★ Chapter 4 introduces the default loss function of AlphaZero-like self-play, and three alternative loss functions. A running Elo is computed and a full

tournament Elo is also employed. Parts of this chapter are published in a conference paper [34] and a preprint [33].

> **Wang H.**, Emmerich M., Preuss M., Plaat A. (2019) Alternative loss functions in AlphaZero-like self-play. 2019 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE, pp. 155-162.
>
> **Wang H.**, Emmerich M., Preuss M., Plaat A. (2020) Analysis of Hyper-Parameters for Small Games: Iterations or Epochs in Self-Play? arXiv preprint 2003.05988, submitted to journal.

★ Chapter 5 investigates the AlphaZero-like self-play start phase by employing MCTS enhancements to improve training performance. The description and analysis of RAVE and RHEA is provided. The work of the chapter is published in a conference paper [35].

> **Wang H.**, Preuss M., Plaat A. (2020) Warm-Start AlphaZero Self-play Search Enhancements. In: Bäck T. et al. (eds) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020. Lecture Notes in Computer Science, vol 12270. Springer.

★ Chapter 6 introduces the adaptive warm-start method, and a parameter tuning for fixed I' is also provided. The work of the chapter is published as a preprint [36].

> **Wang H.**, Preuss M., Plaat A. (2021) Adaptive Warm-Start MCTS in AlphaZero-like Deep Reinforcement Learning. arXiv preprint 2105.06136, submitted to conference.

★ Chapter 7 introduces how to embed the ranked reward mechanism into AlphaZero-like self-play. A description of Morpion Solitair game is provided. And a near-human level solution with 67 steps is presented. The work of the chapter is published in a conference paper [37].

> **Wang, H.**, Preuss, M., Emmerich, M. and Plaat, A. (2020) Tackling Morpion Solitaire with AlphaZero-like Ranked Reward Reinforcement Learning. 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). IEEE, pp. 149-152

★ Chapter 8 summarizes the contributions of this dissertation and highlights points to some interesting directions for future work.

★ In addition to the aforementioned publications, another publication of the author is [38]. This publication is related to the research, but was not part of this thesis.

> **Wang H.**, Tang Y., Liu J., Chen W. (2018) A Search Optimization Method for Rule Learning in Board Games. In: Geng X., Kang BH. (eds) PRICAI 2018: Trends in Artificial Intelligence. PRICAI 2018. Lecture Notes in Computer Science, vol 11013. Springer.

Chapter 2

Classical Q-learning in GGP

2.1 Introduction

Traditional game playing programs are written to play a single specific game, such as Chess, or Go. The aim of *General* Game Playing [39] (GGP) is to create adaptive game playing programs; programs that can play more than one game well. To this end, GGP uses a so-called Game Description Language (GDL) [40]. GDL-authors write game-descriptions that specify the rules of a game. The challenge for GGP-authors is to write a GGP player that will play any game well. GGP players should ensure that a wide range of GDL-games can be played well. Comprehensive tool-suites exist to help researchers write GGP and GDL programs, and an active research community exists [38, 41, 42, 43].

The GGP model follows the state/action/result paradigm of reinforcement learning [6], a paradigm that has yielded many successful problem solving algorithms. For example, the successes of AlphaGo are based on two reinforcement learning algorithms, MCTS [7] and Deep Q-learning (DQN) [16, 44]. MCTS, in particular, has been successful in GGP [45]. However, few works analyze the potential of Q-learning for GGP, not to mention DQN. The aim of this chapter is to be a basis for further research of DQN for GGP and applying promising MCTS enhancements for neural network based reinforcement learning approaches.

Q-learning with deep neural networks requires extensive computational resources. Table-based Q-learning might offer a viable alternative for small games. Therefore, following Banerjee [46], in this chapter the convergence speed of table-based Q-learning is addressed. Three small two-player zero-sum games: Tic-Tac-Toe,

Hex and Connect Four, and table-based Q-learning are used. This chapter introduces two enhancements: dynamic ϵ, and, borrowing an idea from [18], a new version of Q-learning is created, inserting MCS into Q-learning, using online search for offline learning[1].

The main contributions of this chapter can be summarized as follows:

1. **Dynamic ϵ:** The classical Q-learning is evaluated, finding (1) that Q-learning works and converges in GGP, and (2) that Q-learning with a dynamic ϵ can enhance the performance of TD(λ)[2] baseline with a fixed ϵ [46].

2. **QM-learning:** To further improve performance the classical Q-learning is enhanced by adding a modest amount of Monte Carlo lookahead (QM-Player) [47]. This improves the convergence rate of Q-learning, and shows that online search can also improve the offline learning in GGP.

The chapter is organized as follows. Sect. 2.2 presents related work and recalls basic concepts of GGP and reinforcement learning. Sect. 2.3 presents the designs of the QPlayer with fixed and dynamic ϵ and QMPlayer for two-player zero-sum games for GGP to assess the potential of classical Q-learning in detail. Sect. 2.4 presents the experimental results. Sect. 2.5 summarizes the chapter and discusses directions for future work.

2.2 Related Work and Preliminaries

2.2.1 GGP

A General Game Player must be able to accept formal GDL descriptions of a game and play games effectively without human intervention [42], where the GDL has been defined to describe the game rules [48]. An interpreter program [43] generates legal moves (actions) for a specific board (state). Furthermore, a Game Manager (GM) is at the center of the software ecosystem. The GM interacts with game players through TCP/IP protocol to control the match. The GM manages game descriptions and matches records and temporary states of matches while the game is running. The system also contains a viewer interface for users who are interested in running matches and a monitor to analyze the match process.

[1] source code: https://github.com/wh1992v/ggp-rl

[2] one of temporal difference methods, see [6]

2.2.2 Reinforcement Learning

Since Watkins proposed Q-learning in 1989 [49], much progress has been made in reinforcement learning [50, 51, 52, 53]. However, few works report on the use of Q-learning in GGP. In [46], Banerjee and Stone propose a method to create a general game player to study knowledge transfer [54], combining Q-learning and GGP. Their aim is to improve the performance of Q-learning by transferring the knowledge learned in one game to a new, but related, game. They found knowledge transfer with Q-learning to be expensive. In [18], Gelly and Silver combine online and offline knowledge to improve learning performance.

Recently, DeepMind published work on mastering Chess and Shogi by self-play with a deep, generalized reinforcement learning algorithm [10]. With a series of landmark publications from AlphaGo to AlphaZero [10, 16, 17], these works showcase the promise of general reinforcement learning algorithms. However, such learning algorithms are very resource-intensive and typically require special GPU/TPU hardware. Furthermore, the neural network-based approach is quite inaccessible to theoretical analysis. Therefore, this chapter starts from studying the performance of table-based Q-learning.

In GGP, variants of MCTS [7] are used with great success [45]. Tom Vodopivec et al. studied MCTS with planning methods inspired by reinforcement learning [55]. Méhat et al. combined UCT (Upper Confidence bound applied to Trees) and nested MCS for single-player GGP [56]. Cazenave et al. further proposed a nested MCS for two-player games [57]. Monte Carlo techniques have proved a viable approach for searching intractable game spaces and other optimization problems [58]. Therefore, in this chapter MCS is combined as an enhancement to improve performance.

2.2.3 Q-learning

A basic distinction between reinforcement learning methods is that of "on-policy" and "off-policy" methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy *different* from that used to make decisions [6]. Q-learning is an off-policy method. The reinforcement learning model consists of an *agent*, a set of states S, and a set of actions A available in state S [6]. The agent can move to the next state s', $s' \in S$ from state s after following action a, $a \in A$, denoted as $s \xrightarrow{a} s'$. After finishing the action a, the agent gets an immediate reward $R(s, a)$, usually a numerical score. The cumulative return of current state s by taking

the action a, denoted as $Q(s, a)$, is a weighted sum, calculated by $R(s, a)$ and the maximum $Q(s', a')$ value of all next states:

$$Q(s, a) = R(s, a) + \gamma \; max_{a'} Q(s', a') \tag{2.1}$$

where $a' \in A'$ and A' is the set of actions available in state s'. γ is the discount factor of $max_{a'} Q(s', a')$ for next state s'. $Q(s, a)$ can be updated by online interactions with the environment using the following rule:

$$Q(s, a) \leftarrow (1 - \alpha) \; Q(s, a) + \alpha \; (\; R(s, a) + \gamma \; max_{a'} Q(s', a')) \tag{2.2}$$

where $\alpha \in [0, 1]$ is the learning rate. The Q-values are guaranteed to converge after iteratively updating.

2.3 Design

2.3.1 Classical Q-learning for Two-Player Games

GGP games in experiments for this chapter are two-player zero-sum games that alternate moves. Therefore, the same rule can be used, see Algorithm 1 line 5, to create $R(s, a)$, rather than to use a reward table. In experiments, $R(s, a) = 0$ is set for non-terminal states, and call the *getGoal*() function for terminal states. In order to improve the learning effectiveness, the $Q(s, a)$ table is updated only at the end of the match. During offline learning, QPlayer uses an ϵ-greedy strategy to balance exploration and exploitation towards convergence. While the ϵ-greedy strategy is enabled, QPlayer will perform a random action. Otherwise, QPlayer will perform the best action according to Q(S,A) table. If no record matches current state, QPlayer will perform a random action. The pseudo code for this algorithm is given in Algorithm 1.

Algorithm 1 Classical Q-learning Player with Static ϵ

```
1: function QPlayer(current state s, learning rate α, discount factor γ, Q table: Q(S, A))
2:     for each match do
3:         if s terminates then
4:             for each (s, a) from end to the start in current match record do
5:                 R(s,a) =s′ is terminal state? getGoal(s′, myrole) : 0
6:                 Update Q(s, a) ← (1−α) Q(s, a)+α ( R(s, a)+γ max_a′ Q(s′, a′))
7:         else
8:             if ϵ-greedy is enabled then
9:                 selected_action = Random()
10:            else
11:                selected_action = SelectFromQTable()
12:                if no s record in Q(S, A) then
13:                    selected_action = Random()
14:                                          ▷ To be changed for different versions
15:            performAction(s, selected_action)
16:    return Q(S, A)
```

2.3.2 Dynamic ϵ Enhancement

In contrast to the baseline of [46], which uses a fixed ϵ value, a dynamically decreasing ϵ-greedy Q-learning [50] is used. In the implementation, the function

$$\epsilon(m) = \begin{cases} a(\cos(\frac{m}{2l}\pi)) + b & m \leq l \\ 0 & m > l \end{cases} \tag{2.3}$$

is used for ϵ, where m is the current match count, and l is a number of matches set in advance to control the decaying speed of ϵ. During offline learning, if $m = l$, ϵ decreases to 0. a and b is set to limit the range of ϵ, where $\epsilon \in [b, a + b]$, $a, b \geq 0$ and $a + b \leq 1$. The player generates a random number num where $num \in [0, 1]$. If $num < \epsilon$, the player will explore a random action, else the player will exploit best action from the currently learnt $Q(s, a)$ table. Note that in this function, in order to assess the potential of Q-learning in detail, l is introduced for controlling the decay of ϵ. This parameter determines the value and changing speed of ϵ in current match count m. Instances in experiments are shown in Fig 2.1:

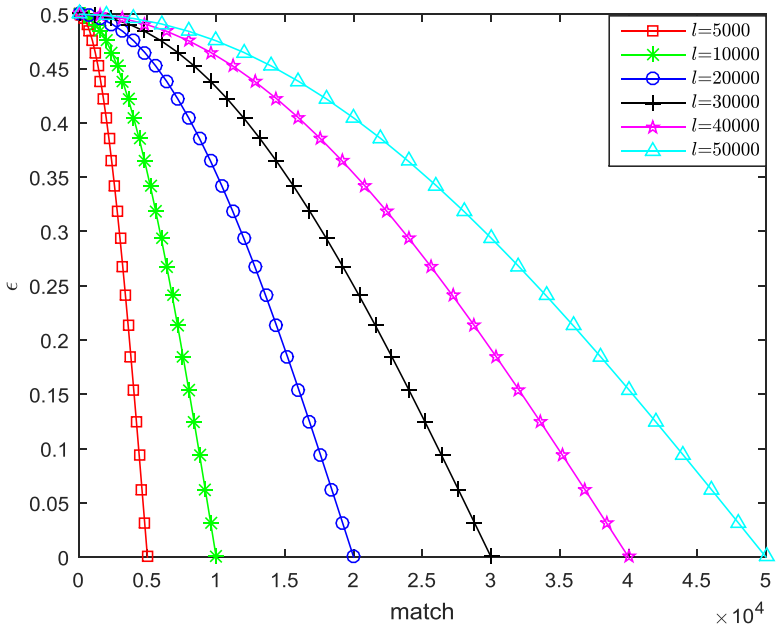


Figure 2.1: Decaying Curves of ϵ with Different l. Every curve decays from 0.5 (learning start, explore & exploit) to 0 ($m \geq l$, fully exploit).

2.3.3 QM-learning Enhancement

The main idea of MCS [47] is to make some lookahead probes from a non-terminal state to the end of the game by selecting random moves for the players to estimate the value of that state. To apply Monte Carlo in game playing, a time-limited version is used, since in competitive game playing time for each move is an important factor for the player to consider. The time limited MCS used in GGP is written as ***MonteCarloSearch(time_limit)***.

In Algorithm 1 (line 13), we see that a *random action* is chosen when QPlayer can not find an existing value in the $Q(s, a)$ table. In this case, QPlayer acts like a random player, which will lead to a low win rate and slow learning speed. In order to address this problem, a variant of Q-learning combined with MCS is introduced. MCS performs a time limited lookahead to find better moves. The more time it has, the better the action it finds will be. To achieve this, ***selected_action = MonteCarloSearch(time_limit)*** is used to replace the line 13, giving QM-learning. By adding MCS, a local version of the last two stages of MCTS is effectively added to Q-learning: the playout and backup stage [7].

2.4 Experiments and Results

2.4.1 Dynamic ϵ Enhancement

In experiments, the ϵ-greedy Q-learning players ($\alpha = 0.1$, $\gamma = 0.9$) with fixed ϵ=0.1, 0.2 and with dynamically decreasing $\epsilon \in [0, 0.5]$ is created to play 30000 matches first (l=30000) against a Random player, respectively. During these 30000 matches, the dynamic ϵ decreases from 0.5 to 0 based on the decay function, see equation 2.3. The fixed values for ϵ are 0.1 and 0.2, respectively. After 30000 matches, fixed ϵ is also set to 0 to continue the competition. For Tic-Tac-Toe, results in Fig. 2.2 show that dynamically decreasing ϵ performs better. We see that the final win rate of dynamically decreasing ϵ is 4% higher than fixed ϵ=0.1 and 7% higher than fixed ϵ=0.2. Therefore, in the rest of the experiments, dynamic ϵ is used for further improvements.

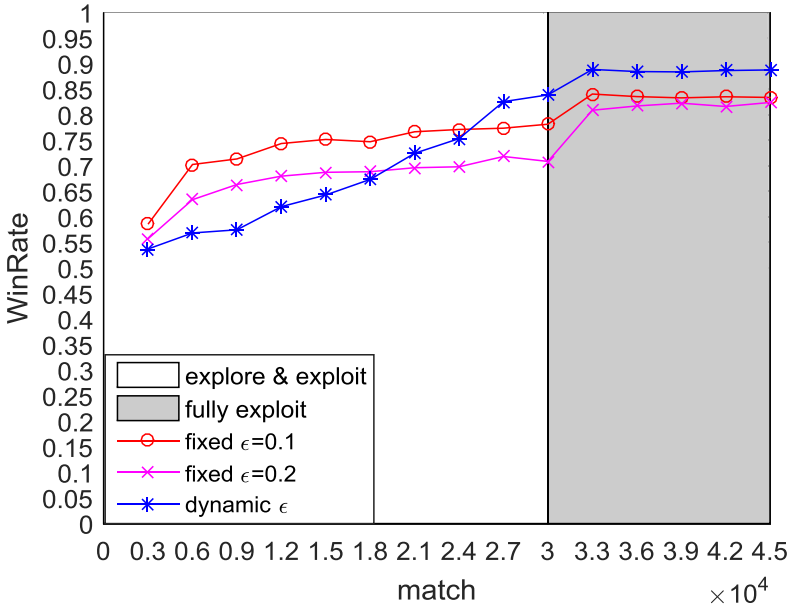


Figure 2.2: Win Rate of the Fixed and Dynamic ϵ Q-learning Player vs a Random Player Baseline. In the white part, the player uses ϵ-greedy to learn; in the grey part, all players set ϵ=0 (stable performance). The color code of the rest figures are the same

To enable comparison with previous work, TD(λ) is also implemented, the baseline learner of [46]($\alpha = 0.3$, $\gamma = 1.0$, $\lambda = 0.7$, $\epsilon = 0.01$), and dynamic ϵ

learner($\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$, l=30000, Algorithm 1). For Tic-Tac-Toe, from Fig. 2.3, It is found that although the TD(λ) player converges more quickly initially (win rate stays at about 75.5% after 9000th match) the dynamic ϵ player performs better when the value of ϵ decreases dynamically with the learning process.

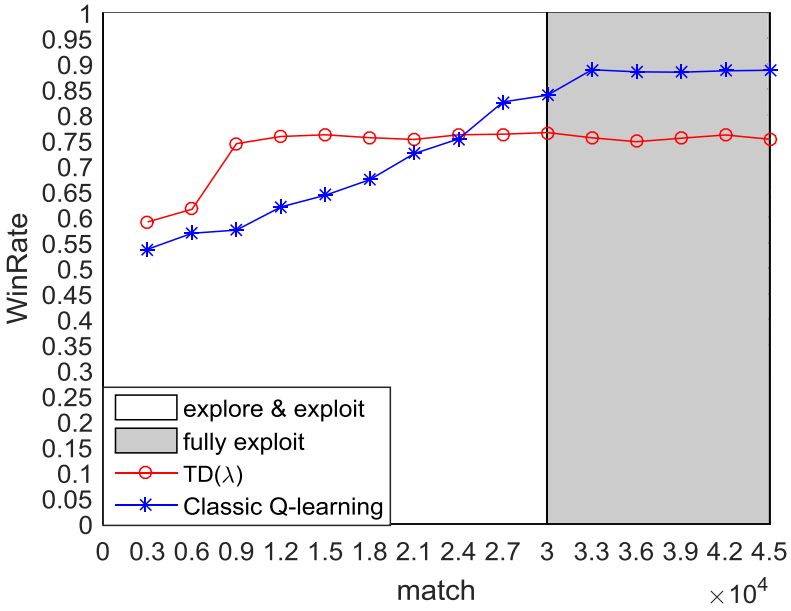
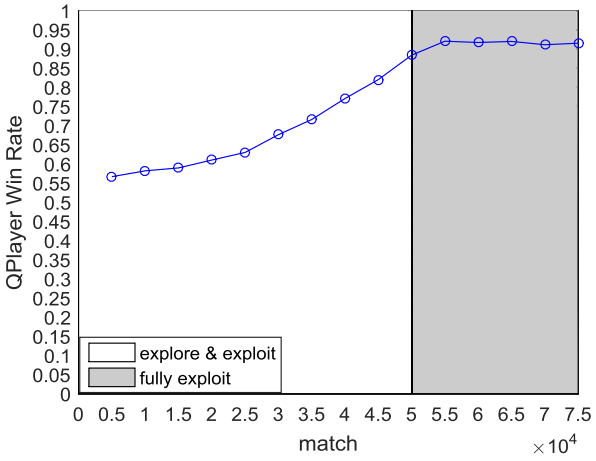
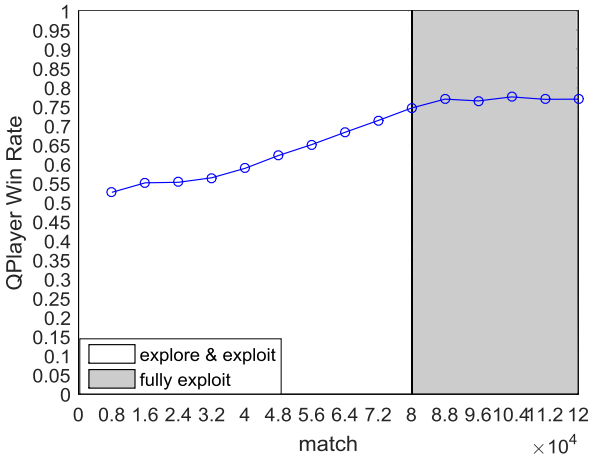


Figure 2.3: Win Rate of Classical Q-learning and [46] Baseline Player vs Random.

Experiments above suggest the following conclusions: that (1) classical Q-learning is applicable to a GGP system, and that (2) a dynamic ϵ can enhance the performance of fixed ϵ. However, beyond the basic applicability in a single game, showing that it can do so (1) *efficiently*, and (2) in more than one game is needed. Thus, further experiments with QPlayer to play Hex (l=50000) and Connect Four (l=80000) against the Random player are investigated. In order to limit excessive learning times, following [46], a very small 3×3 board Hex and ConnectFour on a 4×4 board are played. The results of these experiments are given in Fig. 2.4. We see that QPlayer can also play these other games effectively. Note that the reason why the player achieves different win rates could be that the game space of 3×3 Hex is much smaller than 4×4 ConnectFour.



(a) 3×3 Hex



(b) 4×4 Connect Four

Figure 2.4: Win Rate of QPlayer vs Random Player in Different Games. For Hex and Connect-Four the win rate of Q-learning also converges

However, so far, all studied games are small. QPlayer should be able to learn to play larger games. The complexity influences how many matches the QPlayer should learn. Now results will be shown to demonstrate how QPlayer performs while playing more complex games. QPlayer plays Tic-Tac-Toe (a line of 3 stones is a win, l=50000) in 3×3, 4×4 and 5×5 boards, respectively. and the results are shown in Fig. 2.5.

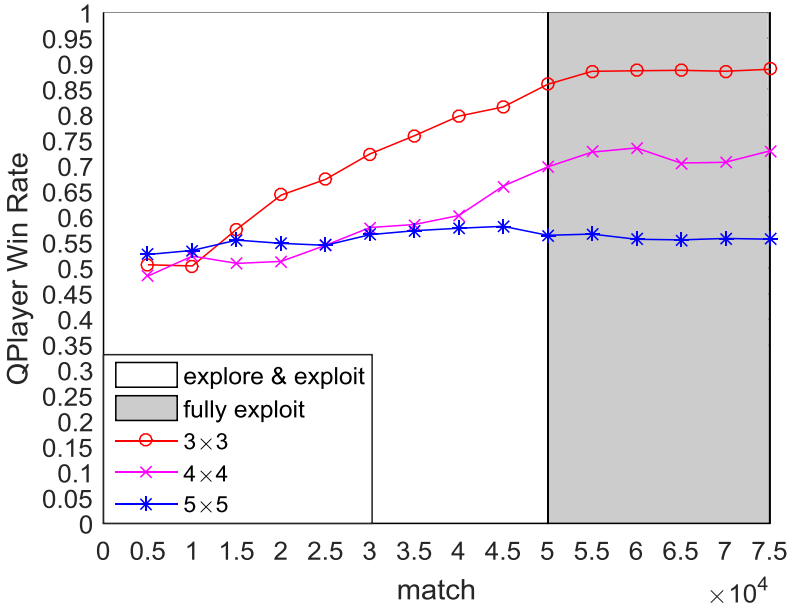
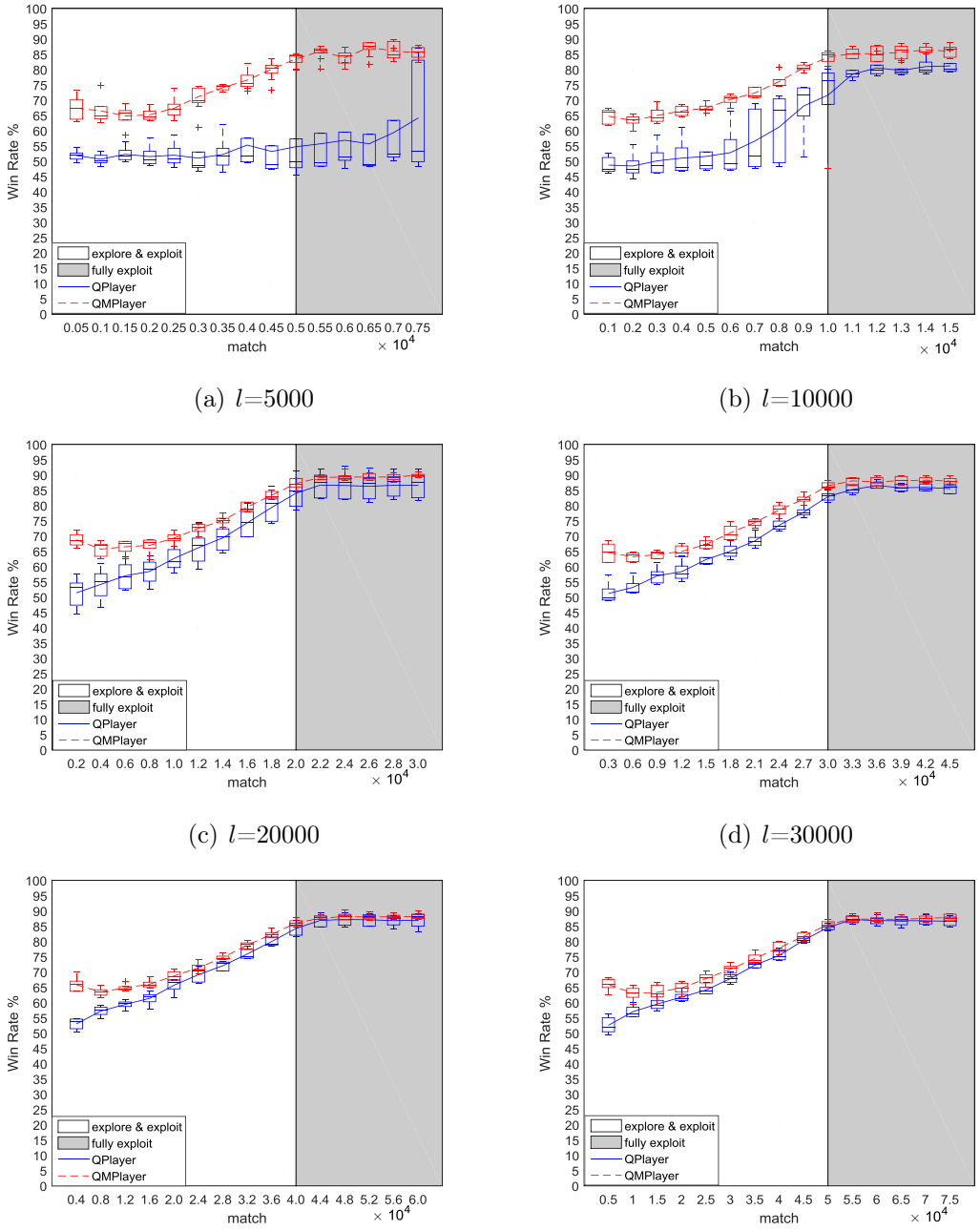


Figure 2.5: Win Rate of QPlayer vs Random in Tic-Tac-Toe on Different Board Size. For larger board sizes convergence slows down

The results show that with the increase of game board size, QPlayer performs worse. For larger boards can not achieve convergence. The reason for the lack of convergence is that QPlayer has not learned enough knowledge. The experiments also show that for table-based Q-learning in GGP, large game complexity leads to slow convergence, which confirms the well-known drawback of classical Q-learning.

2.4.2 QM-learning Enhancement

The second contribution of this chapter is QM-learning enhancement, the QPlayer and QMPlayer are both implemented based on Algorithm 1 and section 2.3.3. For both players, parameters are set to $\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$ respectively and the l=5000, 10000, 20000, 30000, 40000, 50000, respectively. For QMPlayer, $time\_limit = 50ms$. Next both players play the game with the Random baseline player for $1.5 \times l$ matches for 5 rounds respectively. The comparison between QPlayer and QMPlayer is shown in Fig. 2.6.



(a) l=5000

(b) l=10000

(c) l=20000

(d) l=30000

(e) l=40000

(f) l=50000

Figure 2.6: Win Rate of QMPlayer (QPlayer) vs Random in Tic-Tac-Toe for 5 experiments. Small Monte Carlo lookaheads improve the convergence of Q-learning, especially in the early part of learning. QMPlayer always outperforms Qplayer

Fig. 2.6(a) shows that QPlayer has the most unstable performance (the largest variance in 5 experiments) and only wins around 55% matches after training 5000 matches. Fig. 2.6(b) illustrates that after training 10000 matches QPlayer wins about 80% matches. However, during the exploration period (the white part of the figure) the performance is still very unstable. Fig. 2.6(c) shows that QPlayer wins about 86% of the matches while learning 20000 matches still with high variance. Fig. 2.6(d), Fig. 2.6(e), Fig. 2.6(f), show us that after training 30000, 40000, 50000 matches, QPlayer gets a similar win rate, which is nearly 86.5% with smaller and smaller variance.

In Fig. 2.6(a), QMPlayer gets a high win rate (about 67%) at the very beginning. Then the win rate decreases to 66% and 65%, and then increases from 65% to around 84% at the 5000th macth. Finally, the win rate stays at around 85%. Also in the other sub figures, for QMPlayer, the curves all decrease first and then increase until reaching a stable state. This is because at the very beginning, QMPlayer chooses more actions from MCS. Then as the learning period moves forward, it chooses more actions from Q table.

Overall, as the l increases, the win rate of QPlayer becomes higher until leveling off around 86.5%. The variance becomes smaller and smaller, which proves that Q-learning can achieve convergence in GGP games and that a proper ϵ decaying speed makes sense for classical Q-learning. Note that in every sub figure, QMPlayer can always achieve a higher win rate than QPlayer, not only at the beginning but also at the end of the learning period. Overall, QMPlayer achieves a better performance than QPlayer with the higher convergence win rate (at least 87.5% after training 50000 matches). To compare the convergence speeds of QPlayer and QMPlayer, the convergence win rates of different l is summarized according to Fig. 2.6 in Fig. 2.7.

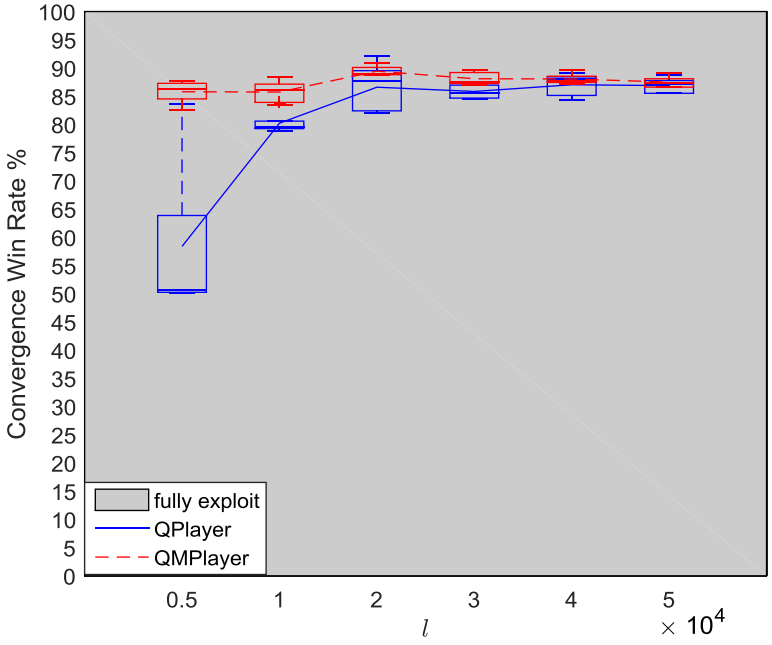


Figure 2.7: Convergence Win Rate of QMPlayer (QPlayer) vs Random in Tic-Tac-Toe

These results show that combining online MCS with classical Q-learning for GGP can improve the win rate both at the beginning and at the end of the offline learning period. The main reason is that QM-learning allows the $Q(s, a)$ table to be filled quickly with good actions from MCS, achieving a quick and direct learning rate. It is worth to note that, QMPlayer will spend slightly more time (at most is *search time limit*$\times$ *number of (state-action) pairs*) in training than QPlayer. It will be time consuming for MCS to compute a large game, and this is also the essential drawback of table-based Q-learning, so currently QM-learning is also only applicable for small games.

2.5 Summary

This chapter examines the applicability of Q-learning, a canonical reinforcement learning algorithm, to create general players for GGP programs. Firstly, it has shown how good canonical implementations of Q-learning perform on GGP games. The GGP system allows to easily use three real games for the experiments: Tic-Tac-Toe, Connect Four, and Hex. It is found that (1) Q-learning

is indeed general enough to achieve convergence in GGP games. However, convergence is slow. In accordance with Banerjee [46], who used a static value for ϵ, it is also found that (2) a value for ϵ that changes with the learning phases gives better performance (start with more exploration, become more greedy later on). The table-based implementation of Q-learning facilitates theoretical analysis, and comparison against some baselines [46]. However, it is only suitable for small games. A neural network implementation facilitates the study of larger games, and allows meaningful comparison to DQN variants [44].

Still using the table-based implementation, Q-learning is then enhanced with an MCS based lookahead. It is found that, especially at the start of the learning, this speeds up convergence considerably. The Q-learning is table-based, limiting it to small games. Even with the MCS enhancement, convergence of QM-learning does not yet allow its direct use in larger games. The QPlayer needs to learn a large number of matches to get good performance in playing larger games. The results with the improved Monte Carlo algorithm show a real improvement of the player's win rate, and learn the most probable strategies to get high rewards faster than learning completely from scratch. This enhancement shows how online search can be used to improve the performance of offline learning in GGP. On this basis, different offline learning algorithms can be assessed (or follow Gelly [18] to combine it with neural networks for larger games in GGP).

The use of Monte Carlo in QM-learning is different from the AlphaGo architecture, where MCTS is wrapped around Q-learning (DQN) [44]. In this work, Monte Carlo is inserted *within* the Q-learning loop. Future work should show if the QM-learning results transfer to AlphaGo-like uses of DQN inside MCTS, if QM-learning can achieve faster convergence, reducing the high computational demands of AlphaGo [16]. Additionally, nested MCS in Q-learning [57] could also be an option. Implementing Neural Network based players also allows the study of more complex GGP games [59].

Importantly, this work inspires a deep reinforcement learning framework to play GDL-games on GGP [60] and inspires the further studies of applying MCTS enhancements to improve neural network based reinforcement learning (Alphazero-like self-play) in Chapter 5 and Chapter 6.

Chapter 3

Hyper-Parameters for AlphaZero-like Self-play

3.1 Introduction

In order to further investigate AGI on games in a deep learning era, we can use a neural network based framework in GGP or switch to use an existing neural network based deep reinforcement learning system for games. We noticed that AlphaZero provides a successful general framework to play complex board games like Go, Chess and Shogi [10], and these AlphaGo series papers [10, 16, 17] have sparked much interest of researchers and the general public alike into deep reinforcement learning. Despite the success of AlphaGo and related methods in Go and other application areas [61, 62], there are unexplored and unsolved puzzles in the parameterization and design of the algorithms. For example, could MCTS enhancements also improve the performance in AlphaZero-like self-play as MCS enhancement did in table based Q-learning in GGP (Chapter 2), and how? (These questions will be further studied in Chapter 5 and Chapter 6.)

As for parameterization, different hyper-parameter settings can lead to very different results. However, hyper-parameter design-space sweeps are computationally very expensive, and in the original publications, we can only find limited information of how to set the values of some important parameters and why. Also, there are few works on how to set the hyper-parameters for these algorithms, and more insight into the hyper-parameter interactions is necessary. To this end, and there are some interesting re-implementations of AlphaZero [63, 64], we study the most

general framework algorithm in the aforementioned AlphaGo series by using a lightweight re-implementation of AlphaZero: AlphaZeroGeneral [65].

In order to optimize hyper-parameters, it is important to understand their function and interactions in an algorithm. A single iteration in the AlphaZeroGeneral framework consists of three stages: self-play, neural network training and arena comparison. In these stages, we explore 12 hyper-parameters (see section 3.4.1) in AlphaZeroGeneral. Furthermore, we observe 2 objectives (see section 3.4.2): training loss and time cost in each single run. A sweep of the hyper-parameter space is computationally demanding. In order to provide a meaningful analysis we use small board sizes of typical combinatorial games. This sweep provides an overview of the hyper-parameter contributions and provides a basis for further analysis. Based on these results, we choose 4 interesting parameters to further evaluate in depth.

As performance measure, we use the Elo rating that can be computed during training time of the self-play system, as a running relative Elo, and computed separately, in a dedicated tournament between different trained players.

Our contributions can be summarized as follows:

1. We find that in general higher values of most hyper-parameters lead to higher playing strength.
2. And within a limited budget, a higher number of outer iterations is more promising than higher numbers of inner iterations: these are subsumed by outer iterations.

This chapter is structured as follows. We first give an overview of the most relevant literature, before describing the considered test games in Sect. 3.3. Then we describe the AlphaZero-like self-play algorithm in Sect. 3.4. After setting up experiments in Sect. 3.5, we present the results in Sect. 3.6. Finally, we summarize the chapter and discuss promising future work.

3.2 Related work

Hyper-parameter tuning by optimization is very important for many practical algorithms. In reinforcement learning, for instance, the ϵ-greedy strategy of classical Q-learning is used to balance exploration and exploitation. Different ϵ values lead to different learning performance [31]. Another well known example of hyper-parameter tuning is the parameter C_p in MCTS [7, 66, 67]. There are many works on tuning C_p for different kinds of tasks. These provide insight on

setting its value for MCTS in order to balance exploration and exploitation [58]. In deep reinforcement learning, the effect of the many neural network parameters are a black-box that precludes understanding, although the strong decision accuracy of deep learning is undeniable [68], as the results in Go (and many other applications) have shown [69]. After AlphaGo [16], the role of self-play became more and more important. Earlier works on self-play in reinforcement learning are [70, 71, 72, 73]. An overview is provided in [74].

On hyper-parameters for AlphaZero-like systems there are a few studies: [75] tuned some parameters (in particular MCTS-related parameters in self-play game playing) in AlphaGo with Bayesian optimization, which leads to abandoning the fast rollout in AlphaGo Zero and AlphaZero.

Our experiments are also performed using AlphaZeroGeneral [65] on 6×6 Othello [76]. The smaller size of the game allows us to do more experiments, and these lead us into largely uncharted territory where we hope to find effects that cannot be seen in Go or Chess.

3.3 Test Game

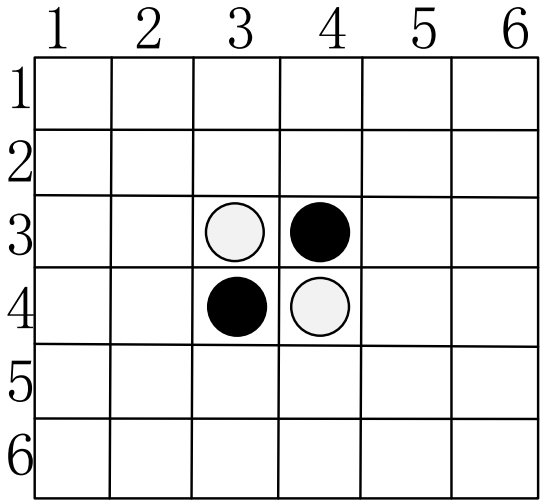


Figure 3.1: Starting position for 6×6 Othello

In our hyper-parameter sweep experiments, we use Othello with a 6×6 board size, see Fig. 3.1. Othello is a two-player game. Players take turns placing their own color pieces. Any opponent's color pieces that are in a straight line and bounded

by the piece just placed and another piece of the current player's are flipped to the current player's color. While the last legal position is filled, the player who has most pieces wins the game. Fig. 3.1 shows the start configurations for 6×6 Othello.

There is a wealth of research on finding playing strategies for these three games by means of different methods. For example, Buro created Logistello [77] to play Othello. Chong et al. described the evolution of neural networks for learning to play Othello [78]. Moreover, Banerjee et al. tested knowledge transfer in GGP on small games including 4×4 Othello [46]. The board size gives us a handle to reduce or increase the overall difficulty of these games. In our experiments we use AlphaZero-like zero learning, where a reinforcement learning system learns from tabula rasa, by playing games against itself using a combination of deep reinforcement learning and MCTS.

3.4 AlphaZero-like Self-play

3.4.1 The Base Algorithm

Following the works by Silver et al. [10, 16] the fundamental structure of AlphaZero-like Self-play is an iteration over three different stages (see Algorithm 2).

The first stage is a **self-play** tournament. The computer player performs several games against itself in order to generate data for further training. In each step of a game (episode), the player runs MCTS to obtain, for each move, an enhanced policy π based on the probability $\mathbf{p}$ provided by the neural network f_θ. We now introduce the hyper-parameters, and their abbreviation that we use in this thesis. In MCTS, hyper-parameter C_p is used to balance exploration and exploitation of game tree search, and we abbreviate it to c (the equation of P-UCT with c for MCTS can be found in Chapter 5). Hyper-parameter m is the number of times to run down from the root for building the game tree, where the parameterized network f_θ provides the value (v) of the states for MCTS. For actual (self-)play, from T' steps on, the player always chooses the best move according to π. Before that, the player always chooses a random move based on the probability distribution of π. After finishing the games, the new examples are normalized as a form of (s_t, π_t, z_t) and stored in D.

The second stage consists of **neural network training**, using data from the self-play tournament. Training lasts for several epochs. In each epoch (ep), training examples generated in the most recent rs iterations are stored in a retrain buffer

Algorithm 2 AlphaZero-like Self-play Algorithm

function AlphaZeroGeneral
  Initialize f_θ with random weights; Initialize retrain buffer D with capacity N
  for iteration=1, ..., I **do**
    for episode=1,..., E **do** ▷ stage 1
      for t=1, ..., $T\prime$, ..., T **do**
        Get the best move prediction π_t by performing MCTS based on $f_\theta(s_t)$
        if Before step $T\prime$ **then**
          select random action a_t based on probability π_t
        else
          select action $a_t = \arg\max_a(\pi_t)$
        Store example (s_t, π_t, z_t) in D
        Set s_t=excuteAction(s_t, a_t)
      Label reward z_t ($t \in [1, T]$) as z_T in examples
    Randomly sample minibatch of examples (s_j, π_j, z_j) from D ▷ stage 2
    $f_{\theta\prime} \leftarrow$ Train f_θ by minimizing Equation 3.1 based on sampled examples
    Set $f_\theta = f_{\theta\prime}$ if $f_{\theta\prime}$ is better than f_θ ▷ stage 3
  return f_θ;

and are divided into several small batches [79] according to the specific batch size (bs). The neural network is trained to minimize [80] the value of the *loss function* which (see Equation 3.1) sums up the mean-squared error between predicted outcome and real outcome and the cross-entropy losses between $\mathbf{p}$ and π with a learning rate (lr) and dropout (d). Dropout is used as probability to randomly ignore some nodes of the hidden layer in order to avoid overfitting [81].

The last stage is **arena comparison**, in which the newly trained neural network model ($f_{\theta\prime}$) is run against the previous neural network model (f_θ). The better model is adopted for the next iteration. In order to achieve this, $f_{\theta\prime}$ and f_θ play against each other for n games. If $f_{\theta\prime}$ wins more than a fraction of u games, it is replacing the previous best f_θ. Otherwise, $f_{\theta\prime}$ is rejected and f_θ is kept as current best model. Compared with AlphaGo Zero, AlphaZero does not entail the arena comparison stage anymore. However, we keep this stage for making sure that we can safely recognize improvements.

Furthermore, we present a conceptual diagram to describe the Algorithm 2 with necessary components for 3 stages and corresponding hyper-parameters in Fig. 3.2:

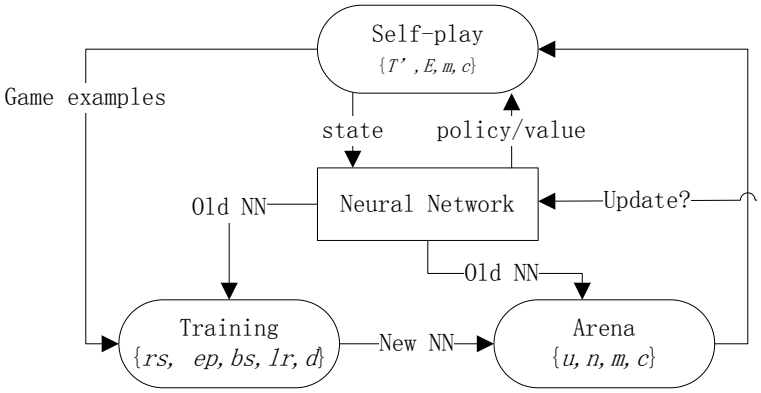


Figure 3.2: A diagram of the schema of AlphaZero-like Self-play Algorithm over 3 stages with corresponding hyper-parameters. Hyper-parameter I controls the loop over these 3 stages.

3.4.2 Loss Function

The **training loss function** consists of $l_{\mathbf{p}}$ and l_v. The neural network f_θ is parameterized by θ. f_θ takes the game board state s as input, and provides the value $v_\theta \in [-1, 1]$ of s and a policy probability distribution vector $\mathbf{p}$ over all legal actions as outputs. $\mathbf{p}_\theta$ is the policy provided by f_θ to guide MCTS for playing games. After performing MCTS, we obtain an improvement estimate as policy π. Training aims at making $\mathbf{p}$ more similar to π. This can be achieved by minimizing the cross entropy of both distributions. Therefore, $l_{\mathbf{p}}$ is defined as $-\pi^\top \log \mathbf{p}$. The other aim is to minimize the difference between the output value $(v_\theta(s_t))$ of the state s according to f_θ and the real outcome $(z_t \in \{-1, 1\})$ of the game. Therefore, l_v is defined as the mean squared error $(v - z)^2$. Summarizing, the total loss function of AlphaZero is defined in Equation 3.1.

$$l_+ = -\pi^\top \log \mathbf{p} + (v - z)^2 \tag{3.1}$$

Note that in AlphaZero's loss function, there is an extra regularization term to guarantee the training stability of the neural network. In order to pay more attention to two evaluation function components, instead, we apply standard measures to avoid overfitting such as the **dropout** mechanism.

3.4.3 Bayesian Elo System

The **Elo rating function** has been developed as a method for calculating the relative skill levels of players in games. Usually, in zero-sum games, there are

two players, A and B. If their Elo ratings are R_A and R_B, respectively, then the expectation that player A wins the next game is $E_A = \frac{1}{1+10^{(R_B-R_A)/400}}$. If the real outcome of the next game is S_A, then the updated Elo of player A can be calculated from its original Elo by $R_A = R_A + K(S_A - E_A)$, where K is the factor of the maximum possible adjustment per game. In practice, K should be bigger for weaker players but smaller for stronger players. Following [10], in our design, we adopt the Bayesian Elo system [82] to show the improvement curve of the learning player during self-play. We also employ this method to assess the playing strength of the final models.

3.4.4 Time Cost Function

Because of the high computational cost of self-play reinforcement learning, the running time of self-play is of great importance. We have created a **time cost function** to predict the running time, based on the algorithmic structure in Algorithm 2. According to Algorithm 2, the whole training process consists of several iterations with three steps as introduced in section 3.4.1. Please refer to the algorithm and to equation 3.2. In ith iteration ($1 \leq i \leq I$), if we assume that in jth episode ($1 \leq j \leq E$), for kth game step (the size of k mainly depends on the game complexity), the time cost of lth MCTS ($1 \leq l \leq m$) simulation is $t^{(i)}_{jkl}$, and assume that for pth epoch ($1 \leq p \leq ep$), the time cost of pulling qth batch ($1 \leq q \leq trainingExampleList.size/bs$)[1] through the neural network is $t^{(i)}_{pq}$, and assume that in wth arena comparison ($1 \leq w \leq n$), for xth game step, the time cost of yth MCTS simulation ($1 \leq y \leq m$) is $t^{(i)}_{xyw}$. The time cost of the whole training process is summarized in equation 3.2.

$$\sum_i (\overbrace{\sum_j \sum_k \sum_l t^{(i)}_{jkl}}^{self-play} + \overbrace{\sum_p \sum_q t^{(i)}_{pq}}^{training} + \overbrace{\sum_x \sum_y \sum_w t^{(i)}_{xyw}}^{arena\ comparison}) \tag{3.2}$$

Please refer to Table 3.1 for an overview of the hyper-parameters. From Algorithm 2 and equation 3.2, we can see that the hyper-parameters, such as I, E, m, ep, bs, rs, n etc., influence training time. In addition, $t^{(i)}_{jkl}$ and $t^{(i)}_{xyw}$ are simulation time costs that rely on hardware capacity and game complexity. $t^{(i)}_{uv}$ also relies on the structure of the neural network. In our experiments, all neural network models share the same structure, which consists of 4 convolutional layers and 2 fully connected layers.

[1] the size of *trainingExampleList* is also relative to the game complexity

3.5 Experimental Setup

We sweep the 12 hyper-parameters by configuring 3 different values (minimum value, default value and maximum value) to find the most promising parameter values. In each single run of training, we play 6×6 Othello [76] and change the value of one hyper-parameter, keeping the other hyper-parameters at default values (see Table 3.1).

Our experiments are run on a machine with 128GB RAM, 3TB local storage, 20-core Intel Xeon E5-2650v3 CPUs (2.30GHz, 40 threads), 2 NVIDIA Titanium GPUs (each with 12GB memory) and 6 NVIDIA GTX 980 Ti GPUs (each with 6GB memory). In order to keep using the same GPUs, we deploy each run of experiments on the NVIDIA GTX 980 Ti GPU. Each run of experiments takes 2 to 3 days.

3.5.1 Hyper-Parameter Sweep

In order to train a player to play 6×6 Othello based on Algorithm 2, we employ the parameter values in Table. 3.1. Each experiment only observes one hyper-parameter, keeping the other hyper-parameters at default values.

Table 3.1: Hyper-Parameter Setting

-	Description	Minimum	Default	Maximum
I	number of iteration	50	100	150
E	number of episode	10	50	100
T'	step threshold	10	15	20
m	MCTS simulation times	25	100	200
c	weight in UCT	0.5	1.0	2.0
rs	number of retrain iteration	1	20	40
ep	number of epoch	5	10	15
bs	batch size	32	64	96
lr	learning rate	0.001	0.005	0.01
d	dropout probability	0.2	0.3	0.4
n	number of comparison games	20	40	100
u	update threshold	0.5	0.6	0.7

3.5.2 Hyper-Parameters Correlation Evaluation

Based on the above experiments, we further explore the correlation of interesting hyper-parameters (i.e. I, E, m and ep) in terms of their best final player's playing strength and overall training time. We set values for these 4 hyper-parameters as Table 3.2, and other parameters values are set to the default values in Table. 3.1. In addition, for (and only for) this part of experiments, the stage 3 of Algorithm 2 is cut off. Instead, for every iteration, the trained model $f_{\theta'}$ is accepted as the current best model f_θ automatically, which is also adopted by AlphaZero and saves a lot of time.

Table 3.2: Correlation Evaluation Hyper-Parameter Setting

-	Description	Minimum	Middle	Maximum
I	number of iteration	25	50	75
E	number of episode	10	20	30
m	MCTS simulation times	25	50	75
ep	number of epoch	5	10	15

Note that due to computation resource limitations, for hyper-parameter sweep experiments on 6×6 Othello, we only perform single run experiments. This may cause noise, but still provides valuable insights on the importance of hyper-parameters under the AlphaZero-like self-play framework.

3.6 Experimental Results

In order to better understand the training process, first, we depict training loss evolution for default settings in Fig. 3.3.

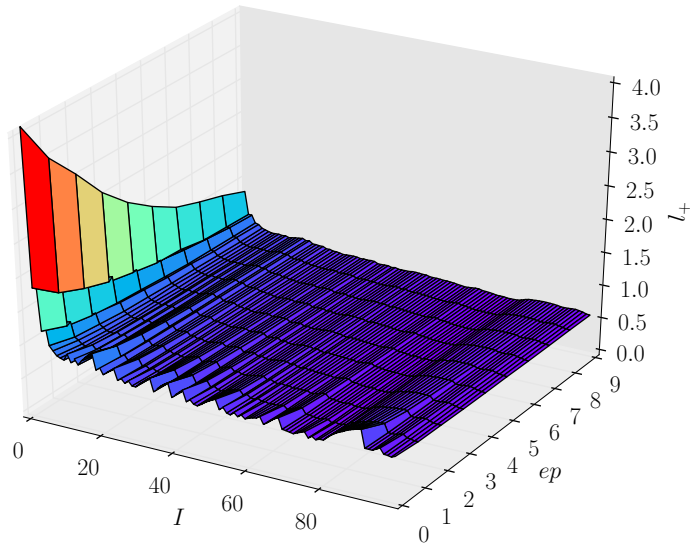


Figure 3.3: Single run training loss over iterations I and epochs ep

We plot the training loss of each epoch in every iteration and see that (1) in each iteration, loss decreases along with increasing epochs, and that (2) loss also decreases with increasing iterations up to a relatively stable level.

3.6.1 Hyper-Parameter Sweep Results

I: In order to find a good value for I (iterations), we train 3 different models to play 6×6 Othello by setting I at minimum, default and maximum value respectively. We keep the other hyper-parameters at their default values. Fig. 3.4(a) shows that training loss decreases to a relatively stable level. However, after iteration 120, the training loss unexpectedly increases to the same level as for iteration 100 and further decreases. This surprising behavior could be caused by a too high learning rate, an improper update threshold, or overfitting, or the learner suddenly explores something new.

E: Since more episodes mean more training examples, it can be expected that more training examples lead to more accurate results. However, collecting more training examples also needs more resources. This shows again that hyper-parameter optimization is necessary to find a reasonable value of for E. In Fig. 3.4(b), for E=100, the training loss curve is almost the same as the 2 other curves for a long time before eventually going down.

T': The step threshold controls when to choose a random action or the one suggested by MCTS. This parameter controls exploration in self-play, to prevent deterministic policies from generating training examples. Small T' results in more deterministic policies, large T' in policies more different from the model. In Fig. 3.4(c), we see that T'=10 is a good value.

m: In theory, more MCTS simulations m should provide better policies. However, higher m requires more time to get such a policy. Fig. 3.4(d) shows that a value for 200 MCTS simulations achieves the best performance in the 70th iteration, then has a drop, to reach a similar level as 100 simulations in iteration 100.

c: This hyper-parameter C_p is used to balance the exploration and exploitation during tree search. It is often set at 1.0. However, in Fig. 3.4(e), our experimental results show that more exploitation (c=0.5) can provide smaller training loss.

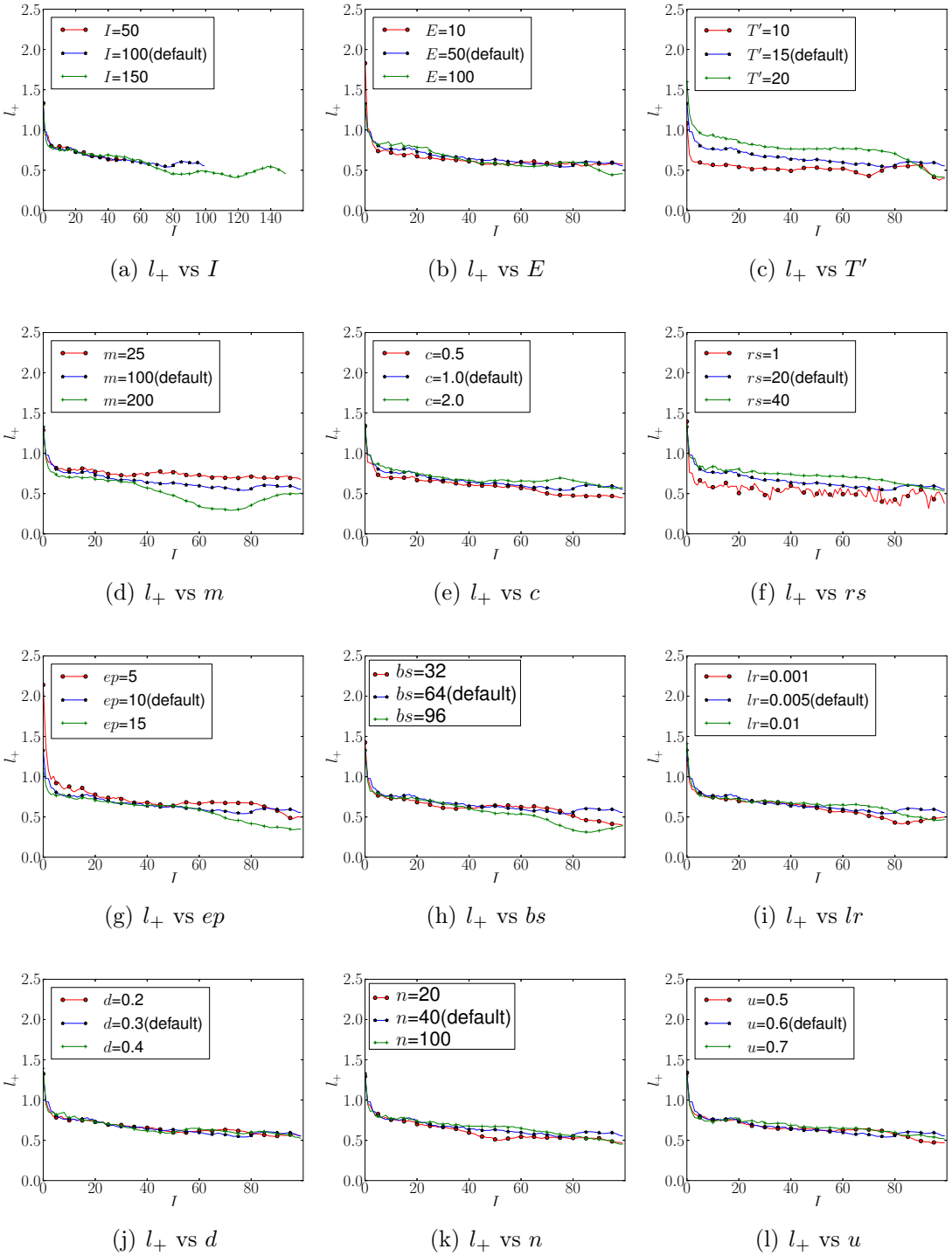
rs: In order to reduce overfitting, it is important to retrain models using previous training examples. Finding a good retrain length of historical training examples is necessary to reduce training time. In Fig. 3.4(f), we see that using training examples from the most recent single previous iteration achieves the smallest training loss. This is an unexpecrted result, suggesting that overfitting is prevented by other means and that the time saving works out best overall.

ep: The training loss of different ep is shown in Fig. 3.4(g). For ep=15 the training loss is the lowest. This result shows that along with the increase of epoch, the training loss decreases, which is as expected.

bs: a smaller batch size bs increases the number of batches, leading to higher time cost. However, smaller bs means less training examples in each batch, which may cause more fluctuation (larger variance) of training loss. Fig. 3.4(h) shows that bs=96 achieves the smallest training loss in iteration 85.

lr: In order to avoid skipping over optima, a small learning rate is generally suggested. However, a smaller learning rate learns (accepts) new knowledge slowly. In Fig. 3.4(i), lr=0.001 achieves the lowest training loss around iteration 80.

d: Dropout is a popular method to prevent overfitting. Srivastava et al. claim that dropping out 20% of the input units and 50% of the hidden units is often found to be good [81]. In Fig. 3.4(j), however, we can not see a significant difference.



(a) l_+ vs I (b) l_+ vs E (c) l_+ vs T'

(d) l_+ vs m (e) l_+ vs c (f) l_+ vs rs

(g) l_+ vs ep (h) l_+ vs bs (i) l_+ vs lr

(j) l_+ vs d (k) l_+ vs n (l) l_+ vs u

Figure 3.4: Training loss for different parameter settings over iterations. Larger figures in arXiv version can be found in [33].

n: The number of games in the arena comparison is a key factor of time cost. A small value may miss accepting good new models and too large a value is a waste of time. Our experimental results in Fig. 3.4(k) show that there is no significant difference. A combination with u can be used to determine the acceptance or rejection of a newly learnt model. In order to reduce time cost, a small n combined with a large u may be a good choice.

u: This hyper-parameter is the update threshold. Normally, in two-player games, player A is better than player B if it wins more than 50% games. A higher threshold avoids fluctuations. However, if we set it too high, it becomes too difficult to accept better models. Fig. 3.4(l) shows that u=0.7 is too high, 0.5 and 0.6 are acceptable.

Table 3.3: Time Cost (hr) of Different Parameter Setting

Parameter	Minimum	Default	Maximum	Type
I	**23.8**	44.0	60.3	time-sensitive
E	**17.4**	44.0	87.7	time-sensitive
T'	41.6	44.0	40.4	time-friendly
m	**26.0**	44.0	64.8	time-sensitive
c	50.7	44.0	49.1	time-friendly
rs	**26.5**	44.0	50.7	time-sensitive
ep	**43.4**	44.0	55.7	time-sensitive
bs	47.7	44.0	**37.7**	time-sensitive
lr	47.8	44.0	40.3	time-friendly
d	51.9	44.0	51.4	time-friendly
n	**33.5**	44.0	57.4	time-sensitive
u	39.7	44.0	40.4	time-friendly

To investigate the impact on running time, we present the effect of different values for each hyper-parameter in Table 3.3. We see that for parameter *I*, *E*, *m*, *rs*, *n*, smaller values lead to quicker training, which is as expected. For *bs*, larger values result in quicker training. The other hyper-parameters are indifferent, changing their values will not lead to significant changes in training time. Therefore, tuning these hyper-parameters shall reduce training time or achieve better quality in the same time.

Based on the aforementioned results and analysis, we summarize the importance by evaluating the contribution of each parameter to training loss and time cost, respectively, in Table 3.4 (best values in bold font). For training loss, different values of n and u do not result in a significant difference. Modifying time-indifferent hyper-parameters changes training time only slightly , whereas larger value of time-sensitive hyper-parameters lead to higher time cost.

Table 3.4: A Summary of Importance in Different Objectives

Parameter	Default Value	Loss	Time Cost
I	100	100	**50**
E	50	**10**	**10**
T'	15	**10**	similar
m	100	**200**	**25**
c	1.0	**0.5**	similar
rs	20	**1**	**1**
ep	10	**15**	**5**
bs	64	**96**	**96**
lr	0.005	**0.001**	similar
d	0.3	0.3	similar
n	40	insignificant	**20**
u	0.6	insignificant	similar

3.6.2 Hyper-Parameter Correlation Evaluation Results

In this part, we investigate the correlation between promising hyper-parameters in terms of time cost and playing strength. There are $3^4 = 81$ final best players trained based on 3 different values of 4 hyper-parameters (I, E, m and ep) plus a random player (i.e. 82 in total). Any 2 of these 82 players play with each other. Therefore, there are $82\times81/2$=3321 pairs, and for each of these, 10 games are played.

In each sub-figures of Fig. 3.5, all models are trained from the same value of I and E, according to the different values in x-axis and y-axis, we find that, generally, larger m and larger ep lead to higher Elo ratings. However, in the last sub-figure, we can clearly notice that the Elo rating of ep=10 is higher than that of ep=15 for m=75, which shows that sometimes more training can not improve the

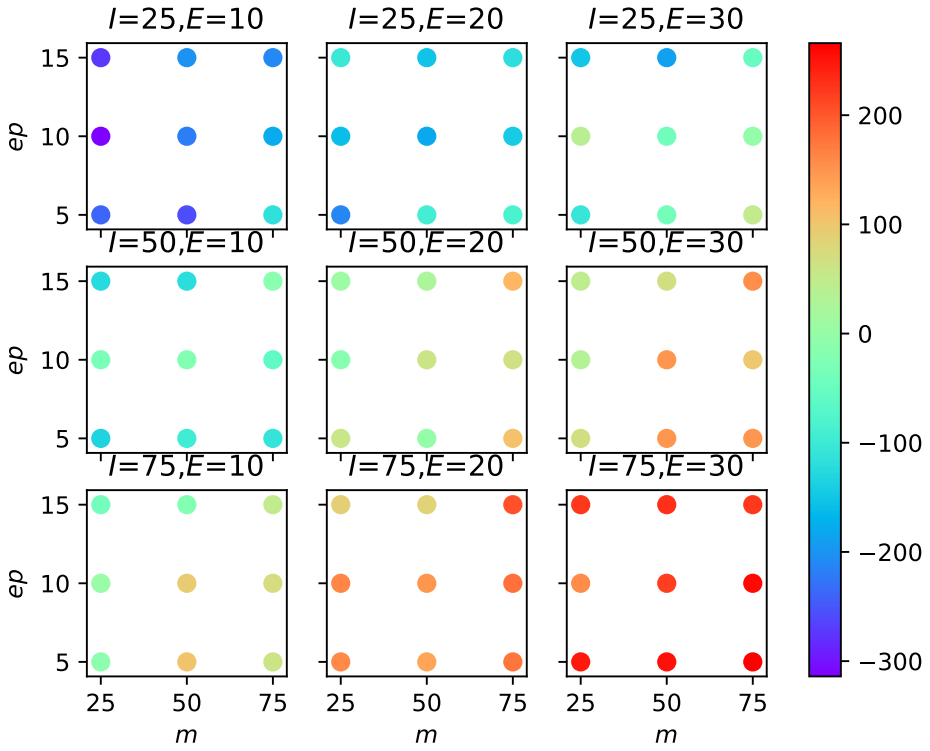


Figure 3.5: Elo ratings of the final best players of the full tournament (3 parameters, 1 target value)

playing strength but decreases the training performance. We suspect that this is caused by overfitting. Looking at the sub-figures, the results also show that more (outer) training iterations can significantly improve the playing strength, also more training examples in each iteration (bigger E) helps. These outer iterations are clearly more important than optimizing the inner hyper-parameters of m and ep. Note that higher values for the outer hyper-parameters imply more MCTS simulations and more training epochs, but not vice versa (more MCTS simulations and more training epochs could also come from more inner MCTS simulation and training epochs). This is an important insight regarding tuning hyper-parameters for self-play.

According to (3.2) and Table. 3.4, we know that smaller values of time-sensitive hyper-parameters result in quicker training. However, some time-sensitive hyper-parameters influence the training of better models. Therefore, we analyze training time versus Elo rating of the hyper-parameters, to achieve the best training

performance for a fixed time budget.

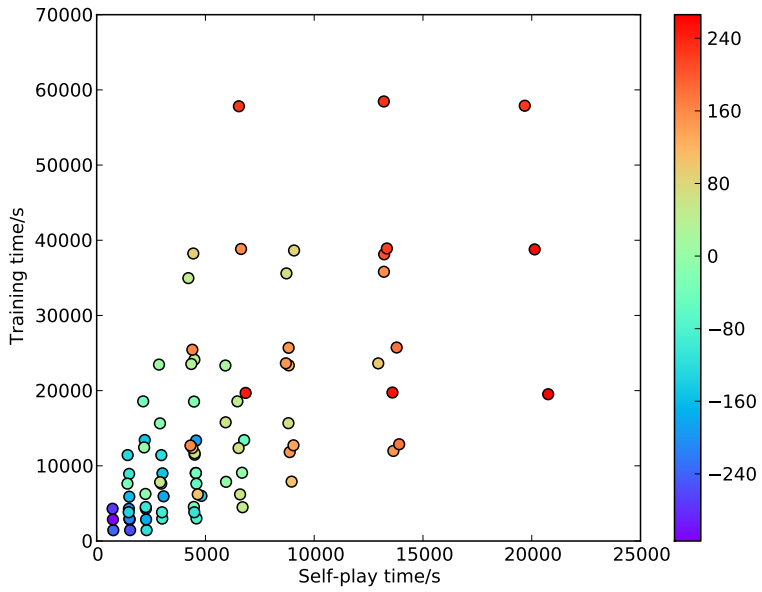


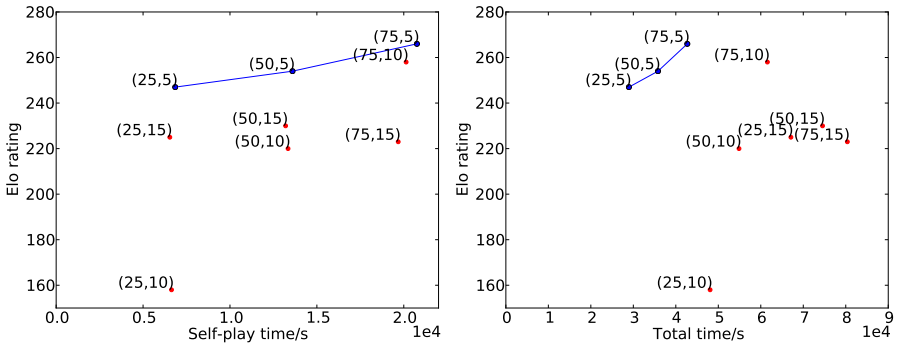
Figure 3.6: Elo ratings of the final best players with different time cost of Self-play and neural network training (same base data as in Fig. 3.5)

In order to find a way to assess the relationship between time cost and Elo ratings, we categorize the time cost into two parts, one part is the self-play (stage 1 in Algorithm 2, iterations and episodes) time cost, the other is the training part (stage 2 in Algorithm 2, training epochs). In general, spending more time in training and in self-play gives higher Elo. In self-play time cost, there is also an other interesting variable, searching time cost, which is influenced by the value of m.

In Fig. 3.6 we also find high Elo points closer to the origin, confirming that high Elo combinations of low self-play time and low training time exist, as was indicated above, by choosing low epoch ep and simulation m values, since the outer iterations already imply adequate training and simulation.

In order to further analyze the influence of self-play and training on time, we present in Fig. 3.7(a) the full-tournament Elo ratings of the lower right panel in Fig. 3.5. The blue line indicates the Pareto front of these combinations. We find that low epoch values achieves the highest Elo in a high iteration training session: more outer self-play iterations implies more training epochs, and the data

generated is more diverse such that training reaches more efficient stable state (no overfitting). All the points below the Pareto lines are sub optimal: it does not make much sense to use them. For all the Pareto points, it is a question of what is more important, time or strength.



(a) Self-play time vs Elo

(b) Total time vs Elo

Figure 3.7: Elo ratings of final best players to self-play, training and total time cost while I=75 and E=30.The values of tuple (m, ep) are given in the figures for every data point. In long total training, for m, larger values cost more time and generally improve the playing strength. For ep, more training within one iteration does not show improvement for Elo ratings. The lines indicate the Pareto fronts of Elo rating vs. time.

3.7 Summary

AlphaGo has taken reinforcement learning by storm. The performance of the novel approach to self-play is stunning, yet the computational demands are high, prohibiting the wider applicability of this method. Little is known about the impact of the values of the many hyper-parameters on the speed and quality of learning. In this work, we analyze important hyper-parameters and combinations of loss-functions. We gain more insight and find recommendations for faster and better self-play. We have used small games to allow us to perform a thorough sweep using a large number of hyper-parameters, within a reasonable computational budget. We sweep 12 parameters in AlphaZeroGeneral [65] and analyse loss and time cost for 6×6 Othello, and select the 4 most promising parameters for further optimization.

We more thoroughly evaluate the interaction between these 4 time-related hyper-parameters, and find that i) generally, higher values lead to higher playing strength; ii) within a limited budget, a higher number of the outer self-play iterations is more promising than higher numbers of the inner training epochs, search simulations, and game episodes. At first this is a surprising result, since conventional wisdom tells us that deep learning networks should be trained well, and MCTS needs many play-out simulations to find good training targets.

In AlphaZero-like self-play, the outer-iterations subsume the inner training and search. Performing more outer iterations automatically implies that more inner training and search is performed. The training and search improvements carry over from one self-play iteration to the next, and long self-play sessions with many iterations can get by with surprisingly little inner training epochs and MCTS simulations. The sample-efficiency of self-play is higher than simple composition of the constituent elements would predict. Also, the implied high number of training epochs may cause overfitting, to be reduced by small values for epochs.

In our experiments we also noted that care must be taken in computing Elo ratings. Computing Elo based on game-play results during training typically gives biased results that differ greatly from tournaments between multiple opponents. Final best models tournament Elo calculation should be used.

For future work, more insight into training bias is needed. Also, automatic optimization frameworks can be explored, such as [83, 84]. Also, reproducibility studies should be performed to see how our results carry over to larger games (like Go), computational load permitting. Given that [75] tuned some MCTS-related parameters (like exploration and exploitation balancing which we also adopt as parameter c) in AlphaGo with Bayesian optimization, resulting in Elo improvements, which evidenced our findings in self-play. However, [75] did not directly study the parameters in neural network training, we believe our work provides insightful analysis for future work on larger games.

Chapter 4

Loss Functions of AlphaZero-like Self-play

4.1 Introduction

As we introduced in Chapter 3, the AlphaGo series of papers [10, 16, 17] have sparked enormous interest of researchers and the general public alike into deep reinforcement learning [85, 86, 87, 88]. AlphaGo Zero [17], the successor of AlphaGo, masters the game of Go even without human knowledge. It generates game playing data purely by an elegant form of self-play, training a single unified neural network with a policy head and a value head, in an MCTS searcher. AlphaZero [10] uses a single architecture for playing three different games (Go, Chess and Shogi) without human knowledge. Many applications and optimization methods [89, 90] have been published and transformed the research field into one of the most active of current computer science.

Despite the success of AlphaGo and related methods in various application areas, there are unexplored and unsolved puzzles in the design and parameterization of the algorithms. We have demonstrated the hyper-parameter tuning results of a light-weight AlphaZero-like self-play framework in Chapter 3. Therefore, in this Chapter, we focus on the loss function design of the AlphaZero-like self-play.

The neural network in AlphaZero is represented as $f_\theta = (\mathbf{p}, v)$ (a unified deep network with a policy head and a value head). A policy $\mathbf{p}$ is a probability distribution for choosing the best move. A lower policy loss ($l_\mathbf{p}$) indicates a more accurate selection of the best move. A value function v is the prediction of the

final outcome. A lower value loss (l_v) indicates a more accurate prediction of the final outcome. The use of a double-headed network by Alpha(Go) Zero is innovative, and we know of no in-depth study of how the two losses ($l_{\mathbf{p}}$ and l_v) contribute to the playing strength of the final player. In Alpha(Go) Zero the sum of the two losses is used. Other studies based on the AlphaGo series algorithms just use it that way. However, the finding in the work of Matsuzaki et al. [91] is different, which reminds us to carefully study alternative evaluation functions. Thus, In order to increase our understanding of the inner workings of the minimization of the double-headed network we study different combinations of policy and value loss in this chapter. Therefore, in this work, we investigate:

a) what will happen if we only minimize a single target?

b) is a product combination a good alternative to summation?

We perform our experiments using a light-weight AlphaZero implementation named AlphaZeroGeneral [65] and focus on smaller games, namely 5×5 and 6×6 Othello [76], 5×5, 6×6 Connect Four games [92] and 5×5 and 6×6 Gobang [93].

As performance measure we use the Elo rating that can be computed during training time of the self-play system, as a running relative Elo. It can also be computed separately, in a dedicated tournament between different trained players. Our contributions can be summarized as follows:

- Experimental results show that there is a high self-play bias in computing training Elo ratings, such that it is incomparable among different training runs. A full tournament is necessary to compare final best players' Elo ratings and accurately measure the playing strength of different players relative to each other.
- We evaluate 4 alternative loss functions for 3 games and 2 board sizes, and find that the best setting depends on the game and is usually not the sum of policy and value loss. However, the sum might be considered as a good default compromise if no further information about the game is present.

The chapter is structured as follows. Part 4.2 presents related work. Part 4.3 presents games tested in the experiments. Part 4.4 introduces the default loss function of AlphaZero-like self-play and alternative loss functions. Part 4.5 sets up the experiments. Part 4.6 presents the experimental results. Part 4.7 discusses future work and summarizes this chapter.

4.2 Related Work

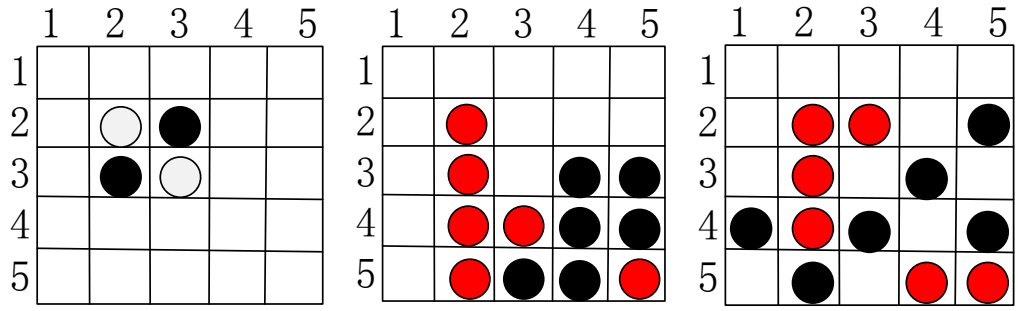
Deep reinforcement learning [94] is currently one of the most active research areas in AI, reaching human level performance for difficult games such as Go [69], which was almost unthinkable 10 years ago. Since Mnih et al. reported human-level control for playing Atari 2600 games by means of deep reinforcement learning [44], the performance of deep Q-networks (DQN) improved dramatically.

We have also observed a shift in DQN from imitating and learning from expert human players [16] to relying more on self-play. This has been advocated in the area of reinforcement learning [72, 73] for quite some time already. Silver et al. [17] turned to self-play to generate training data instead of training from human data (AlphaGo Zero), which not only saves a lot of work of collecting and labeling data from human experts, but also shifts the constraining factor for learning from available data to computing power, and achieves a form of efficient curriculum learning [95]. This approach was generalized to a framework (AlphaZero), showing the same approach that worked in Go, also worked in Shogi and Chess, demonstrating how to transfer the learning process [10].

Reinforcement learning is a very active field. We see a move away from human data to self-play. After many years of active research in MCTS [7], currently most research effort is in improving DQN variants. AlphaGo is a complex system with many tunable hyper-parameters. It is unclear if the many choices concerning parameters and methods that have been made in the AlphaGo series are close to optimal or if they can be improved by, e.g., changing parameters [32]. This includes the choice of minimization tasks (loss functions) used for measuring training success. For instance, [96] studied policy and value network optimization as a multi-task learning problem [97]. Even if the choices were very good for Go and other complex games, this does not necessarily transfer well to less complex tasks. For example, AlphaGo's PUCT achieves better results than a single evaluation function, but the result in [91] is different while playing Othello. Moreover, [98] showed that the value function has more importance than the policy function in the P-UCT algorithm for Othello.

4.3 Test Games

In our experiments, we use the games Othello, Connect Four and Gobang, each with 5×5 and 6×6 board sizes. As described in Chapter 3, Othello is a popular two-player game. Players take turns placing their own color pieces. Any opponent's color pieces that are in a straight line and bounded by the piece just



(a) 5×5 Othello (b) 5×5 Connect Four (c) 5×5 Gobang

Figure 4.1: Our test games on 5×5 boards

placed and another piece of the current player's are flipped to the current player's color. After the last legal position is filled, the player who has most pieces wins the game. Fig. 4.1(a) is the start configuration for 5×5 Othello. Connect Four is a two-player connection game. Players take turns dropping their own pieces from the top into a vertically suspended grid. The pieces fall straight down and occupy the lowest position within the column. The player who first forms a horizontal, vertical, or diagonal line of four pieces wins the game. Fig. 4.1(b) is a game termination example for 5×5 Connect Four where the red player wins the game. Gobang is another connection game that is traditionally played with Go pieces (black and white stones) on a Go board. Players alternate turns, placing a stone of their color on an empty position. The winner is the first player to form an unbroken chain of 4 stones horizontally, vertically, or diagonally. Fig. 4.1(c) is a game termination example for 5×5 Gobang where the black player wins the game.

There is a wealth of research on finding playing strategies for these three games by means of different methods. For example, Buro created Logistello [77] to play Othello. Chong et al. described the evolution of neural networks for learning to play Othello [78]. Thill et al. applied temporal difference learning to play Connect Four [99]. Zhang et al. designed evaluation functions for Gobang [100]. Moreover, Banerjee et al. tested knowledge transfer in GGP on small games including 4×4 Othello [46]. Wang et al. assessed the potential of classical Q-learning based on small games including 4×4 Connect Four [31]. Obviously, these two games are commonly tested in game playing.

4.4 Loss Function

4.4.1 Minimization Targets

As we want to assess the effect of minimizing different loss functions for AlphaZero-like self-play (Algorithm 2), besides the default loss function (Equation: 3.1), we employ a weighted sum loss function based on Equation 3.1:

$$l_\lambda = \lambda(-\pi^\top \log \mathbf{p}) + (1-\lambda)(v-z)^2 \tag{4.1}$$

where λ is a weight parameter. This provides some flexibility to gradually change the nature of the function. In our experiments, we first set λ=0 and λ=1 in order to assess $l_{\mathbf{p}}$ or l_v independently. Then we use Equation 3.1 as training loss function. Furthermore, inspired by that, in the theory of multi-attribute utility functions in multi-criteria optimization [101], a sum tends to prefer extreme solutions, whereas product prefers more balanced solution in case of coefficient objectives. Thus a product combination loss function is employed as follows:

$$l_\times = -\pi^\top \log \mathbf{p} \times (v-z)^2 \tag{4.2}$$

For all experiments, each setting is run 8 times to get statistically significant results (with error bars) using the parameters of Table 4.1 as default values. However, in order to save training time, we reduce the iteration number to 100 in the larger games (6×6 Othello and 6×6 Connect Four).

4.5 Experimental Setup

Our experiments are performed on a GPU server with 128G RAM, 3TB local storage, 20 Intel Xeon E5-2650v3 CPUs (2.30GHz, 40 threads), 2 NVIDIA Titanium GPUs (each with 12GB memory) and 6 NVIDIA GTX 980 Ti GPUs (each with 6GB memory). On these GPUs, every algorithm training run takes 2∼3 days. In this work, all neural network models share the same structure as used in Chapter 3, which consists of 4 convolutional layers and 2 fully connected layers [65]. The parameter values for Algorithm 2 used in our experiments are given in Table 4.1. In order to enhance reproducibility, we used values based on work reported by [32].

Table 4.1: Default Parameter Settings

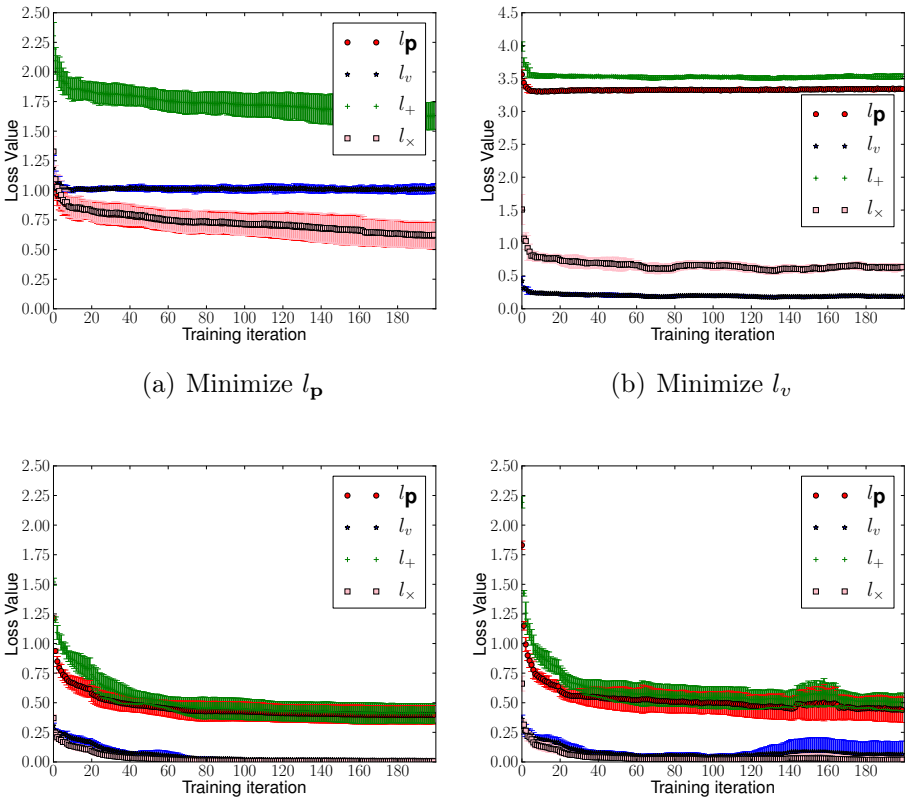
Parameter	Brief Description	Default Value
I	number of iteration	200
E	number of episode	50
T'	step threshold	15
m	MCTS simulation times	100
c	weight in UCT	1.0
rs	number of retrain iteration	20
ep	number of epoch	10
bs	batch size	64
lr	learning rate	0.005
d	dropout probability	0.3
n	number of comparison games	40
u	update threshold	0.6

4.5.1 Measurements

The chosen loss function is used to guide each training process, with the expectation that smaller loss means a stronger model. However, in practise, we have found that this is not always the case and another measure is needed to check based on trained models real playing performance in competitions. Therefore, following Deep Mind's work, we employ Bayesian Elo ratings [82] to describe the playing strength of the model in every iteration. In addition, for each game, we use all best players trained from the four different targets ($l_{\mathbf{p}}$, l_v, l_+, $l_\times$) and 8 repetitions plus a random player to play the game with each other for 20 times. From this, we calculate the Elo ratings of these 33 players to show the real playing strength of a player, rather than the playing strength only based on its own self-play training.

4.6 Experiment Results

In the following, we present the results of different loss functions. We have measured the individual value loss, the individual policy loss, the sum of the two, and the product of the two, for the three games. We report training loss, the training Elo rating and the tournament Elo rating of the final best players. Error bars indicate standard deviations of 8 runs.



(a) Minimize $l_{\mathbf{p}}$

(b) Minimize l_v

(c) Minimize l_+

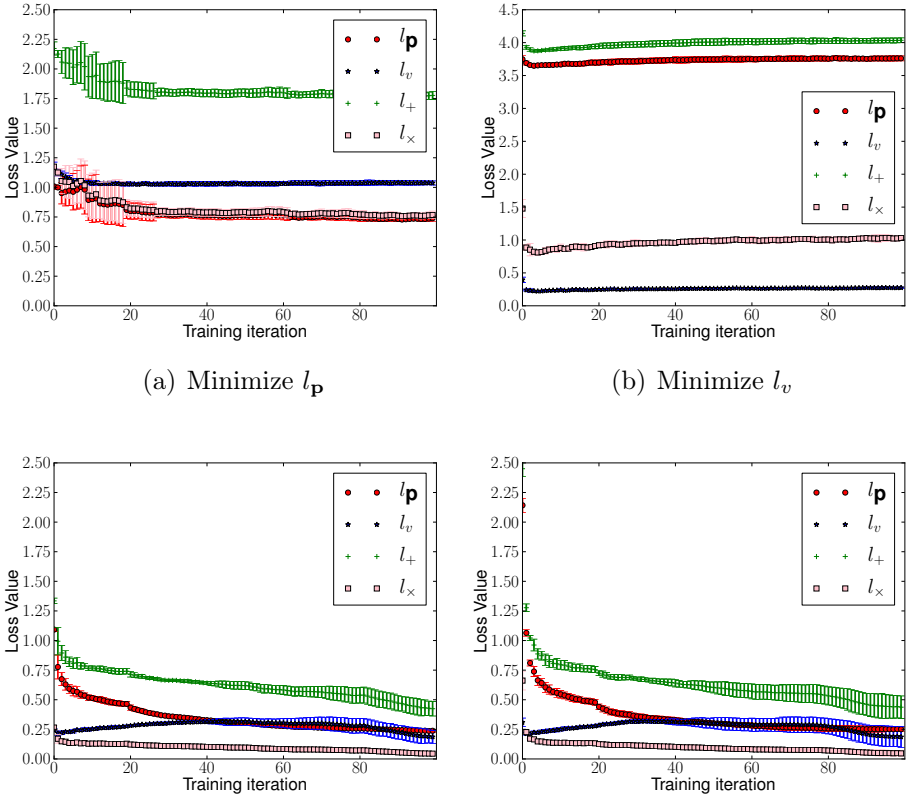
(d) Minimize $l_\times$

Figure 4.2: Training losses for minimizing different targets in 5×5 Othello, averaged over 8 runs. All measured losses are shown, but only one of these is minimized for. Note the different scaling for subfigure (b). Except for the l_+, the target that is minimized for is also the lowest

4.6.1 Training Loss

We first show the training losses in every iteration with one minimization task per diagram, hence we need four of these per game. In these graphs we see what minimizing for a specific target actually means for the other loss types.

For 5×5 Othello, from Fig. 4.2(a), we find that when minimizing $l_{\mathbf{p}}$ only, the loss decreases significantly to about 0.6 at the end of each training, whereas l_v stagnates at 1.0 after 10 iterations. Minimizing only l_v (Fig. 4.2(b)) brings it down from 0.5 to 0.2, but $l_{\mathbf{p}}$ remains stable at a high level. In Fig. 4.2(c), we see that when the l_+ is minimized, both losses are reduced significantly. The $l_{\mathbf{p}}$



(a) Minimize $l_{\mathbf{p}}$

(b) Minimize l_v

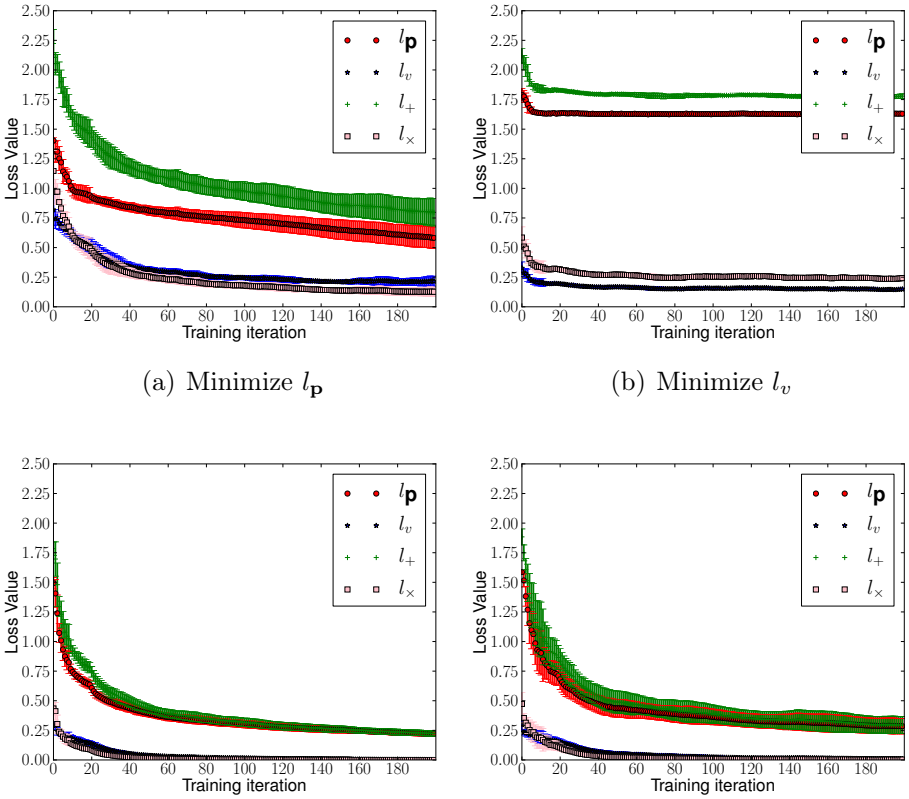
(c) Minimize l_+

(d) Minimize $l_\times$

Figure 4.3: Training losses for minimizing different targets in 6×6 Othello, averaged from 8 runs. All losses are shown while we minimize only one (similar to Fig 4.2). Note the different scaling for subfigure (b). Except for l_+, the target that is minimized for is the lowest

decreases from about 1.2 to 0.5, l_v surprisingly decreases to 0. Fig. 4.2(d), it is similar to Fig. 4.2(c), while the $l_\times$ is minimized, the $l_{\mathbf{p}}$ and l_v are also reduced. The $l_{\mathbf{p}}$ decreases to 0.5, the l_v also surprisingly decreases to about 0.

For the larger 6×6 Othello, we find that minimizing only $l_{\mathbf{p}}$ reduces it significantly to about 0.75, where l_v is stable again after about 10 iterations (Fig. 4.3(a)). For minimizing l_v (Fig. 4.3(b)), the results show that l_v is reduced from more than 0.5 to about 0.25 at the end of each training, but $l_{\mathbf{p}}$ seems to remain almost unchanged. For minimizing the l_+ (Fig. 4.3(c)), we find in contrast to 5×5 Othello that $l_{\mathbf{p}}$ decreases from about 1.1 to 0.4, whereas l_v increases slightly from about 0.2 and then decreases to about 0.2 again. We also find a similar behavior of l_v



(a) Minimize $l_{\mathbf{p}}$

(b) Minimize l_v

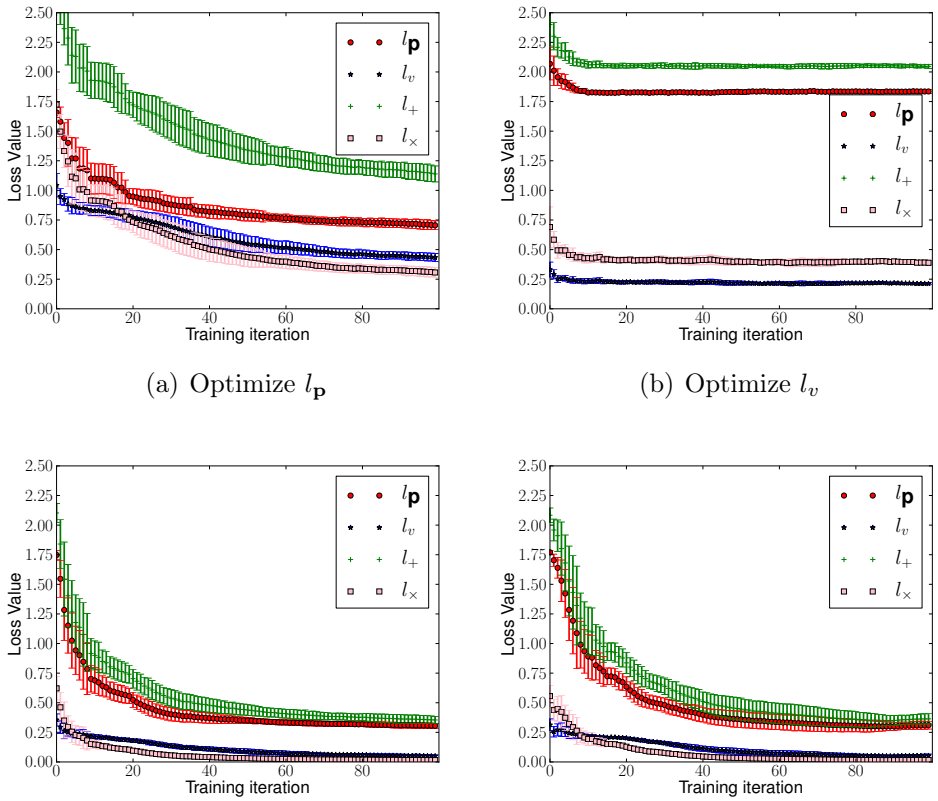
(c) Minimize l_+

(d) Minimize $l_\times$

Figure 4.4: Training losses for minimizing the four different targets in 5×5 Connect Four, averaged from 8 runs. l_v is always the lowest

when minimizing the $l_\times$ (Fig. 4.3(d)), with the difference that the final computed loss is much lower as the values are usually smaller than one. However, the similarity of the single losses is striking.

For 5×5 Connect Four (see Fig. 4.4(a)), we find that when only minimizing $l_{\mathbf{p}}$, it significantly reduces from 1.4 to about 0.6, whereas l_v is minimized much quicker from 1.0 to about 0.2, where it is almost stationary. Minimizing l_v (Fig. 4.4(b)) leads to some reduction from more than 0.5 to about 0.15, but $l_{\mathbf{p}}$ is not moving much after an initial slight decrease to about 1.6. For minimizing the l_+ (Fig. 4.4(c)) and the $l_\times$ (Fig. 4.4(d)), the behavior of $l_{\mathbf{p}}$ and l_v is very similar, they both decrease steadily, until l_v surprisingly reaches 0. Of course the l_+ and the $l_\times$ arrive at different values, but in terms of both $l_{\mathbf{p}}$ and l_v they are not different.



(a) Optimize $l_{\mathbf{p}}$

(b) Optimize l_v

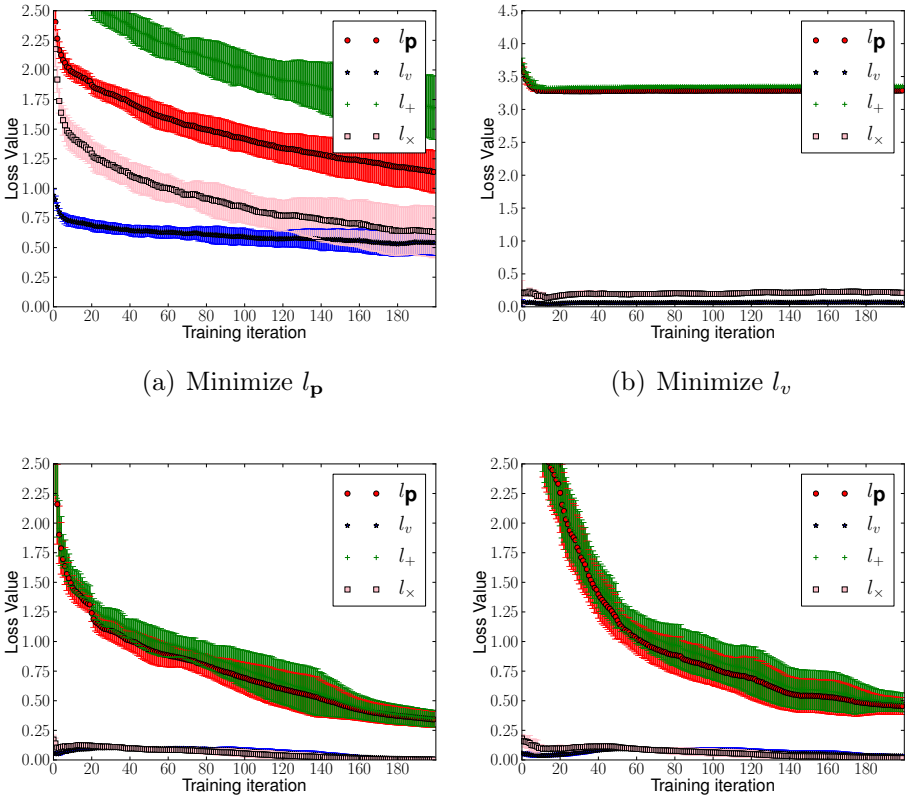
(c) Optimize sum

(d) Optimize product

Figure 4.5: Training losses for optimizing different targets in 6×6 Connect Four, averaged from 8 Runs. l_v is the lowest except for the product target

The training process of the larger 6×6 Connect Four is investigated in Fig 4.5(a). We find that optimizing $l_{\mathbf{p}}$ reduces it significantly from 1.7 to about 0.7 at the end of each training, where l_v is minimized from 1.2 to about 0.4. For the scenario with optimizing l_v (Fig 4.5(b)), we find a similar behavior than for the smaller Connect Four. After some initial progress, there is only stagnation. Again, for optimizing the sum and the product, the target value changes, but the single loss values $l_{\mathbf{p}}$ and l_v behave similarly (Figs 4.5(c) and 4.5(d)). Thus we see that both targets lead to very similar training processes.

For 5×5 Gobang game, we find that, in Fig. 4.6, when only minimizing $l_{\mathbf{p}}$, the $l_{\mathbf{p}}$ value decreases from around 2.5 to about 1.25 while the l_v value reduces from 1.0 to 0.5 (see Fig. 4.6(a)). When minimizing l_v, l_v value quickly reduces to a very low level which is lower than 0.1 (see Fig. 4.6(b)). Minimizing l_+ and $l_\times$ both



(a) Minimize $l_{\mathbf{p}}$

(b) Minimize l_v

(c) Minimize l_+

(d) Minimize $l_\times$

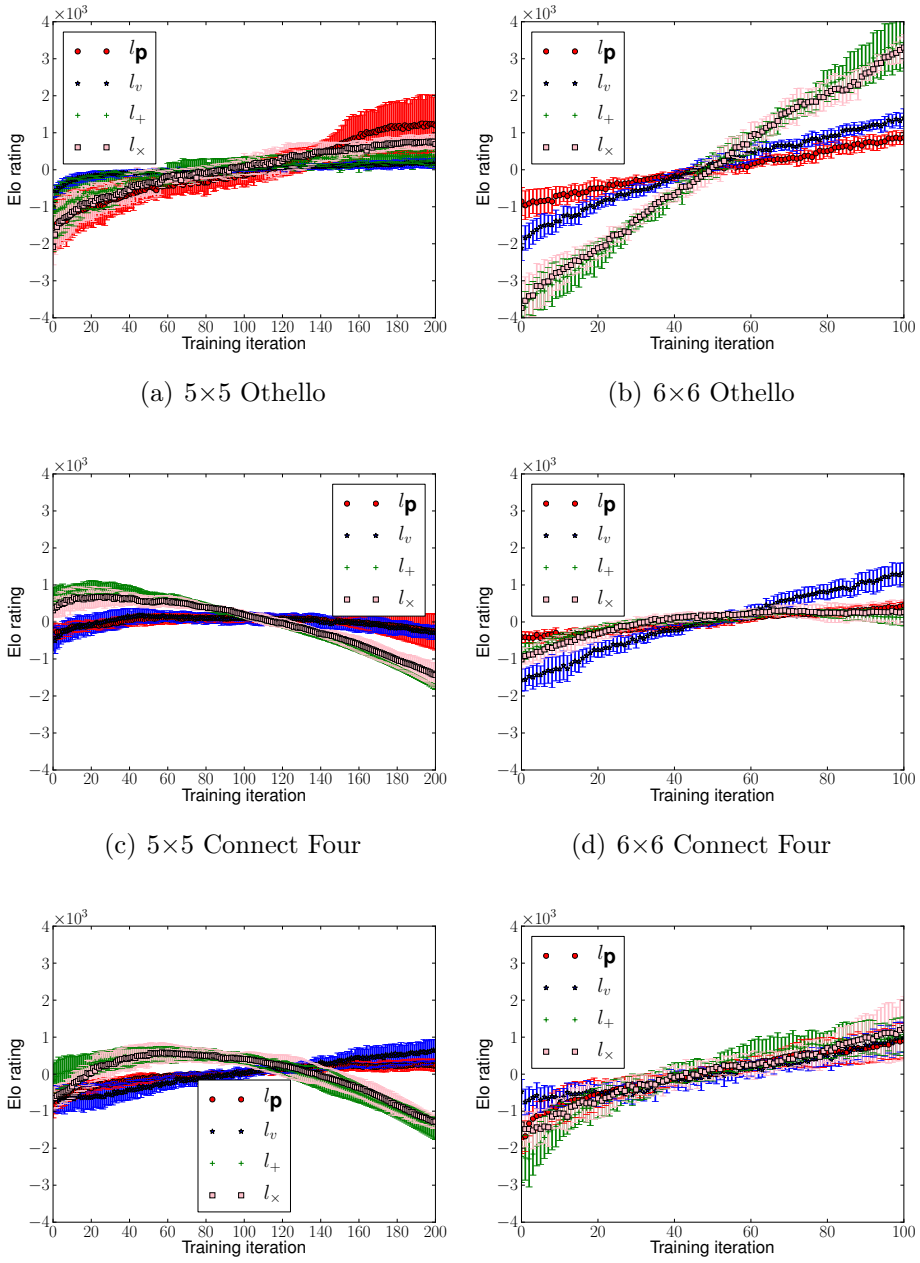
Figure 4.6: Training losses for minimizing the four different targets in 5×5 Gobang, averaged from 8 runs. l_v is always the lowest

lead to stationary low l_v levels from the beginning of training which is different from Othello and Connect Four.

4.6.2 Training Elo Rating

Following the AlphaGo papers, the training Elo rating of every iteration during training is investigated. Instead of showing results from single runs, means and variances for 8 runs for each target are provided, categorized by different games in Fig. 4.7.

From Fig. 4.7(a) (small 5×5 Othello) we see that for all minimization tasks, Elo values steadily improve, while they raise fastest for $l_{\mathbf{p}}$. In Fig. 4.7(b), we find that for 6×6 Othello version, Elo values also always improve, but much faster for



(a) 5×5 Othello

(b) 6×6 Othello

(c) 5×5 Connect Four

(d) 6×6 Connect Four

(e) 5×5 Gobang

(f) 6×6 Gobang

Figure 4.7: The whole history Elo rating at each iteration during training for different games, aggregated from 8 runs. The training Elo for l_+ and $l_\times$ in panel b and c for example shows inconsistent results

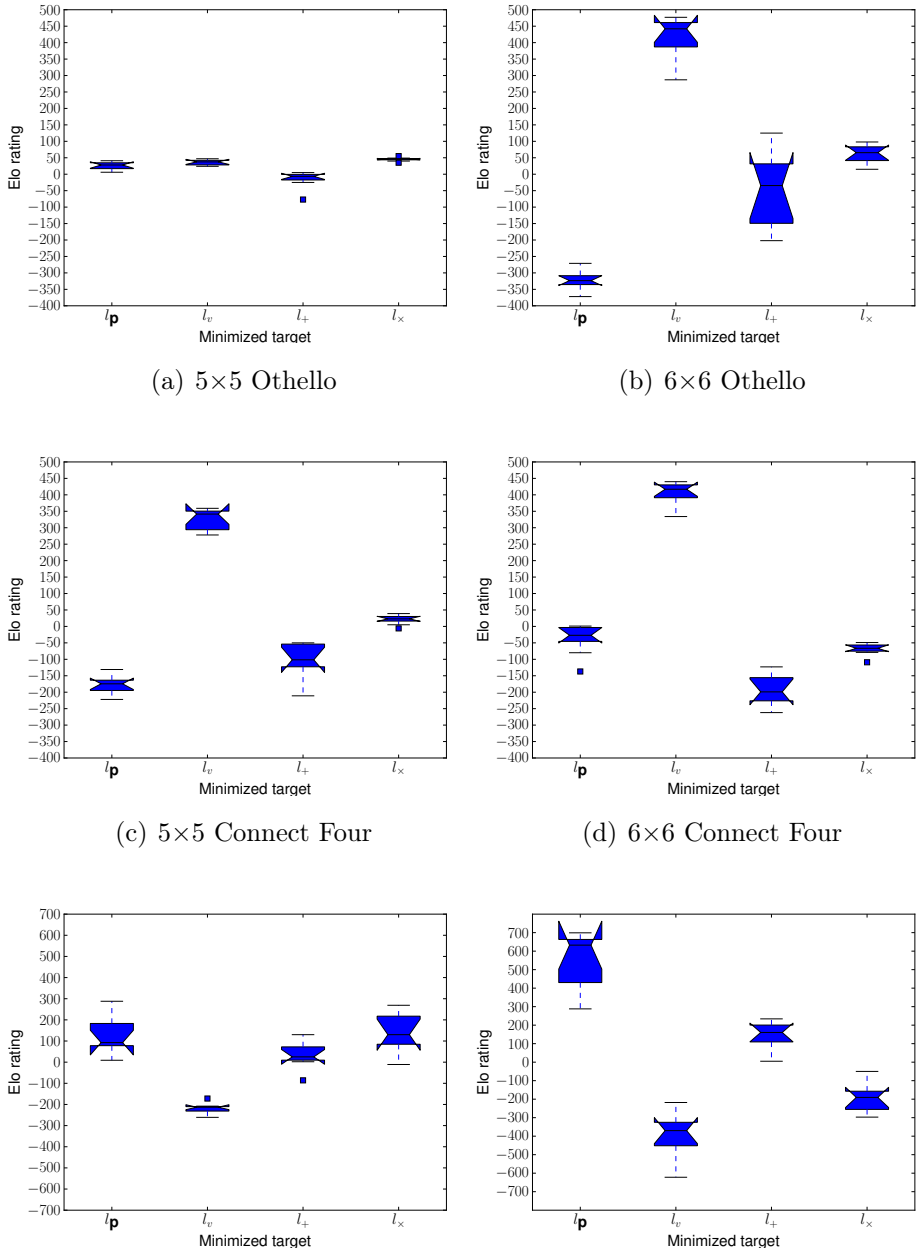
the l_+ and $l_\times$ target, compared to the single loss targets.

Fig. 4.7(c) and Fig. 4.7(d) show the Elo rate progression for training players with the four different targets on the small and larger Connect Four setting. This looks a bit different from the Othello results, as we find stagnation (for 6×6 Connect Four) as well as even degeneration (for 5×5 Connect Four). The latter actually means that for decreasing loss in the training phase, we achieve decreasing Elo rates, such that the players get weaker and not stronger. In the larger Connect Four setting, we still have a clear improvement, especially if we minimize for l_v. Minimizing for $l_\mathbf{p}$ leads to stagnation quickly, or at least to a very slow improvement.

Overall, we display the Elo progression obtained from the different minimization targets for one game together. However, one must be aware that their numbers are not directly comparable due to the high self-play bias (as they stem from players who have never played against each other). Nevertheless, the trends as observed for single self-play are of interest, and it is especially interesting to see if Elo values correlate with the progression of losses. Based on the experimental results, we can conclude that the training Elo rating is certainly good for assessing if training actually works, whereas the losses alone do not always show that. We may even experience contradicting outcomes as stagnating losses and rising Elo ratings (for the big Othello setting and l_v) or completely counterintuitive results as for the small Connect Four setting where Elo ratings and losses are partly anti-correlated. We have experimental evidence for the fact that training losses and Elo ratings are by no means interchangeable as they can provide very different impressions of what is actually happening.

4.6.3 The Final Best Player Tournament Elo Rating

In order to measure which target can achieve better playing strength, we let all final models trained from 8 runs and 4 targets plus a random player pit against each other for 20 times in a full round robin tournament. This enables a direct comparison of the final outcomes of the different training processes with different targets. It is thus more informative than the training Elo due to the self-play bias, but provides no information during the self-play training process. In principle, it is possible to do this also during the training at certain iterations, but this is computationally very expensive.



(a) 5×5 Othello

(b) 6×6 Othello

(c) 5×5 Connect Four

(d) 6×6 Connect Four

(e) 5×5 Gobang

(f) 6×6 Gobang

Figure 4.8: Round-robin tournament of all final models from minimizing different targets. For each game 8 final models from 4 different targets plus a random player (i.e. 33 in total). In panel (a) the difference is small. In panel b, c, and d, the Elo rating of l_v minimized players clearly dominates. However, in panel (f), the Elo rating of $l_{\mathbf{p}}$ minimized players clearly achieve the best performance.

The results are presented in Fig. 4.8. and show that minimizing l_v achieves the highest Elo rating with small variance for 6×6 Othello, 5×5 Connect Four and 6×6 Connect Four. For 5×5 Othello, with 200 training iterations, the difference between the results is small. We therefore presume that minimizing l_v is the best choice for the games we focus on. This is surprising because we expected the l_+ to perform best as documented in the literature. However, this may apply to smaller games only, and 5×5 Othello already seems to be a border case where overfitting levels out all differences.

In conclusion, we find that solely minimizing l_v is an alternative to the default sum objective l_+ in many cases. We also report exceptions, especially in relation to the Elo rating as calculated during training. The relation between Elo and loss during training is sometimes inconsistent (5×5 Connect Four training shows Elo decreasing while the losses are actually minimized) due to training bias. And for Gobang game, only minimizing $l_{\mathbf{p}}$ is the best alternative. A combination achieves lowest loss, but l_v achieves the highest training Elo. If we minimize product loss $l_\times$, this can result in higher Elo rating for certain games. More research (such as training bias and in which case which objective function (combination) should be employed) should be studied further.

4.7 Summary

Most function approximators in supervised learning and reinforcement learning use a single neural network with a single input and output. In reinforcement learning, this is either a policy or a value network. Alpha(Go) Zero innovatively minimizes *both* policy and value, using a single unified network with two heads, a policy head and a value head. Alpha(Go) Zero and other works minimize the sum of policy and value loss. Here, we study four different loss function combinations: (1) $l_{\mathbf{p}}$, (2) l_v, (3) l_+, (4) $l_\times$. We use the open source AlphaZeroGeneral system for light-weight self-play experiments on two small games, Connect Four and Othello. Surprisingly, we find that in many cases l_v achieves the highest tournament Elo rating, in contrast to the default sum objective in AlphaZero and AlphaZeroGeneral. The obtained experimental results in this chapter however indicate that relying on default setting, major performance gains are likely to be missed out. Much research in self-play is recently going on using the default loss function without questioning this default choice. More research is needed into the relative importance of value function and policy function in small games. Furthermore, default hyper-parameter settings may be non-optimal, especially for the smaller games we investigate here.

During training, we compute a running Elo rating. We find that the training losses trend and the Elo ratings trend are inconsistent in some games (5×5 Connect Four and 6×6 Othello). Training Elo, while cheap to compute, can be a misleading indicator of playing strength, because it is influenced by self-play training bias [17]. Our results provide the methodological contribution that for comparing playing strength among players, tournament Elo ratings should be used, instead of running training Elo ratings.

This chapter shows that the choice of the optimal combined loss function can have a huge impact on Elo performance. Unfortunately, our computational resources did not allow us to test the approach on large board sizes, but the results should encourage similar research of loss functions and alternative Elo computation also for large scale games.

Chapter 5

Warm-Starting AlphaZero-like Self-Play

5.1 Introduction

In Chapter 2, we showed that MCS enhancements can improve table based Q-learning, which suggests whether MCTS enhancements could also improve neural network based deep reinforment learning? We have seen that the AlphaGo series of programs [10, 16, 17] achieve impressive super human level performance in board games. Subsequently, there is much interest among deep reinforcement learning researchers in self-play, and self-play is applied to many applications [61, 62]. In self-play, MCTS [7] is used to train a deep neural network, that is then employed in tree searches, in which MCTS uses the network that it helped train in previous iterations.

On the one hand, self-play is utilized to generate game playing records and assign game rewards for each training example automatically. Next, these examples are fed to the neural network for improving the model. No database of labeled examples is used. Self-play learns tabula rasa, from scratch. However, self-play suffers from a cold-start problem, and may also easily suffer from bias since only a small part of the search space is used for training, and training samples in reinforcement learning are heavily correlated [17, 44].

On the other hand, the MCTS search enhances performance of the trained model by providing improved training examples. There has been much research into enhancements to improve MCTS [7, 74], but to the best of our knowledge, few of

these are used in Alphazero-like self-play, which we find surprising, given the large computational demands of self-play and the cold-start and bias problems.

A reason may be that AlphaZero-like self-play is still young. Another reason could be that the original AlphaGo paper [16] remarks about AMAF and RAVE [18], two of the best known MCTS enhancements, that "AlphaGo does not employ the *all-moves-as-first* (AMAF) or *rapid action value estimation* (RAVE) [24] heuristics used in the majority of Monte Carlo Go programs; when using policy networks as prior knowledge, these biased heuristics do not appear to give any additional benefit". Our experiments indicate otherwise, and we believe there is merit in exploring warm-start MCTS in an AlphaZero-like self-play setting.

We agree that when the policy network is well trained, then heuristics may not provide significant added benefit. However, when this policy network has not been well trained, especially at the beginning of the training, the neural network provides approximately random values for MCTS, which can lead to bad performance or biased training. The MCTS enhancements or specialized evolutionary algorithms such as *RHEA* [102] may benefit the searcher by compensating the weakness of the early neural network, providing better training examples at the start of iterative training for self-play, and quicker learning. Therefore, in this work, we first test the possibility of MCTS enhancements and RHEA for improving self-play, and then choose MCTS enhancements to do full scale experiments, the results show that MCTS with warm-start enhancements in the start period of AlphaZero-like self-play improve iterative training with tests on 3 different regular board games, using an AlphaZero re-implementation [65].

Our main contributions can be summarized as follows:

1. We test MCTS enhancements and RHEA, and then choose warm-start enhancements (Rollout, RAVE and their combinations) to improve MCTS in the start phase of iterative training to enhance AlphaZero-like self-play. Experimental results show that in all 3 tested games, the enhancements can achieve significantly higher Elo ratings, indicating that warm-start enhancements can improve AlphaZero-like self-play.

2. In our experiments, a weighted combination of Rollout and RAVE with a value from the neural network always achieves better performance, suggesting also for how many iterations to enable the warm-start enhancement.

This chapter is structured as follows. After giving an overview of the most relevant literature in Sect. 5.2, we describe the AlphaZero-like self-play algorithm in

Sect. 5.3. Before the full length experiments in Sect. 5.5, an orientation experiment is performed in Sect. 5.4. Finally, we summarize the chapter and discuss future work.

5.2 Related Work

Since MCTS was created [103], many variants have been studied [7, 104], especially in games [105]. In addition, enhancements such as RAVE and AMAF have been created to improve MCTS [18, 24]. Specifically, [24] can be regarded as one of the early prologues of the AlphaGo series, in the sense that it combines online search (MCTS with enhancements like RAVE) and offline knowledge (table based model) in playing small board Go.

In self-play, the large number of parameters in the deep network as well as the large number of hyper-parameters (see Table 5.2) are a black-box that precludes understanding. The high decision accuracy of deep learning [106], however, is undeniable [68], as the results in Go (and many other applications) have shown [69]. After AlphaGo Zero [17], which uses an MCTS searcher for training a neural network model in a self-play loop, the role of self-play has become more and more important. The neural network has two heads: a policy head and a value head, aimed at learning the best next move, and the assessment of the current board state, respectively.

Earlier works on self-play in reinforcement learning are [70, 71, 72, 73, 107]. An overview is provided in [74]. For instance, [70, 72] compared self-play and using an expert to play backgammon with temporal difference learning. [107] studied co-evolution versus self-play temporal difference learning for acquiring position evaluation in small board Go. All these works suggest promising results for self-play.

More recently, [30] assessed the potential of classical Q-learning by introducing MCS enhancement to improve training examples efficiency. [108] uses domain-specific features and optimizations, but still starts from random initialization and makes no use of outside strategic knowledge or preexisting data, that can accelerate the AlphaZero-like self-play. Cazenave et al. improved the Zero learning using different structure of neural networks [109]. And Albert Silver improved the Fat Fritz by learning from the surgical precision of Stockfish's [10] legendary search with a massive new neural network [110].

However, to the best of our knowledge there is no further study on applying MCTS enhancements in AlphaZero-like self-play despite the existence of many practical and powerful enhancements.

5.3 AlphaZero-like Self-play Algorithms

5.3.1 The Algorithm Framework

According to [10, 34], the basic structure of warm-start AlphaZero-like self-play is also an iterative process over three different stages (see Algorithm 3).

Algorithm 3 Warm-Start AlphaZero-like Self-play Algorithm

function AlphaZeroGeneralwithEnhancements
  Initialize f_θ with random weights; Initialize retrain buffer D with capacity N
  for iteration=1, ...,I', ..., I **do** ▷ play curriculum of I tournaments
    for episode=1,..., E **do** ▷ stage 1, play tournament of E games
      for t=1, ..., T', ..., T **do** ▷ play game of T moves
        $\pi_t \leftarrow$ **MCTS Enhancement** before I' or **MCTS** after I' iteration
        a_t =randomly select on π_t before T' or $\arg\max_a(\pi_t)$ after T' step
        executeAction(s_t, a_t)
      Store every (s_t, π_t, z_t) with game outcome z_t $(t \in [1, T])$ in D
    Randomly sample minibatch of examples (s_j, π_j, z_j) from D ▷ stage 2
    Train $f_{\theta'} \leftarrow f_\theta$
    $f_\theta = f_{\theta'}$ if $f_{\theta'}$ is better than f_θ using **MCTS** mini-tournament ▷ stage 3
  return f_θ;

The first stage is a **self-play** tournament. The player plays several games against itself to generate game playing records as training examples. In each step of a game episode, the player runs MCTS (or one of the MCTS enhancements before *I'* iteration) to obtain, for each move, an enhanced policy π based on the probability **p** provided by the policy network f_θ. The hyper-parameters, and the abbreviation that we use in this chapter is given in Table 5.2. In MCTS, hyper-parameter C_p is used to balance exploration and exploitation of the tree search, and we abbreviate it to c. Hyper-parameter m is the number of times to search down from the root for building the game tree, where the value (v) of the states is provided by f_θ. In (self-)play game episode, from *T'* steps on, the player always chooses the best action based on π. Before that, the player always chooses a random move according to the probability distribution of π to obtain more diverse training

examples. After the game ends, the new examples are normalized as a form of (s_t, π_t, z_t) and stored in D.

The second stage consists of **neural network training**, using data from stage 1. Several epochs are usually employed for the training. In each epoch (ep), training examples are randomly selected as several small batches [79] based on the specific batch size (bs). The neural network is trained with a learning rate (lr) and dropout (d) by minimizing [80] the value of the *loss function* which is the sum of the mean-squared error between predicted outcome and real outcome and the cross-entropy losses between $\mathbf{p}$ and π. Dropout is a probability to randomly ignore some nodes of the hidden layer to avoid overfitting [81].

The last stage is the **arena comparison**, where a competition between the newly trained neural network model (f'_θ) and the previous neural network model (f_θ) is run. The winner is adopted for the next iteration. In order to achieve this, the competition runs n rounds of the game. If $f_{\theta'}$ wins more than a fraction of u games, it is accepted to replace the previous best f_θ. Otherwise, $f_{\theta'}$ is rejected and f_θ is kept as current best model. Compared with AlphaGo Zero, AlphaZero does not employ this stage anymore. However, we keep it to make sure that we can safely recognize improvements.

Algorithm 4 Neural Network Based MCTS

```
1: function MCTS(s, f_θ)
2:     Search(s)
3:     π_s ←normalize(Q(s, ·))
4:     return π_s
5: function SEARCH(s)
6:     Return game end result if s is a terminal state
7:     if s is not in the Tree then
8:         Add s to the Tree, initialize Q(s, ·) and N(s, ·) to 0
9:         Get P(s, ·) and v(s) by looking up f_θ(s)
10:        return v(s)
11:    else
12:        Select an action a with highest UCT value
13:        s' ←getNextState(s, a)
14:        v ←Search(s')
15:        Q(s, a) ← (N(s,a)*Q(s,a)+v) / (N(s,a)+1)
16:        N(s, a) ← N(s, a) + 1
17:    return v;
```

5.3.2 MCTS

In self-play, MCTS is used to generate high quality examples for training the neural network. A recursive MCTS pseudo code is given in Algorithm 4. For each search, the value from the value head of the neural network is returned (or the game termination reward, if the game terminates). During the search, for each visit of a non-leaf node, the action with the highest P-UCT value is selected to investigate next [17, 111]. After the search, the average win rate value $Q(s, a)$ and visit count $N(s, a)$ in the followed trajectory are updated correspondingly. The P-UCT formula that is used is as follows (with c as constant weight that balances exploitation and exploration):

$$U(s, a) = Q(s, a) + c * P(s, a) \frac{\sqrt{N(s, \cdot)}}{N(s, a) + 1} \tag{5.1}$$

In the whole training iterations (including the first I' iterations), the **Baseline** player always runs neural network based MCTS (i.e line 6 in Algorithm 3 is simply replaced by $\pi_t \leftarrow$ **MCTS**).

5.3.3 MCTS Enhancements

In this chapter, we introduce 2 individual enhancements and 3 combinations to improve neural network training based on MCTS (Algorithm 4).

Rollout Algorithm 4 uses the value from the value network as return value at leaf nodes. However, if the neural network is not yet well trained, the values are not accurate, and even random at the start phase, which can lead to biased and slow training. Therefore, as warm-start enhancement we perform a classic MCTS random rollout to get a value that provides more meaningful information. We thus simply add a random rollout function which returns a terminal value after line 9 in Algorithm 4, written as *Get result v(s) by performing random rollout until the game ends.*[1] See Algorithm 10 in Appendix A.3.

RAVE is a well-studied enhancement for improving the cold-start of MCTS in games like Go (for details see [18]). The same idea can be applied to other domains where the playout-sequence can be transposed. Standard MCTS only updates the (s, a)-pair that has been visited. The RAVE enhancement extends this rule to any action a that appears in the sub-sequence, thereby rapidly collecting more statistics in an off-policy fashion. The idea to perform RAVE at

[1] In contrast to AlphaGo [16], where random rollouts were mixed in with all value-lookups, in our scheme they replace the network lookup at the start of the training.

startup is adapted from AMAF in the game of Go [18]. The main pseudo code of RAVE is similar to Algorithm 4, the differences are in line 3, line 12 and line 16. For RAVE, in line 3, policy π_s is normalized based on $Q_{rave}(s,\cdot)$. In line 12, the action a with highest UCT_{rave} value, which is computed based on Equation 5.2, is selected. After line 16, the idea of AMAF is applied to update N_{rave} and Q_{rave}, which are written as: $N_{rave}(s_{t_1},a_{t_2}) \leftarrow N_{rave}(s_{t_1},a_{t_2})+1$, $Q_{rave}(s_{t_1},a_{t_2}) \leftarrow \frac{N_{rave}(s_{t_1},a_{t_2})*Q_{rave}(s_{t_1},a_{t_2})+v}{N_{rave}(s_{t_1},a_{t_2})+1}$, where $s_{t_1} \in VisitedPath$, and $a_{t_2} \in A(s_{t_1})$, and for $\forall t < t_2, a_t \neq a_{t_2}$. More specifically, under state s_t, in the visited path, a state s_{t_1}, all legal actions a_{t_2} of s_{t_1} that appear in its sub-sequence ($t \leq t_1 < t_2$) are considered as a (s_{t_1},a_{t_2}) tuple to update their Q_{rave} and N_{rave}. See Algorithm 11 in Appendix A.3.

$$UCT_{rave}(s,a) = (1-\beta)*U(s,a) + \beta * U_{rave}(s,a) \tag{5.2}$$

where

$$U_{rave}(s,a) = Q_{rave}(s,a) + c * P(s,a)\frac{\sqrt{N_{rave}(s,\cdot)}}{N_{rave}(s,a)+1}, \tag{5.3}$$

and

$$\beta = \sqrt{\frac{equivalence}{3*N(s,\cdot)+equivalence}} \tag{5.4}$$

Usually, the value of equivalence is set to the number of MCTS simulations (i.e m), as is also the case in our following experiments.

RoRa Based on Rollout and Rave enhancement, the first combination is to simply add the random rollout to enhance RAVE. See Algorithm 12 in Appendix A.3.

WRo As the neural network model is getting better, we introduce a weighted sum of rollout value and the value network as the return value. See Algorithm 13 in Appendix A.3. In our experiments, $v(s)$ is computed as follows:

$$v(s) = (1-weight)*v_{network} + weight * v_{rollout} \tag{5.5}$$

WRoRa In addition, we also employ a weighted sum to combine the value a neural network and the value of RoRa. See Algorithm 14 in Appendix A.3. In our experiments, weight $weight$ is related to the current iteration number $i, i \in [0, I']$. $v(s)$ is computed as follows:

$$v(s) = (1-weight)*v_{network} + weight * v_{rora} \tag{5.6}$$

where

$$weight = 1 - \frac{i}{I'} \tag{5.7}$$

5.4 Initial Experiment: MCTS(RAVE) vs. RHEA

Before running full scale experiments on warm-start self-play that take days to weeks, we consider other possibilities for methods that could be used instead of MCTS variants. Justesen et al. [102] have recently shown that depending on the type of game that is played, RHEA can actually outperform MCTS variants also on adversarial games. Especially for long games, RHEA seems to be strong because MCTS is not able to reach a good tree/opening sequence coverage.

The general idea of RHEA has been conceived by Perez et al. [112] and is simple: they directly optimize an action sequence for the next actions and apply the first action of the best found sequence for every move. Originally, this has been applied to one-player settings only, but recently different approaches have been tried also for adversarial games, as the co-evolutionary variant of Liu et al. [113] that shows to be competitive in 2 player competitions [114]. The current state of RHEA is documented in [115], where a large number of variants, operators and parameter settings is listed. No one-beats-all variant is known at this moment.

Generally, the horizon (number of actions in the planned sequence) is often much too short to reach the end of the game. In this case, either a value function is used to assess the last reached state, or a rollout is added. For adversarial games, opponent moves are either co-evolved, or also played randomly. We do the latter, with a horizon size of 10. In preliminary experiments, we found that a number of 100 rollouts is already working well for MCTS on our problems, thus we also applied this for the RHEA. In order to use these 100 rollouts well, we employ a population of only 10 individuals, using only cloning+mutation (no crossover) and a (10+1) truncation selection (the worst individual from 10 parents and 1 offspring is removed). The mutation rate is set to 0.2 per action in the sequence. However, parameters are not sensitive, except rollouts. RHEA already works with 50 rollouts, albeit worse than with 100. As our rollouts always reach the end of the game, we usually get back $Q_i(as) = \{1, -1\}$ for the i-th rollout for the action sequence as, meaning we win or lose. Counting the number of steps until this happens h, we compute the fitness of an individual to $Q(as) = \frac{\sum_{i=1}^{n} Q_i(as)/h}{n}$ over multiple rollouts, thereby rewarding quick wins and slow losses. We choose $n = 2$ (rollouts per individual) as it seems to perform a bit more stable than $n = 1$. We thus evaluate 50 individuals per run.

In our comparison experiment, we pit a random player, MCTS, RAVE (both without neural network support but a standard random rollout), and RHEA (see Algorithm 15 in Appendix A.3) against each other with 500 repetitions over all

three games, with 100 rollouts per run for all methods. The results are shown in Table 5.1.

Table 5.1: Comparison of random player, MCTS, Rave, and RHEA on the three games, win rates in percent (column vs row), 500 repetitions each.

	Gobang				Connect Four				Othello			
adv	rand	mcts	rave	rhea	rand	mcts	rave	rhea	rand	mcts	rave	rhea
random		97.0	100.0	90.0		99.6	100.0	80.0		98.50	98.0	48.0
mcts	3.0		89.4	34.0	0.4		73.0	3.0	1.4		46.0	1.0
rave	0.0	10.6		17.0	0.0	27.0		4.0	2.0	54.0		5.0
rhea	10.0	66.0	83.0		20.0	97.0	96.0		52.0	99.0	95.0	

The results indicate that in nearly all cases, RAVE is better than MCTS is better than RHEA is better than random, according to a binomial test at a significance level of 5%. Only for Othello, RHEA does not convincingly beat the random player. We can conclude from these results that RHEA is no suitable alternative in our case. The reason for this may be that the games are rather short so that we always reach the end, providing good conditions for MCTS and even more so for RAVE that more aggressively summarizes rollout information. Besides, start sequence planning is certainly harder for Othello where a single move can change large parts of the board.

5.5 Full Length Experiment

Taking into account the results of the comparison of standard MCTS/RAVE and RHEA at small scale, we now focus on the previously defined neural network based MCTS and its enhancements and run them over the full scale training.

5.5.1 Experiment Setup

For all 3 tested games and all experimental training runs based on Algorithm 3, we set parameters values in Table 5.2. Since tuning I' requires enormous computation resources, we set the value to 5 based on an initial experiment test, which means that for each self-play training, only the first 5 iterations will use one of the warm-start enhancements, after that, there will be only the MCTS in Algorithm 4. Other parameter values are set based on [32, 33].

Our experiments are run on a GPU-machine with 2x Xeon Gold 6128 CPU at 2.6GHz, 12 core, 384GB RAM and 4x NVIDIA PNY GeForce RTX 2080TI. We use small versions of games (6×6) in order to perform a sufficiently high number

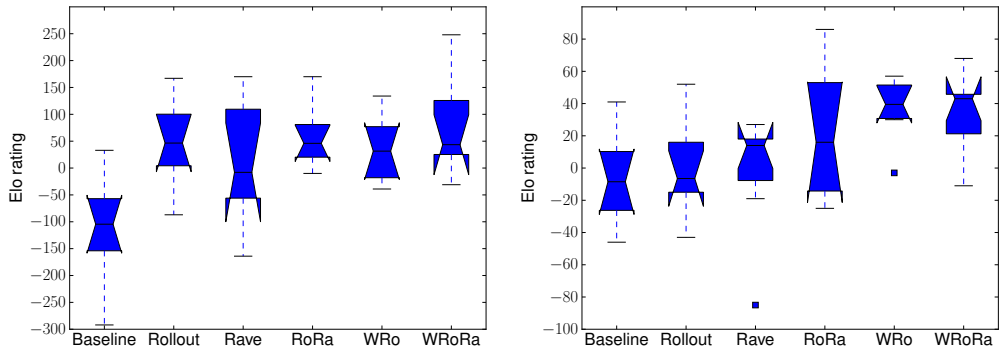
of computationally demanding experiments. Shown are graphs with errorbars of 8 runs, of 100 iterations of self-play. Each single run takes 1 to 2 days.

Table 5.2: Default Parameter Setting

Para	Description	Value	Para	Description	Value
I	number of iteration	100	rs	number of retrain iteration	20
I'	iteration threshold	5	ep	number of epoch	10
E	number of episode	50	bs	batch size	64
T'	step threshold	15	lr	learning rate	0.005
m	MCTS simulation times	100	d	dropout probability	0.3
c	weight in UCT	1.0	n	number of comparison games	40
u	update threshold	0.6			

5.5.2 Results

After training, we collect 8 repetitions for all 6 categories players. Therefore we obtain 49 players in total (a Random player is included for comparison). In a full round robin tournament, every 2 of these 49 players are set to pit against each other for 20 matches on 3 different board games (Gobang, Connect Four and Othello). The Elo ratings are calculated based on the competition results using the same Bayesian Elo computation [82] as AlphaGo papers.



(a) 6×6 Gobang

(b) 6×6 Connect Four

Figure 5.1: Tournament results for 6×6 Gobang and 6×6 Connect Four among *Baseline*, *Rollout*, *Rave*, *RoRa*, *WRo* and *WRoRa*. Training with enhancements tends to be better than baseline MCTS.

Fig. 5.1(a) displays results for training to play the 6×6 Gobang game. We can clearly see that all players with the enhancement achieve higher Elo ratings than the Baseline player. For the Baseline player, the average Elo rating is about -100. For enhancement players, the average Elo ratings are about 50, except for Rave, whose variance is larger. Rollout players and its combinations are better than the single Rave enhancement players in terms of the average Elo. In addition, the combination of Rollout and RAVE does not achieve significant improvement of Rollout, but is better than RAVE. This indicates than the contribution of the Rollout enhancement is larger than RAVE in Gobang game.

Figure 5.1(b) shows that all players with warm-start enhancement achieve higher Elo ratings in training to play the 6×6 Connect Four game. In addition, we find that comparing Rollout with WRo, a weighted sum of rollout value and neural network value achieves higher performance. Comparing Rave and WRoRa, we see the same. We conclude that in 5 iterations, for Connect Four, enhancements that combine the value derived from the neural network contribute more than the pure enhancement value. Interestingly, in Connect Four, the combination of Rollout and RAVE shows improvement, in contrast to Othello (next figure) where we do not see significant improvement. However, this does not apply to WRoRa, the weighted case.

In Fig 5.2 we see that in Othello, except for Rollout which holds the similar Elo rating as Baseline setting, all other investigated enhancements are better than the Baseline. Interestingly, the enhancement with weighted sum of RoRa and neural network value achieves significant highest Elo rating. The reason that Rollout does not show much improvement could be that the rollout number is not large enough for the game length (6×6 Othello needs 32 steps for every episode to reach the game end, other 2 games above may end up with vacant positions). In addition, Othello does not have many transposes as Gobang and Connect Four which means that RAVE can not contribute to a significant improvement. We can definitively state that the improvements of these enhancements are sensitive to the different games. In addition, for all 3 tested games, at least WRoRa achieves the best performance according to a binomial test at a significance level of 5%.

5.6 Summary

Self-play has achieved much interest due to the AlphaGo Zero results. However, self-play is currently computationally very demanding, which hinders reproducibility and experimenting for further improvements. In order to improve

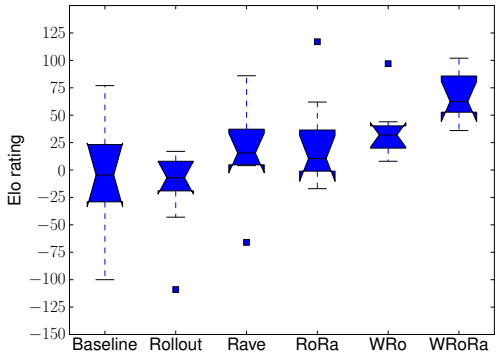


Figure 5.2: Tournament results for 6×6 Othello among *Baseline*, *Rollout*, *Rave*, *RoRa*, *WRo* and *WRoRa*. Training with enhancements is mostly better than the baseline setting.

performance and speed up training, in this chapter, we investigate the possibility of utilizing MCTS enhancements to improve AlphaZero-like self-play. We embed Rollout, RAVE and their possible combinations as enhancements at the start period of iterative self-play training. The hypothesis is, that self-play suffers from a cold-start problem, as the neural network and the MCTS statistics are initialized to random weights and zero, and that this can be cured by prepending it with running MCTS enhancements or similar methods alone in order to train the neural network before "switching it on" for playing.

We introduce Rollout, RAVE, and combinations with network values, in order to quickly improve MCTS tree statistics before we switch to Baseline-like self-play training, and test these enhancements on 6x6 versions of Gobang, Connect Four, and Othello. We find that, after 100 self-play iterations, we still see the effects of the warm-start enhancements as playing strength has improved in many cases. For different games, different methods work best; there is at least one combination that performs better. It is hardly possible to explain the performance coming from the warm-start enhancements and especially to predict for which games they perform well, but there seems to be a pattern: Games that enable good static opening plans probably benefit more. For human players, it is a common strategy in Connect Four to play a middle column first as this enables many good follow-up moves. In Gobang, the situation is similar, only in 2D. It is thus harder to counter a good plan because there are so many possibilities. This could be the reason why the warm-start enhancements work so well here. For Othello, the situation is different, static openings are hardly possible, and are thus seemingly

not detected. One could hypothesize that the warm-start enhancements recover human expert knowledge in a generic way. Recently, we have seen that human knowledge is essential for mastering complex games as StarCraft [23], whereas others as Go [17] can be learned from scratch. Re-generating human knowledge may still be an advantage, even in the latter case.

We also find that often, a single enhancement may not lead to significant improvement. There is a tendency for the enhancements that work in combination with the value of the neural network to be stronger, but that also depends on the game. Concluding, we can state that we find moderate performance improvements when applying warm-start enhancements and that we expect there is untapped potential for more performance gains here.

Chapter 6

Adaptive Warm-Start AlphaZero-like Self-play

6.1 Introduction

Following Chapter 5, in this chapter, we will further propose an adaptive warm-start method on AlphaZero-like deep reinforcement learning framework.

The combination of online MCTS [7] in self-play and offline neural network training has been widely applied as a deep reinforcement learning technique, in particular for solving game-related problems by means of the AlphaGo series programs [10, 16, 17]. The approach of this paradigm is to use game playing records from self-play by MCTS as training examples to train the neural network, whereas this trained neural network is used to inform the MCTS value and policy. Note that in contrast to AlphaGo Zero or AlphaZero, the original AlphaGo also uses large amounts of expert data to train the neural network and a fast rollout policy together with the policy provided by neural network to guide the MCTS search.

However, although the transition from a combination of using expert data and self-play (AlphaGo) to only using self-play (AlphaGo Zero and AlphaZero) appears to have only positive results, it does raise some questions.

The first question is: 'should all human expert data be abandoned?' In other games we have seen that human knowledge is essential for mastering complex

games, such as StarCraft [23]. Then when should expert data be taken into consideration while training neural networks?

The second question is: 'should the fast rollout policy be abandoned?' Chapter 5 has proposed to use warm-start search enhancements **at the start phase** in AlphaZero-like self-play, which improves performance in 3 small board games. Instead of only using the neural network for value and policy, in the first few iterations, classic rollout can be used (or RAVE, or a combination, or a combination with the neural network). This can improve training especially at the start phase of self-play training.

In fact, the essence of the warm-start search enhancement is to re-generate expert knowledge in the start phase of self-play training, to reduce the cold-start problem of playing against untrained agents. The method uses rollout (which can be seen as experts) instead of a randomly initialized neural network, up until a number of I' iterations, when it switches to the regular value network. In their experiments, the I' was fixed at 5. Obviously, a fixed I' may not be optimal. Therefore, in this work, we propose an adaptive switch method. The method uses an **arena** in the self-play stage (see Algorithm 6), where the search enhancement and the default MCTS are matched, to judge whether to switch or not. With this mechanism, we can dynamically switch off the enhancement if it is no longer better than the default MCTS player, as the neural network is being trained.

Our main contributions can be summarized as follows:

1. Warm-start method improves the Elo of AlphaZero-like self-play in small games, but it introduces a new hyper-parameter. Adaptive warm-start further improves performance and removes the hyper-parameter.

2. For deep games (with a small branching factor) warm-start works better than for shallow games. This indicates that the effectiveness of warm-start method may increase for larger games.

The rest of this chapter is designed as follows. An overview of the most relevant literature is given in Sect. 6.2. Before proposing our adaptive switch method in Sect. 6.4, we describe the warm-start AlphaZero-like self-play algorithm in Sect. 6.3. Thereafter, we set up the experiments in Sect. 6.5 and present their results in Sect. 6.6. Finally, we conclude the chapter and discuss future work.

6.2 Related Work

There are a lot of early successful works in reinforcement learning [6], e.g. using temporal difference learning with a neural network to play backgammon [70]. MCTS has also been well studied, and many variants/enhancements were designed to solve problems in the domain of sequential decisions, especially on games. For example, enhancements such as RAVE and All Moves as First (AMAF) have been conceived to improve MCTS [18, 24]. The AlphaGo series algorithms replace the table based model with a deep neural network based model, where the neural network has a policy head (for evaluating of a state) and a value head (for learning a best action) [34], enabled by the GPU hardware development. Thereafter, the structure that combines MCTS with neural network training has become a typical approach for reinforcement learning tasks and many successful applications [37, 89] of this kind model-based deep reinforcement learning [68]. Comparing AlphaGo with AlphaGo Zero and AlphaZero, the latter did not use any expert data to train the neural network, and abandoned the fast rollout policy for improving the MCTS on the trained neural network. Therefore, all training data is generated purely by self-play, which is also a very important feature of reinforcement learning. We base our work on an open reimplementation of AlphaZero, AlphaZero General [65].

There are many interesting works on self-play in reinforcement learning [70, 74, 107]. Temporal difference learning for acquiring position evaluation in small board Go with co-evolution has been compared to self-play [107]. These works demonstrated the impressive results for self-play and emphasized its importance.

Within a GGP framework, in order to improve training examples efficiency, [31] assessed the potential of classical Q-learning by introducing MCS enhancements. In an AlphaZero-like self-play framework, [108] used domain-specific features and optimizations, starting from random initialization and no preexisting data, to accelerate the training.

However, AlphaStar, the acclaimed algorithm for beating human professionals at StarCraft [23], went back to utilizing human expert data, thereby suggesting that this is still an option at the start phase of training. Apart from this, there are few studies on applying MCTS enhancements in AlphaZero-like self-play. Only [35] (presented in Chapter 5), which proposed a warm-start search enhancement method, pointed out the promising potential of utilizing MCTS enhancements (like a rollout policy) to re-generate expert data at the start phase of training. Our approach differs from AlphaStar, as we generate expert data using MCTS enhancements other than collecting it from humans; further, compared to

the static warm-start in Chapter 5, we propose an adaptive method to control the iteration length of using such enhancements instead of a fixed I'.

6.3 Warm-Start AlphaZero Self-play

Based on Chapter 5, we will now briefly recall the warm-start enhancement method.

6.3.1 The Algorithm Framework

Based on [10, 34] and Chapter 5, the core of AlphaZero-like self-play (see Algorithm 5) is an iterative loop which consists of three different stages within the single iteration as follows:

1. **self-play**: The first stage is playing several games against with itself to generate training examples.
2. **neural network training**: The second stage is feeding the neural network with training examples (generated in the first stage) to train a new model.
3. **arena comparison**: The last stage is employing a tournament to compare the newly trained model and the old model to decide whether to update or not.

The detail description of these 3 stages can be found in Chapter 5 (Sect. 5.3.1). Note that in the Algorithm 5, line 5, a fixed I' is employed to control whether to use neural network MCTS or MCTS enhancements, the I' should be set as relatively smaller than I, which is known as warm-start search. The MCTS algorithm and MCTS enhancements will be introduced in next subsections.

6.3.2 MCTS

Classical MCTS has shown successful performance to solve complex games, by taking random samples in the search space to evaluate the state value. Basically, the classical MCTS algorithm can be divided into 4 stages, which are known as *selection*, *expansion*, *rollout* and *backpropagate* [7]. However, for the default MCTS in AlphaZero-like self-play (eg. our Baseline), the neural network directly informs the MCTS state policy and value to guide the search instead of running a rollout (see Algorithm 4).

Algorithm 5 Warm-start AlphaZero-like Self-play Algorithm

Randomly initialize f_θ, assign retrain buffer D
for iteration=1, ...,I', ..., I **do**
  for episode=1,..., E **do** ▷ self-play
    for t=1, ..., T', ..., T **do**
      if $I \leq I'$ **then** $\pi_t \leftarrow$ **MCTS Enhancement**
      else $\pi_t \leftarrow$ **default MCTS**
      if $t \leq T'$ **then** a_t = randomly select on π_t
      else $a_t = \arg\max_a(\pi_t)$
      executeAction(s_t, a_t)
    $D \leftarrow (s_t, \pi_t, z_t)$ with outcome $z_{t\in[1,T]}$
  Sample minibatch (s_j, π_j, z_j) from D ▷ training
  Train $f_{\theta'} \leftarrow f_\theta$
  $f_\theta = f_{\theta'}$ if $f_{\theta'}$ is better, using **default MCTS** ▷ arena
return f_θ;

6.3.3 MCTS enhancements

In this chapter, we adopt the same two individual enhancements and three combinations to improve neural network training as were used in Chapter 5. Here we briefly recall them again with a possible minor change for *weight* calculation.

Rollout is running a classic MCTS random rollout to get a value that provides more meaningful information than a value from random initialized neural network.

RAVE is a well-studied enhancement to cope with the cold-start of MCTS in games like Go [18], where the playout-sequence can be transposed. The core idea of RAVE is using AMAF to update the state visit count N_{rave} and Q-value Q_{rave}, which are written as: $N_{rave}(s_{t_1}, a_{t_2}) \leftarrow N_{rave}(s_{t_1}, a_{t_2}) + 1$, $Q_{rave}(s_{t_1}, a_{t_2}) \leftarrow \frac{N_{rave}(s_{t_1},a_{t_2}) * Q_{rave}(s_{t_1},a_{t_2}) + v}{N_{rave}(s_{t_1},a_{t_2})+1}$, where $s_{t_1} \in VisitedPath$, and $a_{t_2} \in A(s_{t_1})$, and for $\forall t < t_2, a_t \neq a_{t_2}$. The P-UCT of RAVE is calculated as follows:

$$PUCT_{rave}(s,a) = (1-\beta) * U(s,a) + \beta * U_{rave}(s,a) \tag{6.1}$$

where

$$U_{rave}(s,a) = Q_{rave}(s,a) + c * P(s,a)\frac{\sqrt{N_{rave}(s,\cdot)}}{N_{rave}(s,a)+1} \tag{6.2}$$

and

$$\beta = \sqrt{\frac{equivalence}{3 * N(s, \cdot) + equivalence}} \tag{6.3}$$

The value of equivalence is usually set to the number of MCTS simulations (i.e m=100 in our experiments).

RoRa is the combination which simply adds the random rollout to enhance RAVE.

WRo introduces a weighted sum of rollout value and the neural network value as the return value to guide MCTS. In our experiments, $v(s)$ is computed as follows:

$$v(s) = (1 - weight) * v_{network} + weight * v_{rollout} \tag{6.4}$$

WRoRa also employs a weighted sum to combine the value from the neural network and the value of RoRa. The $v(s)$ for MCTS search in WRoRa is computed as follows:

$$v(s) = (1 - weight) * v_{network} + weight * v_{rora} \tag{6.5}$$

Different from Chapter 5, since there is no pre-determined I', in our work, $weight$ is simply calculated as $1/i, i \in [1, I]$, where i is the current iteration number.

6.4 Adaptive Warm-Start Switch Method

The fixed I' to control the length of using warm-start search enhancements as suggested in Chapter 5, but seems to require different parameter values for different games. In consequence, a costly tuning process would be necessary for each game. Thus, an adaptive method would have multiple advantages.

We notice that the core of the warm-start method is re-generating expert data to train the neural network at the start phase of self-training to avoid learning from weak (random or near random) self-play. We suggest to stop the warm-start when the neural network is on average playing stronger than the enhancements. Therefore, in the self-play, we employ a tournament to compare the standard AlphaZero-like self-play model (Baseline) and the enhancements (see Algorithm 6). The switch occurs once the Baseline MCTS wins more than 50%. In order to avoid spending too much time on this, these arena game records will directly be used as training examples, indicating that the training data is played by the enhancements and the Baseline. This scheme enables to switch at individual points in time for different games and even different training runs.

Algorithm 6 Adaptive Warm-Start Switch Algorithm

Initialize f_θ with random weights; Initialize retrain buffer D, Switch←False, $r_{mcts} \leftarrow 0$
for iteration=1, ..., I **do** ▷ no I'
  if not Switch **then** ▷ not switch
    for episode=1,..., E **do** ▷ arena with enhancements
      for t=1, ..., T', ..., T **do**
        if *episode* $\leq E/2$ **then**
          if t is odd **then** $\pi_t \leftarrow$ **MCTS Enhancement**
          else $\pi_t \leftarrow$ **default MCTS**
        else
          if t is odd **then** $\pi_t \leftarrow$ **default MCTS**
          else $\pi_t \leftarrow$ **MCTS Enhancement**
        if $t \leq T'$ **then** a_t = randomly select on π_t
        else $a_t = \arg\max_a(\pi_t)$
        executeAction(s_t, a_t)
      $D \leftarrow (s_t, \pi_t, z_t)$ with outcome $z_{t\in[1,T]}$
      r_{mcts}+= reward of **default MCTS** in this episode
  else ▷ switch
    for episode=1,..., E **do** ▷ purely self-play
      for t=1, ..., T', ..., T **do**
        $\pi_t \leftarrow$ **default MCTS**
        if $t \leq T'$ **then** a_t = randomly select on π_t
        else $a_t = \arg\max_a(\pi_t)$
        executeAction(s_t, a_t)
      $D \leftarrow (s_t, \pi_t, z_t)$ with outcome $z_{t\in[1,T]}$
  Set Switch←True if r_{mcts} >0, and set $r_{mcts} \leftarrow 0$
  Sample minibatch (s_j, π_j, z_j) from D ▷ training
  Train $f_{\theta'} \leftarrow f_\theta$
  $f_\theta = f_{\theta'}$ if $f_{\theta'}$ is better, using **default MCTS** ▷ arena
return f_θ;

6.5 Experimental Setup

Since Chapter 5 only studied the winrate of single rollout and RAVE against a random player, this can be used as a test to check whether rollout and RAVE work. However, it does not reveal any information about relative playing strength, which is necessary to explain how good training examples provided by MCTS enhancements actually are. Therefore, at first we let all 5 enhancements and the baseline MCTS (in this test, the neural network for each player is randomly

initialized) play 100 games with each other on the same 3 games (6×6 Connect Four, Othello and Gobang, game description can be found in Chapter 4) in order to investigate the relative playing strength of each pair.

In the second experiment, we tune the fixed I', where $I' \in \{1, 3, 5, 7, 9\}$, for different search enhancements, based on Algorithm 5 to play 6×6 Connect Four.

In our last experiment, we use new adaptive switch method Algorithm 6 to play 6×6 Othello, Connect Four and Gobang. We set parameters values according to Table 4.1. The parameter choices are based on [33].

Our experiments are run on a high-performance computing (HPC) server, which is a cluster consisting of 20 CPU nodes (40 TFlops) and 10 GPU nodes (40 GPU, 20 TFlops CPU + 536 TFlops GPU). We use small versions of games (6×6) in order to perform a medium number of repetitions. In the following, our figures show error bars of 8 runs, of 100 iterations of self-play. Each single run is deployed in a single GPU which takes several days for different games.

6.6 Results

We list results for a tournament of Baseline and enhancements (Table 6.1). Digging deeper, we also report the effect of the hyper-parameter I' (Fig 6.1). And results for the adaptive warm-start switch are shown in Table 6.2, Fig 6.2 and Fig 6.3.

6.6.1 MCTS vs MCTS Enhancements

Here, we compare the Baseline player (the neural network is initialized randomly which can be regarded as an arena in the first iteration self-play) to the other five MCTS enhancements players on 3 different games. Each pair performs 100 repetitions. In Table 6.1, we can see that for Connect Four, the highest winrate is achieved by WRoRa, the lowest by Rave. Except Rave, others are all higher than 50%, showing that the enhancements (except Rave) are better than the untrained Baseline. In Gobang, it is similar, Rave is the lowest, RoRa is the highest. But the winrates are relatively lower than that in other 2 games. It is interesting that in Othello, all winrates are relatively the highest compared to the 2 other games (nearly 100%), although Rave still achieves the lowest winrate which is higher than 50%.

One reason that enhancements work best in Othello is that the Othello game tree is the longest and narrowest (low branching factor). Enhancements like Rollout

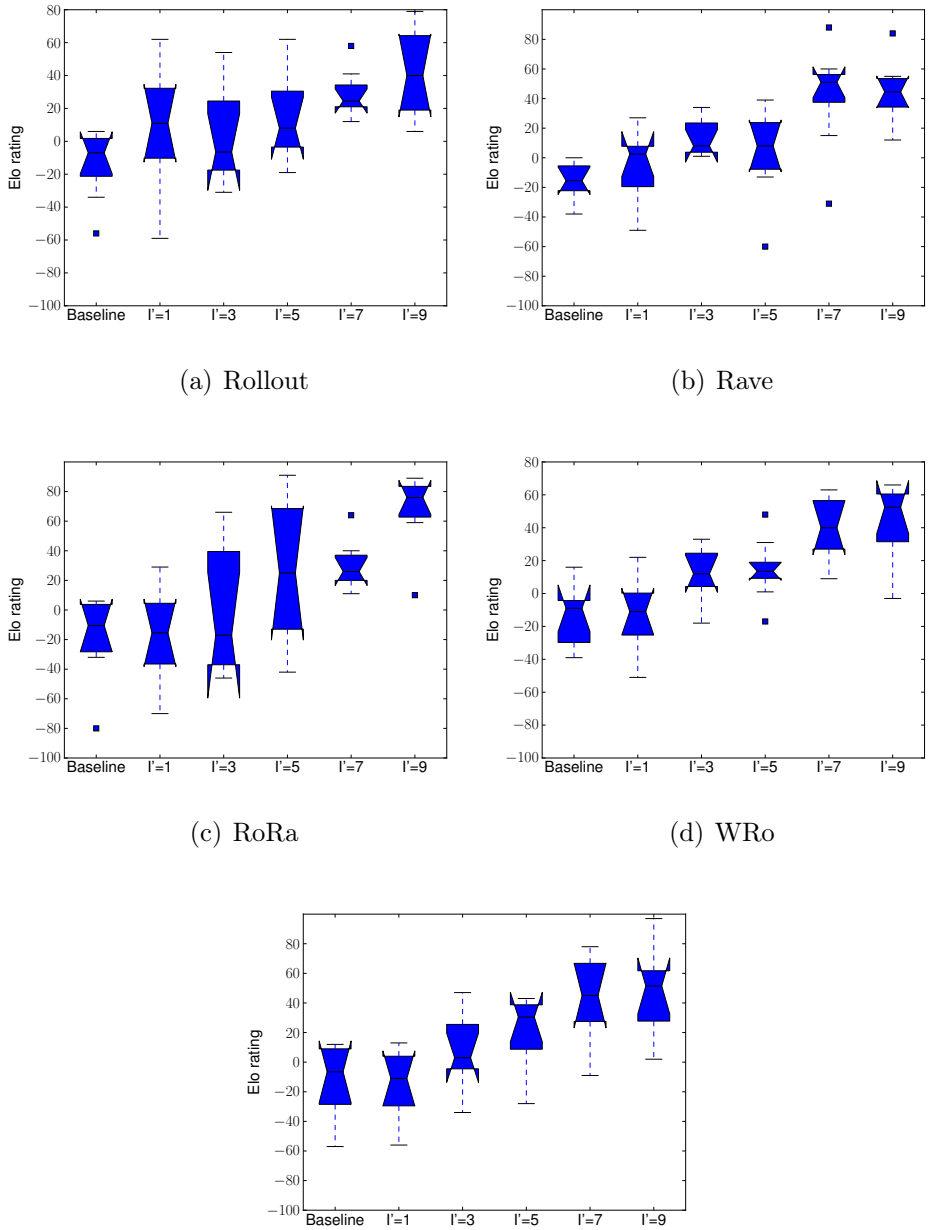
Table 6.1: Results of comparing default MCTS with Rollout, Rave, RoRa, WRo and WRoRa, respectively on the three games with random neural network, weight as 1/2, T'=0, win rates in percent (row vs column), each pair played 100 games.

	Default MCTS		
	ConnectFour	Othello	Gobang
Rollout	64	93	65
Rave	27.5	53	43
RoRa	76	98	70
WRo	82	96	57
WRoRa	82.5	99	62

can provide relatively accurate estimations for these trees. In contrast, Gobang has the shortest game length and the most legal action options. Enhancements like Rollout do not contribute much to the search in short but wide search tree with limited MCTS simulation. As in shorter games it is more likely to reach a terminal state, both Baseline and enhancements get the true result. Therefore, in comparison to MCTS, enhancements like Rollout work better while it does not terminate too fast. Rave is filling more state action pairs based on information from the neural network, its weaknesses at the beginning are more emphasized. After some iterations of training, the neural network becomes smarter, and Rave can therefore enhance the performance as shown in Chapter 5 .

6.6.2 Fixed I' Tuning

Taking Connect Four as an example, in this experiment we search for an optimal fixed I' value, utilizing the warm-start search method proposed in Chapter 5. We set I' as 1, 3, 5, 7, 9 respectively (the value should be relatively small since the enhancement is only expected to be used at the start phase of training). The Elo ratings of each enhancements using different I' are presented in Fig 6.1. The Elo ratings are calculated based on the tournament results using a Bayesian Elo computation system [82], same for Fig 6.3. We can see that for Rave and WRoRa, it turns out that $I' = 7$ is the optimal value for fixed I' warm-start framework, for others, it is still unclear which value is the best, indicating that the tuning is inefficient and costly.



(a) Rollout

(b) Rave

(c) RoRa

(d) WRo

(e) WRoRa

Figure 6.1: Elo ratings for different warm-start phase iterations with different search enhancement on 6×6 Connect Four

6.6.3 Adaptive Warm-Start Switch

In this final experiment, we apply the newly suggested adaptive switch warm-start search enhancement method and compare it to the fixed I'. We are especially interested in the averages and variances of the switching times that result from adaptive switching.

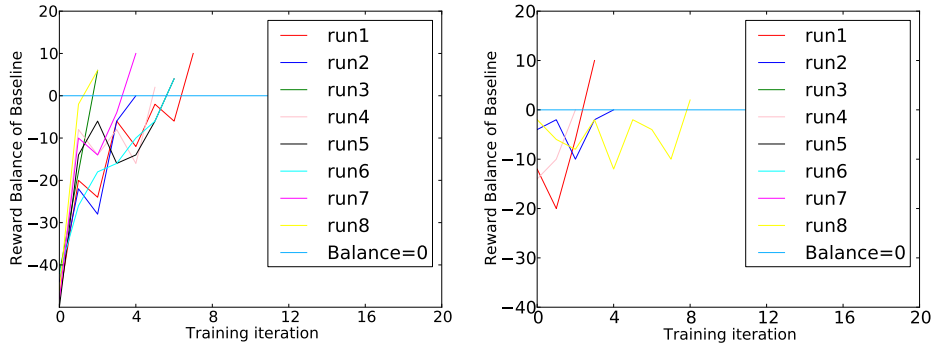
We train models with the parameters in Table 4.1 and then let them compete against each other in different games. In addition, we record the specific iteration number where the switch occurs for every training run and the corresponding self-play arena rewards of MCTS before this iteration. A statistic of the iteration number for 3 games is shown in Table 6.2.

Table 6.2: Switching iterations for training on different games with different enhancements over 8 repetitions (average iteration number ± standard deviation)

	Connect Four	Othello	Gobang
Rollout	6.625± 3.039	5.5± 1.732	1.375±0.484
Rave	2.375±1.218	3.125 ±2.667	1.125±0.331
RoRa	7.75 ±4.74	5.125 ±1.364	1.125±0.331
WRo	4.25±1.561	4.375±1.654	1.125±0.331
WRoRa	4.375±1.576	4.0±1.0	1.25±0.433

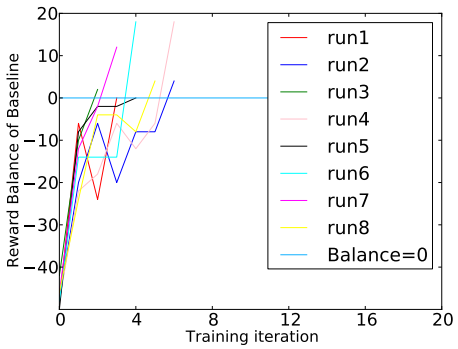
The table shows that, generally, the iteration number is relatively small compared to the total length of the training (100 iterations), and in these small games the neural network is quickly getting stronger. Besides, not only for different games, the switch iteration is different, but also for different training runs on the same game, the switch iteration also varies. This is because for different training runs, the neural network training progresses differently (we already start from different random initializations). Therefore, a fixed I' can not be used for each specific training. Note that for Gobang, a game with a large branching factor, with the default setting, it always switches at the first iteration. Therefore, we also test with larger $m = 200$, thereby providing more time to the MCTS. With this change, there are several runs keeping the enhancements see Table 6.2, but it still shows a small influence on this game.

In addition, we show the arena results (wins of default MCTS minus wins of enhancement) in each training iteration before switch happens in each run over 8 repetitions on Othello as an example in Fig 6.2. In most curves, we can

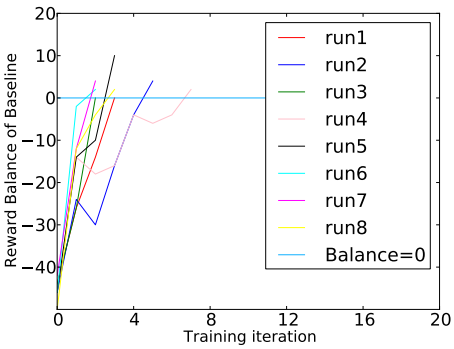


(a) Rollout

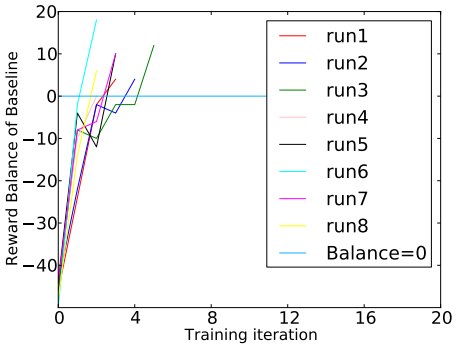
(b) Rave



(c) RoRa

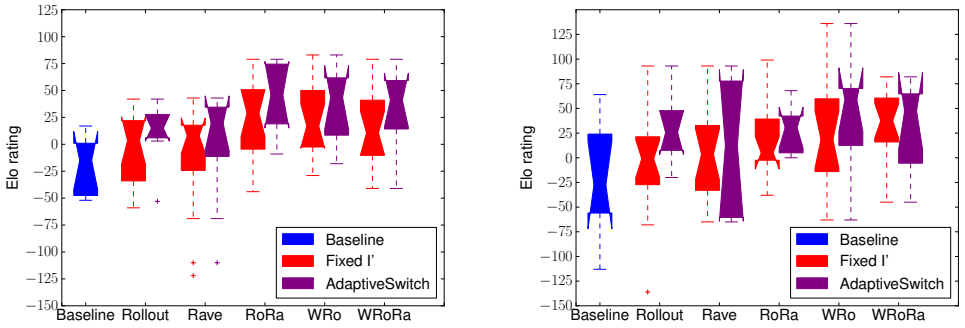


(d) WRo



(e) WRoRa

Figure 6.2: Reward balances of default MCTS while competing with different enhancements in self-play arena for 6×6 Othello. Exceeding 0 means default MCTS defeats the enhancement, switch occurs.



(a) Connect Four

(b) Othello

Figure 6.3: Comparison of adaptive switch method versus fixed I' based on a full tournament for 6×6 Connect Four and Othello

see improving reward balances achieved by default MCTS since it is getting stronger.

More importantly, we collect all trained models based on our adaptive method, and let them compete with the models trained using fixed $I' = 5$ in a full round-robin tournament where each 2 players play 20 games.

From Fig 6.3, we see that, generally, on both Connect Four and Othello, all fixed I' achieve higher Elo ratings than the Baseline, which was also reported in Chapter 5. And all adaptive switch models also perform better than the Baseline. Besides, for each enhancement, it is important that the Elo ratings of the adaptive switch models are higher than for the fixed I' method, which suggests that our adaptive switch method leads to better performance than the fixed I' method when controlling the warm-start iteration length. Specifically, we find that for Connect Four, WRo and RoRa achieve the higher Elo Ratings (see Fig 6.3(a)) and for Othello, WRoRa performs best (see Fig 6.3(b)), which reproduces the consistent conclusion (at least one combination enhancement performs better in different games) as Chapter 5).

In addition, for Connect Four, comparing the tuning results in Fig 6.1 and the *switch iterations* by our method in Table 6.2, we find that our method generally needs a shorter warm-start phase than employing a fixed I'. The reason could be that in our method, there are always 2 different players playing the game, and they provide more diverse training data than a pure self-play player. In consequence, the neural network also improves more quickly, which is highly desired.

Note that while we use the default parameter setting for training in the Gobang game, the *switch* occurs at the first iteration. And even though we enlarge the simulation times for MCTS, only a few training runs shortly keep using the enhancements. We therefore presume that it is meaningless to further perform the tournament comparison for Gobang.

6.7 Summary

Since AlphaGo Zero' results, self-play has become a default approach for generating training data tabula rasa, disregarding other information for training. However, if there is a way to obtain better training examples from the start, why not use them, as has been done recently in StarCraft (see DeepMind's AlphaStar [23]). In addition, Chapter 5 investigated the possibility of utilizing MCTS enhancements to improve AlphaZero-like self-play. They embed Rollout, RAVE and combinations as enhancements at the start period of iterative self-play training and tested this on small board games. Since the neural network and the MCTS statistics are initialized to random weights and zero, self-play suffers from a cold-start problem, and starting from scratch can lead to unstable learning at the start of the training. These problems can be cured by feeding human expert data or running MCTS enhancements or similar methods in order to generate expert data for training the neural network before switching to pure self-play. (Not unlike RAVE warm-starts the winrate statistics of the original MCTS in 2007.)

Confirming Chapter 5, we find that finding an optimal value of fixed I' is difficult, therefore, we propose an adaptive method for deciding when to switch. We also use Rollout, RAVE, and combinations with network values to quickly improve MCTS tree statistics (using RAVE) with meaningful information (using Rollout) before we switch to Baseline-like self-play training. We employed the same games, namely the 6x6 versions of Gobang, Connect Four, and Othello. In these experiments, we find that, for different games, and even different training runs for the same game, the new adaptive method generally switches at different iterations. This indicates the noise in the neural network training progress for different runs. After 100 self-play iterations, we still see the effects of the warm-start enhancements as playing strength has improved in many cases, and for all enhancements, our method performs better than the method proposed in Chapter 5 with I' set to 5. In addition, some conclusions are consistent to Chapter 5, for example, there is also at least one combination that performs better.

The new adaptive method works especially well on Othello and Connect Four, "deep" games with a moderate branching factor, and less well on Gobang, which has a larger branching factor. In the self-play arena, the default MCTS is already quite strong, and for games with a short and wide episode, the MCTS enhancements do not benefit much. Short game lengths reach terminal states early, and MCTS can use the true reward information more often, resulting in a higher chance of winning. Since, Rollout still needs to simulate, with a limited simulation count it is likely to not choose a winning terminal state but a state that has the same average value as the terminal state. In this situation, in a short game episodes, MCTS works better than the enhancement with T'=15. With ongoing training of the neural network, both players become stronger, and as the game length becomes longer, $I' = 5$ works better than the the Baseline.

Our experiments are with small games. Adaptive warm-start works best in deeper games, suggesting a larger benefit for bigger games with deeper lines. Future work includes larger games with deeper lines, and using different but stronger enhancements to generate training examples. Beside, it is also promising to apply the adaptive warm-start idea to master single agent or multi-agent deep reinforcement learning problems.

Chapter 7

Ranked Reward Reinforcement Learning

7.1 Introduction

In previous chapters, we mainly test our approaches on relatively small two-player board games. This chapter will however deal with single-player games and form thereby a bridge to combinatorial optimization.

In recent years, the interest in combinatorial games as a challenge in AI has increased after the first AlphaGo program [16] defeated the human world champion of Go [74]. The great success of the AlphaGo and AlphaZero programs [10, 16, 17] in two-player games, has inspired attempts in other domains [35, 89]. So far, one of the most challenging single player games, Morpion Solitaire [116] has not yet been studied with this promising deep reinforcement learning approach.

Morpion Solitaire is a popular single player game since 1960s [27, 116], because of its simple rules and simple equipment, requiring only paper and pencil. Due to its large state space it is also an interesting AI challenge in single player games, just like the game of Go challenge in two-player turn-based games. Could the AlphaZero self-play approach, that turned out to be so successful in Go, also work in Morpion Solitaire? For ten years little progress has been made in Morpion Solitaire. It is time to take up the challenge and to see if a self-play deep reinforcement learning approach will work in this challenging game as a few works on applying reinforcement learning are studied to deal with combinatorial tasks [117, 118, 119].

AlphaGo and AlphaZero combine deep neural networks [68] and MCTS [7] in a self-play framework that learns by curriculum learning [95]. Unfortunately, these approaches can not be directly used to play single agent combinatorial games, such as Travelling Salesman Problems (TSP) [120] and Bin Packing Problems (BPP) [121], where cost minimization is the goal of the game. To apply self-play for single player games, Laterre et al. proposed a Ranked Reward (R2) algorithm. R2 creates a relative performance metric by means of ranking the rewards obtained by a single agent over multiple games. In two-dimensional and three-dimensional bin packing R2 is reported to out-perform MCTS [122]. In this chapter we use this idea for Morpion Solitaire. Our contributions can be summarized as follows:

1. We present the first implementation[1] of Ranked Reward AlphaZero-style self-play for Morpion Solitaire.
2. On this implementation, we report our current best solution, of 67 steps (see Fig 7.2).

This result is very close to the human record, and shows the potential of the self-play reinforcement learning approach in Morpion Solitaire, and other hard single player combinatorial problems. Our result is even more remarkable, because it has been achieved in a tabula rasa setting, that is starting only with the knowledge [123] about the rules of the game and not encoding strategies and tactics of human players.

This chapter is structured as follows. After giving an overview of related work in Sect. 7.2, we introduce the Morpion Solitaire challenge in Sect. 7.3. Then we present how to integrate the idea of R2 into AlphaZero self-play in Sect. 7.4. Thereafter, we set up the experiment in Sect. 7.5, and show the result and analysis in Sect. 7.6. Finally, we conclude this chapter and discuss future work.

7.2 Related Work

Recent successes of AlphaGo series spark the interest of creating new self-play deep reinforcement learning approaches to deal with problems in the field of game AI, especially for other two player games [30, 31, 33, 64].

However, for single player games, self-play deep reinforcement learning approaches are not yet well studied since the approaches used for two-player games can not directly be used in single player games [122], since the goal of the task changes from

[1]Source code: https://github.com/wh1992v/R2RRMopionSolitaire

winning from an opponent, to minimizing the solution cost. Nevertheless, some researchers did initial works on single games with self-play deep reinforcement learning [124]. The main difficulty is representing single player games in ways that allow the use of a deep reinforcement learning approach. In order to get over this difficulty, Vinyals et al. [125] proposed a neural architecture (Pointer Networks) to represent combinatorial optimization problems as sequence-to-sequence learning problems. Early Pointer Networks achieved decent performance on TSP, but this approach is computationally expensive and requires handcrafted training examples for supervised learning methods. Replacing supervised learning methods by actor-critic methods removed this requirement [126]. In addition, Laterre et al. proposed the R2 algorithm through ranking the rewards obtained by a single agent over multiple games to label win or loss for each search, and this algorithm reportedly outperformed plain MCTS in the bin packing problem (BPP) [122]. Besides, Feng et al. recently used curriculum-driven deep reinforcement learning to cope with hard Sokoban instances [127]. The key of R2 is a progressing (like curriculum learning style) ranked reward list.

In addition to TSP and BPP, Morpion Solitaire has long been a challenge in NP-hard single player problems [27]. In brief, the goal in Morpion Solitaire is to maximally extend a given geometrical structure performing certain legal moves, and given a set of rules. Previous works on Morpion Solitaire mainly employ traditional heuristic search algorithms [116]. Cazenave created Nested Monte-Carlo Search and found an 80 moves record [128]. After that, a new Nested Rollout Policy Adaptation algorithm achieved a new 82 steps record [29]. Thereafter, Cazenave applied Beam Nested Rollout Policy Adaptation [129], which reached the same 82 steps record but did not exceed it, indicating the difficulty of making further progress on Morpion Solitaire using traditional search heuristics.

It is interesting to develop a new approach, applying (self-play) deep reinforcement learning to train a Morpion Solitaire player. The combination of the R2 algorithm with the AlphaZero self-play framework could be a first alternative for above mentioned approaches.

7.3 Morpion Solitaire

Morpion Solitaire is a single player game played on an unlimited grid. The rules of the game are simple. There are 36 black circles as the initial state (see Fig 7.1). A move for Morpion Solitaire consists of two parts: a) placing a new circle on the paper so that this new circle can be connected with four other existing circles horizontally, vertically or diagonally, and then b) drawing a line to connect these

five circles (see action 1, 2, 3 in the figure). Lines are allowed to cross other lines (action 4), but not allowed to overlap. There are two versions: the Touching (5T) version and the Disjoint (5D) version. For the 5T version, it is allowed to touch (action 5, green circle and green line), but for the 5D version, touching is illegal (any circle can not belong to two lines that have the same direction). After a legal action the circle and the line are added to the grid. This chapter focuses on the 5D version.

The best human score for the 5D version is 68 moves [27]. A score of 80 moves was found by means of Nested Monte-Carlo Search [128]. In addition, [29] found a new record with 82 steps, and [129] also found a 82 steps solution. It has been proven mathematically that the 5D version has an upper bound of 121 [28].

7.4 Ranked Reward Reinforcement Learning

AlphaZero self-play achieved milestone successes in two-player games, but can not be directly used for single player cost minimization games. Therefore, the R2 algorithm has been created to use self-play for generic single player MDPs. R2 reshapes the rewards according to player's relative performance over recent games [122]. The pseudo code of R2 is given in Algorithm 7.

Following AlphaZero-like self-play [34], we demonstrate the typical three stages as shown in the pseudo code. Since self-play in Morpion Solitaire MCTS is too time consuming due to the large state space. Thus, we rely on the policy directly from f_θ without tree search (line 6). For stage 3, we directly replace the previous neural network model with the newly trained model and let the newly trained model play a single time with MCTS enhancement (line 15). The R2 idea is integrated (see line 9 to line 11). The reward list B stores the recent game rewards. According to a ratio τ, the threshold of r_τ is calculated. We then compare r_τ to the game reward r_T to reshape the ranked reward z according to Equation 7.1.

$$z = \begin{cases} 1 & r_T > r_\tau \\ -1 & r_T < r_\tau \\ random(1, -1) & r_T = r_\tau \end{cases} \tag{7.1}$$

where r_τ is the stored reward value in B indexed by $L \times \tau$, L is the length of B, τ is a ratio parameter set to control the index of r_τ in B.

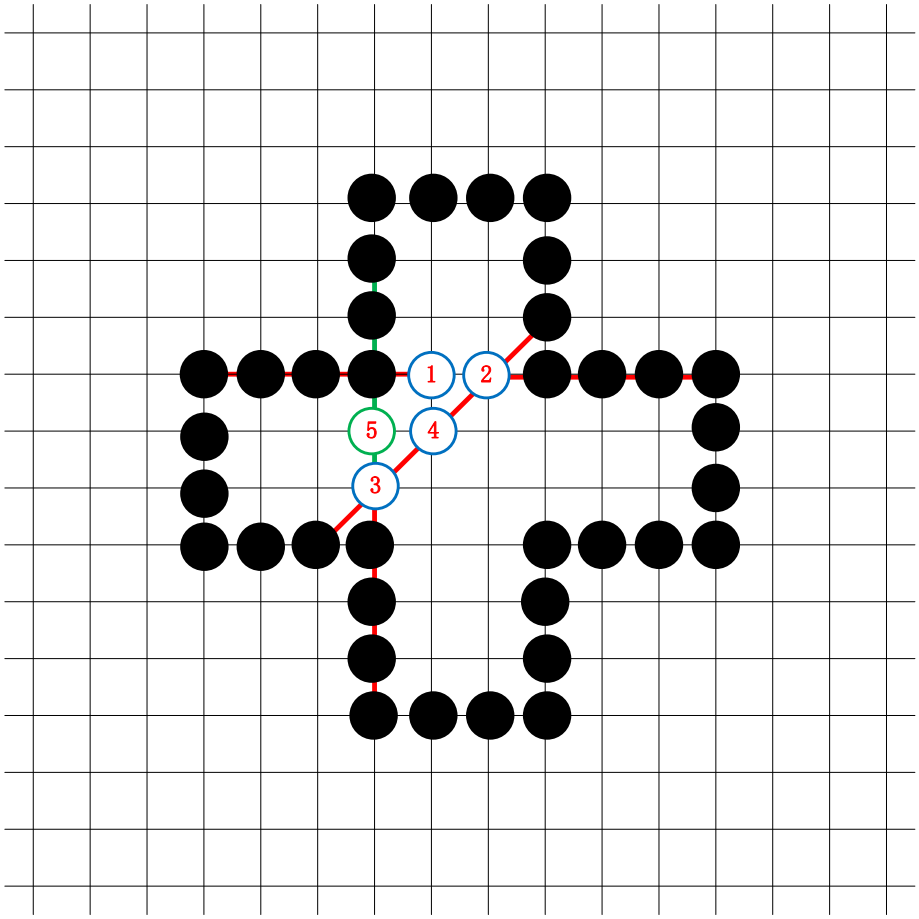


Figure 7.1: Moves Example: Moves 1, 2, 3, 4 are legal moves, move 5 is illegal for the 5D version, but legal for the 5T version. Move 1 is the first move, Move 2 is the second move, and so on.

Algorithm 7 Ranked Reward Reinforcement Learning within AlphaZero-like Self-play Framework

function RANKEDREWARDREINFORCEMENTLEARNING
  Initialize f_θ with random weights; Initialize retrain buffer D and reward list B
  for iteration=1, ..., I **do** ▷ self-play curriculum of I tournaments
    for episode=1,..., E **do** ▷ stage 1, self-play tournament of E games
      for t=1, ..., T', ..., T **do** ▷ play game of T moves
        $\pi_t \leftarrow$ perform MCTS based on f_θ or directly get policy from f_θ
        a_t =randomly select on π_t before T' or $\arg\max_a(\pi_t)$ after T' step
        executeAction(s_t, a_t)
      Calculate game reward r_T and store it in B
      Calculate threshold r_τ based on the recent games rewards in B
      Reshape the ranked reward z following Equation 7.1 ▷ Ranked Reward
      Store every (s_t, π_t, z_t) with ranked rewards z_t $(t \in [1, T])$ in D
    Randomly sample minibatch of examples (s_j, π_j, z_j) from D ▷ stage 2
    Train a new model $f_{\theta'}$ based on f_θ and examples
    Play once with MCTS enhancement on $f_{\theta'}$ ▷ stage 3
    Replace $f_\theta \leftarrow f'_\theta$
  return f_θ;

7.5 Experiment Setup

We perform our experiments on a GPU server with 128G RAM, 3TB local storage, 20 Intel Xeon E5-2650v3 cores (2.30GHz, 40 threads), 2 NVIDIA Titanium GPUs (each with 12GB memory) and 6 NVIDIA GTX 980 Ti GPUs (each with 6GB memory). And the code of framework interface is based on [65].

The hyper-parameters of our current R2 implementation are as much as possible equal to previous work. In this work, all neural network models share the same structure as in [34]. The hyper-parameter values for Algorithm 7 used in our experiments are given in Table 7.1. Partly, these values are set based on the work reported in [32] and the R2 approach for BPP [122]. T' is set to half of the current best record. m is set to 100 if using MCTS in self-play, but 20000 for MCTS in stage 3. Due to that MCTS needs too much computation in our setting, we do not use MCTS to enhance model in self-play (get policy from f_θ directly), but we use MCTS once for every 10 iterations in stage 3. Furthermore, as there is an upper bound of the best score (121), we did experiments on 16×16, 20×20 and 22×22 boards respectively. Training time for every algorithm is about a week.

Table 7.1: Default Parameter Settings

Parameter	Brief Description	Default Value
I	number of iterations	100
E	number of episodes	50
T'	step threshold	41
m	MCTS simulation times	20000
c	weight in UCT	1.0
rs	number of retrain iterations	10
ep	number of epochs	5
bs	batch size	64
lr	learning rate	0.005
d	dropout probability	0.3
L	length of B	200
τ	ratio to compute r_τ	0.75

7.6 Result and Analysis

As we mentioned above, the best score for Morpion Solitaire of 82 steps has been achieved by Nested Rollout Policy Adaptation (NRPA) in 2010. The best score achieved by human is 68. Our first attempt with limited computation resources on a large size board (22×22) achieved a score of 67, very close to the best human score. The resulting solution is shown in Fig 7.2.

Based on these promising results with Ranked Reward Reinforcement Learning we identify areas for further improvement. First, parameter values for the Morpion Solitaire game can be fine-tuned using results of small board games. Especially the parameter $m = 100$ seems not sufficient for large boards. Second, the neural network could be changed to Pointer Networks and the size of neural network should be deeper.

Note that the tuning of parameters is critical; if the reward list B is too small, the reward list can be easily filled up by scores close to 67. The training will then be stuck in a locally optimal solution. As good solutions are expected to be sparsely distributed over the search space, this increases the difficulty to get rid of a locally optimal solution once the algorithm has focused on it.

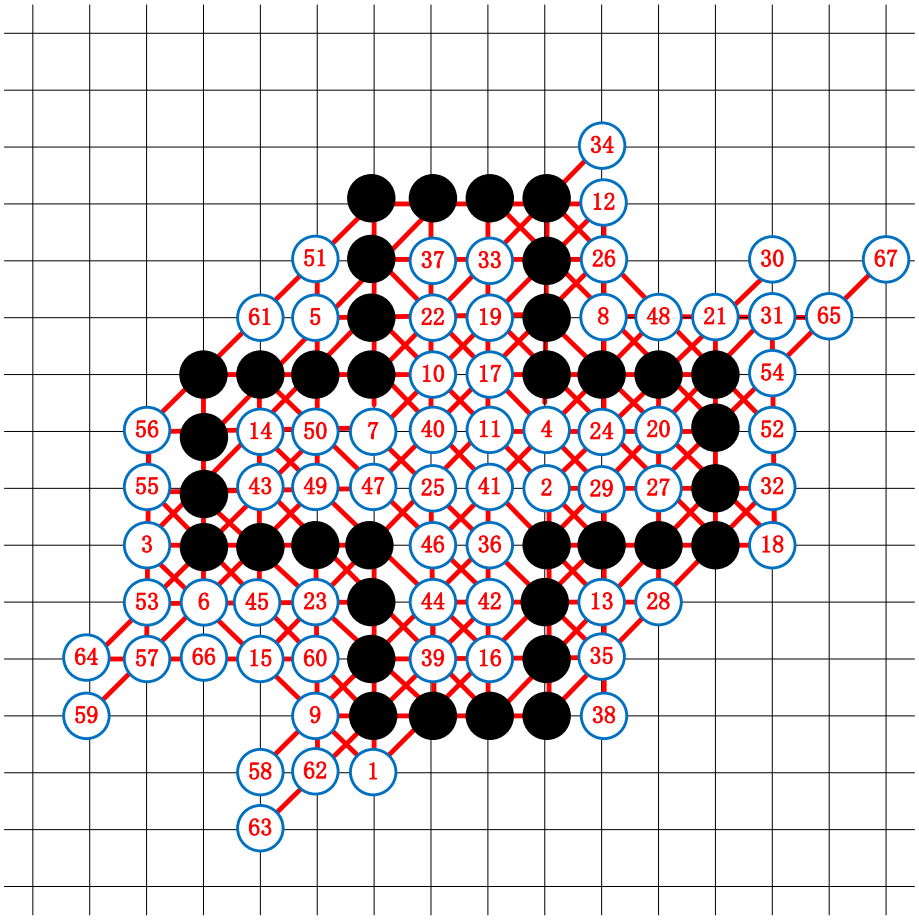


Figure 7.2: Detailed Steps of Our Best Solution

7.7 Summary

In this work, we apply a Ranked Reward Reinforcement Learning AlphaZero-like approach to play Morpion Solitaire, an important NP-hard single player game challenge. We train the player on 16×16, 20×20 and 22×22 boards, and find a near best human performance solution with 67 steps. As a first attempt of utilizing self-play deep reinforcement learning approach to tackle Morpion Solitaire, achieving near-human performance is a promising result.

To summarize, although the problem is difficult due to its large state space and sparsity of good solutions, applying a Ranked Reward self-play Reinforcement Learning approach to tackle Morpion Solitaire is a promising and learns from *tabula rasa*. We present our promising near-human result to stimulate future work on Morpion Solitaire and other single agent games with self-play reinforcement learning.

Chapter 8

Conclusion

This work relies on a framework, as used by AlphaZero, that combines online searching and offline learning. This framework has become an effective approach in deep reinforcement learning since AlphaGo series algorithms achieve super human level performance on playing complex games. Within this framework, the offline learning model provides state values to guide MCTS search, and the neural network is trained by the self-play game records played by MCTS search results.

Before deep neural networks were common in reinforcement learning (due to the limits of hardware computation capacity), table based approaches of Q-learning were used, and MCTS, as online search methods to play small versions of Go. A combination of online search (MCS) and offline learning (table based Q-learning) in GGP was assessed. MCS was used to generate expert data for self-play at the beginning phase for the table based Q-learning. The results show that table based Q-learning converges in GGP and has potential to be improved by MCS techniques. Inspired by our work in Chapter 2, [60] establish their deep reinforcement learning GGP system.

Therefore, in this dissertation, an AlphaZero-like self-play framework was studied to further investigate the combination of online search and offline learning in a deep reinforcement learning context. A detailed analysis of 12 hyper-parameters was provided. Among these hyper-parameters, four interesting hyper-parameters are analyzed further, and several interesting correlations are presented in Chapter 3. Then the alternative loss functions were evaluated to see how value loss and policy loss contribute to training in Chapter 4.

AlphaZero-like self-play initializes its neural network randomly, and therefore suffers from a cold-start problem, just as table based Q-learning, which is initialized as empty. Applying MCTS enhancements to generate expert data at the start phase of training achieves better performance. This methods is called the warm-start method (Chapter 5). Furthermore, an adaptive warm-start method was proposed to control the necessary iteration length, which is more robust to different enhancements and different training runs (Chapter 6).

In Chapter 7, the ranked reward method was combined with AlphaZero-like self-play to tackle a complex single agent combinatorial game, Morpion Solitaire, and achieved a near human level grid. This chapter highlights the potential of the AlphaZero framework to provide competitive results in combinatorial searching games starting from tabula rasa setting.

Next, the main contributions of this dissertation will be presented in Sect 8.1 and directions for future work will be discussed in Sect 8.2.

8.1 Contributions

This dissertation mainly focus on applying searching and learning methods of reinforcement learning in GGP and AlphaZero-like self-play framework. The main contributions can be summarized as follows.

Classical Q-learning can be used to play GGP games, although training is slow. This finding provides a basis for applying deep neural networks to GGP. The MCS enhancement generates better training examples for Q-learning at the start phase of training, which also reveals a promising direction for deep reinforcement learning approaches like AlphaZero-like self-play to be further improved.

For AlphaZero-like self-play, balancing the number of outer iteration loop and the inner epochs training is a key point to generalize more efficient training examples and to avoid useless and redundant training. Since the overall epochs number for training is influenced by the outer iteration, the result is that the neural network is trained by the same training examples for too many epochs, if the epoch number is too big. Similarly, it is found that the MCTS search is costly, but that the improvements brought by more MCTS simulation time can be compensated by a better trained model, which also requires proper balancing method.

The policy loss and value loss functions contribute differently to different games in an AlphaZero-like self-play framework, but a sum of these two losses is a reasonable compromise choice. The neural network model of AlphaZero-like self-play has two heads, i.e. policy head and value head. This poses the question whether policy or value loss is necessary for training, and how they contribute to the training. Again, this confirms that the sum could be a compromise choice, but not necessary the best choice.

In both general games frameworks (GGP and AlphaZero-like self-play), employing search enhancements at the start of training improves final training results. For self-play, learning from scratch is not the default best choice. Existing available human expert data and expert data from AI programs can be used to improve training, especially at the start phase of training. Since the model is initialized randomly, the self-play agent based on such models performs nearly randomly which results in bad training examples. Therefore, search enhancements can be used to generate better examples until the model is improved enough to outperform the search enhancements. Our findings on both frameworks suggest boosting the training at start phase of training by search enhancements like MCS and MCTS with RAVE works better.

The structure of combining online search and offline training can be improved by MCTS enhancements. From table based Q-learning to neural network based reinforcement learning, search and learning are both important. It is also found that, just like MCTS with RAVE improves table based learning in [18], MCTS with RAVE can also be used to improve the performance of neural network based reinforcement learning, see Chapter 5 and 6.

AlphaZero-like self-play can be transferred to solve a complex single agent combinatorial game when assisted by other techniques like ranked reward. Since it is not possible to directly determine a win or loss for single player combinatorial games like Morpion Solitaire, methods like ranked reward can be used to set sub-goals for such sparse reward-long episode scheduling problems. It is found that AlphaZero-like self-play can be used to also solve complex combinatorial games. The result with Morpion Solitaire indicates a promising future for this approach.

8.2 Outlook

The research reported in this thesis has yielded many interesting results. We now enlist some promising avenues for further research.

Inspired by using table based Q-learning in GGP, Goldwasser et al. [60] have build a deep reinforcement learning framework, showing that the deep neural network can easily be embedded into it. The next step could be applying heuristic search enhancements to improve such deep reinforcement learning framework.

Other possible studies on warm-start enhancements of AlphaZero-like self-play have not been conducted yet. Thus, a number of interesting problems remain to be investigated.

- ★ Which enhancements will work best on which games? Does the hypothesis hold that games with more consistent opening plans benefit more from the warm-start?
- ★ How should the weight (weight w is the parameter which is used to combine the different enhancements search results) be changed along with the training iteration progress? Linearly or non-linearly? In our experiments it simply decays linearly.
- ★ There are more parameters that are critical and that could not really be explored yet due to computational cost, but this exploration may reveal important performance gains. For example, the MCTS simulation count (m) and the step threshold (T').
- ★ Other warm-start enhancements, e.g., built on variants of RHEA's or hybrids of it, can be explored.
- ★ All our current test cases are relatively small games. How do the results transfer to larger games, or to different applications?

For single agent problems, our first results on Morpion Solitaire give us reason to believe that there remain ample possibilities to improve the approach by investigating the following aspects:

- ★ Parameter Tuning: such as the Monte Carlo simulation times. Since good solutions are sparse in this game, maybe more exploration is beneficial?
- ★ Neural Network Design: It is reported that Pointer Networks perform better on combinatorial problems [125]. A next step could be to also make the neural network structure deeper.

- ★ Local Optima: By monitoring the reward list B, it can be enlarged to allow more exploration, once it gets stuck in a locally optimal solution.
- ★ By adding more computational resources and parallelization results can be enhanced.

In addition, it is also interesting to further study the importance of searching and learning in AlphaZero-like self-play, as it is still unclear if searching (like MCTS) is necessary for the last part of a long term training. Searching is quite expensive and normally the last part of training does not give too much improvement.

To further explore the AGI in deep reinforcement learning, curriculum learning [95], meta learning [130] and transfer learning techniques [46, 54] should be also combined to deal with more general tasks.

Besides, in our AlphaZero-like self-play experiments, we suffer from the shortage of computation resources. In the future, we can technically applying parallelization programming for the self-play phase of AlphaZero. And optimizing the neural network structure is also useful to speed up the training.

In consequence, this thesis should not only provide new methods and results but also encourage researchers to help explore these approaches and questions in future investigations.

Appendix A

A.1 Symbols

Table A.1: Notations

-	Type	Description	Ref.
γ	$0 \leq \gamma \leq 1$	the discount factor of $max_{a'}Q(s', a')$	Eq. (2.1)
α	$0 \leq \alpha \leq 1$	the learning rate of Q-learning	Eq. (2.2)
ϵ	$0 \leq \epsilon \leq 1$	ϵ-greedy for exploration and exploitation	Eq. (2.3)
l	$\mathbb{N}^+$	match number used to control decaying speed of ϵ	Eq. (2.3)
d	$\mathbb{N}^+$	dimension of action space	Eq: (3.1)
$\mathbf{p}$	$\mathbb{R}^d$	policy provided by the neural network	Eq: (3.1)
π	$\mathbb{R}^d$	improved estimate policy after performing MCTS	Eq: (3.1)
v	$\mathbb{R}$	state value prediction	Eq: (3.1)
z	$\{-1, 0, 1\}$	real game end reward	Eq: (3.1)
λ	$0 \leq \lambda \leq 1$	a weight to balance policy and value loss function	Eq: (4.1)
β	$\mathbb{R}$	a weight number to balance $U(s, a)$ and $U_{rave}(s, a)$	Eq: (5.4)
L	$\mathbb{N}^+$	the length of the reward list	Eq: (7.1)
τ	$0 \leq \tau \leq 1$	a ratio to locate game length threshold in reward list	Eq: (7.1)
r_τ	$\mathbb{N}^+$	threshold reward of game length to judge win or loss	Eq: (7.1)

A.2 Abbreviations

Table A.2: Abbreviations

Abb.	Full Name
AI	Artificial Intelligence
AGI	Artificial General Intelligence
MCS	Monte Carlo Search
MCTS	Monte Carlo Tree Search
GGP	General game playing
RAVE	Rapid Action Value Estimation
GDL	Game Description Language
DQN	Deep Q-networks
GM	Game Manager
TCP/IP	Transmission Control Protocol/Internet Protocol
UCT	Upper Confidence bound applied to Trees
AMAF	All Moves As First
RHEA	Rolling Horizon Evolutionary Algorithm
P-UCT	Policy-Upper Confidence bound applied to Trees
HPC	High Performance Computing
TSP	Travelling Salesman Problems
BPP	Bin Packing Problems
R2	Ranked Reward
MDP	Markov Decision Process
NRPA	Nested Rollout Policy Adaptation

A.3 Algorithms

Algorithm 8 Time Limited Monte Carlo Search Algorithm

```
1: function MonteCarloSearch(time_limit)
2:     get legal actions set A of current state s
3:     get next states set S' where s' ∈ S'
4:     z(s')=0, count(s')=0
5:     while time_cost ≤ time_limit do
6:         for each s' in S' do
7:             outcome(s')←random simulation from s' to game end.
8:             z(s')+=outcome(s')
9:             count(s')+=1
10:        selected_action← getActionFromStates(s, arg max_{s'∈S'} z(S')/count(S'))
11:    return selected_action
```

Algorithm 9 QM-learning Enhancement

function QMPLAYER(current state s, learning rate α, discount factor γ, Q table: $Q(S, A)$)
    for each match **do**
        if s terminates **then**
            for each (s, a) from end to the start in current match record **do**
                R(s,a) =s' is terminal state? getGoal(s', myrole) : 0
                Update $Q(s, a) \leftarrow (1-\alpha)\ Q(s, a)+\alpha\ (\ R(s, a)+\gamma\ max_{a'} Q(s', a'))$
        else
            if ϵ-greedy is enabled **then**
                selected_action = Random()
            else
                selected_action = SelectFromQTable()
                if no s record in $Q(S, A)$ **then**
                    MonteCarloSearch(time_ limit)
            performAction(s, selected_action)
    return $Q(S, A)$

Algorithm 10 Neural Network Based MCTS with Only Rollout Simulation Value

function ROLLOUT(s, f_θ)
    Search(s)
    $\pi_s \leftarrow$normalize($Q(s, \cdot)$)
    return π_s
function SEARCH(s)
    Return game end result if s is a terminal state
    if s is not in the Tree **then**
        Add s to the Tree, initialize $Q(s, \cdot)$ and $N(s, \cdot)$ to 0
        Get $P(s, \cdot)$ and $v(s)$ by looking up $f_\theta(s)$
        Get result $v(s)$ by performing random rollout until the game ends
        return $v(s)$
    else
        Select an action a with highest UCT value
        $s' \leftarrow$getNextState(s, a)
        $v \leftarrow$Search($s\prime$)
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
    return v;

Algorithm 11 Neural Network Based MCTS with Only RAVE Value

function RAVE(s, f_θ)
    Search(s)
    $\pi_s \leftarrow$normalize($Q_{rave}(s, \cdot)$)
    return π_s
function SEARCH(s)
    Return game end result if s is a terminal state
    if s is not in the Tree **then**
        Add s to the Tree,
        Initialize $Q(s, \cdot)$, $N(s, \cdot)$, $Q_{rave}(s, \cdot)$ and $N_{rave}(s, \cdot)$ to 0.
        Get $P(s, \cdot)$ and $v(s)$ by looking up $f_\theta(s)$
        return $v(s)$
    else
        Select an aciton a with highest UCT_{rave} value
        $s\prime \leftarrow$getNextState(s, a)
        $v \leftarrow$Search($s\prime$)
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
        $N_{rave}(s_{t_1}, a_{t_2}) \leftarrow N_{rave}(s_{t_1}, a_{t_2}) + 1$
        $Q_{rave}(s_{t_1}, a_{t_2}) \leftarrow \frac{N_{rave}(s_{t_1},a_{t_2})*Q_{rave}(s_{t_1},a_{t_2})+v}{N_{rave}(s_{t_1},a_{t_2})+1}$
        $\triangleright$ where $s_{t_1} \in VisitedPath$, and $a_{t_2} \in A(s_{t_1})$, and for $\forall t < t_2, a_t \neq a_{t_2}$
    return v;

Algorithm 12 Neural Network Based MCTS with Rollout Simulation and RAVE Value

function RoRa(s, f_θ)
    Search(s)
    $\pi_s \leftarrow$normalize($Q_{rave}(s, \cdot)$)
    return π_s
function Search(s)
    Return game end result if s is a terminal state
    if s is not in the Tree **then**
        Add s to the Tree,
        Initialize $Q(s, \cdot)$, $N(s, \cdot)$, $Q_{rave}(s, \cdot)$ and $N_{rave}(s, \cdot)$ to 0.
        Get $P(s, \cdot)$ and $v(s)$ by looking up $f_\theta(s)$
        Get result $v(s)$ by performing random rollout until the game ends
        return $v(s)$
    else
        Select an aciton a with highest UCT_{rave} value
        $s\prime \leftarrow$getNextState(s, a)
        $v \leftarrow$Search($s\prime$)
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
        $N_{rave}(s_{t_1}, a_{t_2}) \leftarrow N_{rave}(s_{t_1}, a_{t_2}) + 1$
        $Q_{rave}(s_{t_1}, a_{t_2}) \leftarrow \frac{N_{rave}(s_{t_1},a_{t_2})*Q_{rave}(s_{t_1},a_{t_2})+v}{N_{rave}(s_{t_1},a_{t_2})+1}$
        ▷ where $s_{t_1} \in VisitedPath$, and $a_{t_2} \in A(s_{t_1})$, and for $\forall t < t_2, a_t \neq a_{t_2}$
    return v;

Algorithm 13 Neural Network Based MCTS with Neural Network and Rollout Simulation Value

function WRO(s, f_θ)
    Search(s)
    $\pi_s \leftarrow$normalize($Q(s, \cdot)$)
    return π_s
function SEARCH(s)
    Return game end result if s is a terminal state
    if s is not in the Tree **then**
        Add s to the Tree, initialize $Q(s, \cdot)$ and $N(s, \cdot)$ to 0
        Get $P(s, \cdot)$ and $v(s)_{network}$ by looking up $f_\theta(s)$
        Get result $v(s)_{rollout}$ by performing random rollout until the game ends
        $v(s) = (1 - weight) * v_{network} + weight * v_{rollout}$
        return $v(s)$
    else
        Select an action a with highest UCT value
        $s' \leftarrow$getNextState(s, a)
        $v \leftarrow$Search($s\prime$)
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
    return v;

Algorithm 14 Neural Network Based MCTS with Neural Network, Rave and Rollout Simulation Value

function WRoRa(s, f_θ)
    Search(s)
    $\pi_s \leftarrow$normalize($Q(s, \cdot)$)
    return π_s
function Search(s)
    Return game end result if s is a terminal state
    if s is not in the Tree **then**
        Add s to the Tree,
        Initialize $Q(s, \cdot)$, $N(s, \cdot)$, $Q_{rave}(s, \cdot)$ and $N_{rave}(s, \cdot)$ to 0.
        Get $P(s, \cdot)$ and $v(s)_{network}$ by looking up $f_\theta(s)$
        Get result $v(s)_{rollout}$ by performing random rollout until the game ends
        random rollout path added to $VisitedPath$
        $v(s) = (1 - weight) * v_{network} + weight * v_{rollout}$
        return $v(s)$
    else
        Select an aciton a with highest UCT_{rave} value
        $s' \leftarrow$getNextState(s, a)
        $v \leftarrow$Search($s\prime$)
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
        $N_{rave}(s_{t_1}, a_{t_2}) \leftarrow N_{rave}(s_{t_1}, a_{t_2}) + 1$
        $Q_{rave}(s_{t_1}, a_{t_2}) \leftarrow \frac{N_{rave}(s_{t_1},a_{t_2})*Q_{rave}(s_{t_1},a_{t_2})+v}{N_{rave}(s_{t_1},a_{t_2})+1}$
        ▷ where $s_{t_1} \in VisitedPath$, and $a_{t_2} \in A(s_{t_1})$, and for $\forall t < t_2, a_t \neq a_{t_2}$
    return v;

Algorithm 15 Rolling Horizon Evolutionary Algorithm

1: **function** RHEA(s, $time\_limit$)
2: Set up population of n valid action sequences of length l: $A_{n,l}$
3: **for all** $A_{i<n}$ **do** Evaluate(A_i)
4: **repeat**
5: new action sequence A_j = mutate one randomly chosen action sequence by changing every move with a small random chance
6: $f(A_j)$ = Evaluate(A_J)
7: add A_J to population
8: remove A_i with worst $f(A_i)$ from population
9: **until** $time\_cost \geq time\_limit$
10: **return** first action of best sequence in population
11: **function** EVALUATE(A_i)
12: **repeat**
13: Play action sequence in A_i
14: Get result $success, game\_steps$ by performing random rollout until the game ends
15: **until** $repetitions \geq 2$
16: compute fitness $f(A_i)$ from average success probability with sequence length penalty (line 117)
17: **return** $f(A_i)$

A.4 Elo Computation

In this dissertation, like AlphaZero series papers did, a whole history Bayesian Elo computation [82] is also employed to present the relative competence of playing the game of different trained models instead of a win or loss rate. In this section, a full computation process will be described in detail based on the Bayesian Elo computation system (called Bayeselo) provided on github [131].

Bayeselo is a free software tool to compute Elo ratings. It receives a file containing game records written in PGN (Portable Game Notation) format [132], and produces a rating list [131]. Therefore, a full process can be simply described in Fig. A.1.

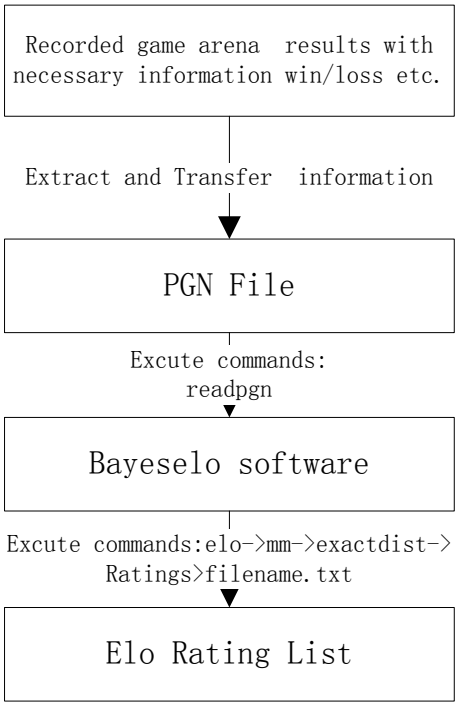


Figure A.1: a full Bayesian Elo Computation Process.

An example of part of a PGN file (arena_othello_final.pgn) generated based on win/loss results recorded during AlphaZero-like self-play arena competition is shown as Fig. A.2.

```
.....
[Event "arena_othello_final.pgn"]
[Iteration "926"]
[Site "liacs server, Leiden"]
[Round "6"]
[White "bestmodel_mcts_rave_run2"]
[Black "bestmodel_mcts_rave_run5"]
[Result "1-0"]
Here are detailed game moves for [Iteration "926, round6"]

[Event "arena_othello_final.pgn"]
[Iteration "926"]
[Site "liacs server, Leiden"]
[Round "7"]
[White "bestmodel_mcts_rave_run2"]
[Black "bestmodel_mcts_rave_run5"]
[Result "0-1"]
Here are detailed game moves for [Iteration "926, round7"]

[Event "arena_othello_final.pgn"]
[Iteration "926"]
[Site "liacs server, Leiden"]
[Round "8"]
[White "bestmodel_mcts_rave_run2"]
[Black "bestmodel_mcts_rave_run5"]
[Result "0-1"]
Here are detailed game moves for [Iteration "926, round8"]

[Event "arena_othello_final.pgn"]
[Iteration "926"]
[Site "liacs server, Leiden"]
[Round "9"]
.....
```

Figure A.2: Small Part of PGN file arena_othello_final.pgn. The file contains much such format iterative arena competition information. Each pair of White and Black players played 20 rounds.

An example of elo rating list generated by operating Bayeselo system with PGN file (arena_othello_final.pgn) as input is shown as follows. See Fig. A.3. The figures of elo ratings in this dissertation are visualized based on such elo rating lists.

Table A.3: An example of Generated Elo Rating List by Bayeselo

Rank	Name	Elo	+	-	games	score	oppo.	draws
1	bestmodel_mcts_rave_rollout_run6	117	22	22	960	62%	-2	0%
2	bestmodel_weight_mcts_rave_rollout_run5	102	21	21	960	57%	-2	0%
3	bestmodel_weight_mcts_rave_rollout_run2	100	21	21	960	58%	-2	0%
4	bestmodel_weight_mcts_rollout_run5	97	22	21	960	58%	-2	0%
5	bestmodel_mcts_rave_run1	86	21	21	960	58%	-2	0%
6	bestmodel_weight_mcts_rave_rollout_run8	81	21	21	960	55%	-2	0%
7	bestmoldel_pi_v_run1	77	22	21	960	61%	-2	0%
8	bestmodel_weight_mcts_rave_rollout_run3	71	21	21	960	54%	-1	0%
9	bestmodel_mcts_rave_rollout_run2	62	21	21	960	57%	-1	0%
10	bestmodel_weight_mcts_rave_rollout_run1	54	21	21	960	53%	-1	0%
11	bestmodel_weight_mcts_rave_rollout_run6	53	21	21	960	52%	-1	0%
12	bestmodel_mcts_rave_run4	53	21	21	960	54%	-1	0%
13	bestmodel_weight_mcts_rave_rollout_run4	52	21	21	960	52%	-1	0%
14	bestmodel_weight_mcts_rollout_run3	44	21	21	960	52%	-1	0%
15	bestmodel_weight_mcts_rollout_run8	39	21	21	960	51%	-1	0%
16	bestmoldel_pi_v_run4	39	21	21	960	56%	-1	0%
17	bestmodel_weight_mcts_rave_rollout_run7	36	21	21	960	50%	-1	0%
18	bestmodel_weight_mcts_rollout_run7	34	21	21	960	51%	-1	0%
19	bestmodel_mcts_rave_run7	32	21	21	960	51%	-1	0%
20	bestmodel_weight_mcts_rollout_run2	30	21	21	960	51%	-1	0%
21	bestmodel_mcts_rave_rollout_run5	28	21	21	960	52%	-1	0%
22	bestmodel_mcts_rave_run2	21	21	21	960	51%	0	0%
23	bestmodel_weight_mcts_rollout_run4	20	21	21	960	49%	0	0%
24	bestmodel_weight_mcts_rollout_run1	20	21	21	960	50%	0	0%
25	bestmoldel_pi_v_run6	18	21	21	960	53%	0	0%
26	bestmodel_mcts_rollout_run3	17	21	21	960	53%	0	0%
27	bestmodel_mcts_rave_rollout_run7	15	21	21	960	51%	0	0%
28	bestmodel_mcts_rollout_run1	11	21	21	960	52%	0	0%
29	bestmodel_mcts_rave_run8	10	21	21	960	49%	0	0%
30	bestmodel_weight_mcts_rollout_run6	8	21	21	960	48%	0	0%
31	bestmodel_mcts_rollout_run2	7	21	21	960	52%	0	0%
32	bestmodel_mcts_rave_rollout_run8	6	21	21	960	49%	0	0%
33	bestmodel_mcts_rave_run5	5	21	21	960	49%	0	0%
34	bestmodel_mcts_rave_run6	4	21	21	960	48%	0	0%
35	bestmodel_mcts_rave_rollout_run3	2	21	21	960	50%	0	0%
36	bestmoldel_pi_v_run7	-2	21	21	960	51%	0	0%
37	bestmodel_mcts_rollout_run6	-6	21	21	960	49%	0	0%
38	bestmoldel_pi_v_run3	-7	21	21	960	51%	0	0%
39	bestmodel_mcts_rollout_run5	-8	21	21	960	49%	0	0%
40	bestmodel_mcts_rave_rollout_run4	-10	21	21	960	48%	0	0%
41	bestmodel_mcts_rollout_run7	-11	21	21	960	49%	0	0%
42	bestmodel_mcts_rave_rollout_run1	-17	21	21	960	48%	0	0%
43	bestmoldel_pi_v_run8	-17	21	21	960	49%	0	0%
44	bestmodel_mcts_rollout_run8	-43	21	21	960	45%	1	0%
45	bestmoldel_pi_v_run5	-65	21	21	960	44%	1	0%
46	bestmodel_mcts_rave_run3	-66	21	22	960	41%	1	0%
47	bestmoldel_pi_v_run2	-100	21	22	960	41%	2	0%
48	bestmodel_mcts_rollout_run4	-109	22	22	960	38%	2	0%
49	randomplayer	-988	124	204	960	0%	21	0%

APPENDIX

Bibliography

[1] Crevier, D.: AI: the tumultuous history of the search for artificial intelligence. Basic Books, Inc. (1993)

[2] Kurzweil, R.: The singularity is near: When humans transcend biology. Penguin (2005)

[3] Searle, J.R., et al.: Minds, brains, and programs. The Turing Test: Verbal Behaviour as the Hallmark of Intelligence (1980) 201–224

[4] Hodson, H.: Deepmind and Google: the battle to control artificial intelligence. The Economist (2019)

[5] Grace, K., Salvatier, J., Dafoe, A., Zhang, B., Evans, O.: When will AI exceed human performance? Evidence from AI experts. Journal of Artificial Intelligence Research **62** (2018) 729–754

[6] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press (2018)

[7] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo Tree Search methods. IEEE Transactions on Computational Intelligence and AI in Games **4** (2012) 1–43

[8] Fagin, R., Moses, Y., Halpern, J.Y., Vardi, M.Y.: Reasoning about knowledge. MIT press (2003)

[9] Bishop, C.M.: Pattern recognition. Machine learning **128** (2006)

[10] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. Science **362** (2018) 1140–1144

[11] Ward, C.D., Cowling, P.I.: Monte Carlo Search applied to card selection in magic: The gathering. In: 2009 IEEE Symposium on Computational Intelligence and Games, IEEE (2009) 9–16

[12] Winands, M.H., Björnsson, Y., Saito, J.T.: Monte-Carlo Tree Search solver. In: International Conference on Computers and Games, Springer (2008) 25–36

[13] Watkins, C.J., Dayan, P.: Q-learning. Machine Learning **8** (1992) 279–292

[14] Fan, J., Wang, Z., Xie, Y., Yang, Z.: A theoretical analysis of deep Q-learning. In: Learning for Dynamics and Control, PMLR (2020) 486–489

[15] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., et al.: Deep Q-learning from demonstrations. In: Proceedings of the AAAI Conference on Artificial Intelligence. Volume 32. (2018)

[16] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529** (2016) 484

[17] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. Nature **550** (2017) 354

[18] Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th International Conference on Machine Learning. (2007) 273–280

[19] Finnsson, H.: Generalized Monte-Carlo Tree Search extensions for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence. Volume 26. (2012)

[20] Sironi, C.F., Winands, M.H.: On-line parameter tuning for Monte-Carlo Tree Search in general game playing. In: Workshop on Computer Games, Springer (2017) 75–95

[21] Sironi, C.F., Liu, J., Winands, M.H.: Self-adaptive Monte Carlo Tree Search in general game playing. IEEE Transactions on Games **12** (2018) 132–144

[22] Méhat, J., Cazenave, T.: A parallel general game player. KI-künstliche Intelligenz **25** (2011) 43–47

[23] Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature **575** (2019) 350–354

[24] Gelly, S., Silver, D.: Monte-Carlo Tree Search and rapid action value estimation in computer Go. Artificial Intelligence **175** (2011) 1856–1875

[25] Sironi, C.F., Winands, M.H.: Comparison of rapid action value estimation variants for general game playing. In: 2016 IEEE Conference on Computational Intelligence and Games (CIG), IEEE (2016) 1–8

[26] Couëtoux, A., Milone, M., Brendel, M., Doghmen, H., Sebag, M., Teytaud, O.: Continuous rapid action value estimates. In: Asian Conference on Machine Learning, PMLR (2011) 19–31

[27] Demaine, E.D., Demaine, M.L., Langerman, A., Langerman, S.: Morpion Solitaire. Theory of Computing Systems **39** (2006) 439–453

[28] Kawamura, A., Okamoto, T., Tatsu, Y., Uno, Y., Yamato, M.: Morpion Solitaire 5D: a new upper bound of 121 on the maximum score. arXiv preprint arXiv:1307.8192 (2013)

[29] Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: Twenty-Second International Joint Conference on Artificial Intelligence. (2011) 649–654

[30] Wang, H., Emmerich, M., Plaat, A.: Monte Carlo Q-learning for general game playing. arXiv preprint arXiv:1802.05944 (2018)

[31] Wang, H., Emmerich, M., Plaat, A.: Assessing the potential of classical Q-learning in general game playing. In: Benelux Conference on Artificial Intelligence, Springer (2018) 138–150

[32] Wang, H., Emmerich, M., Preuss, M., Plaat, A.: Hyper-parameter sweep on AlphaZero General. arXiv preprint arXiv:1903.08129 (2019)

[33] Wang, H., Emmerich, M., Preuss, M., Plaat, A.: Analysis of hyper-parameters for small games: Iterations or epochs in self-play? arXiv preprint arXiv:2003.05988 (2020)

[34] Wang, H., Emmerich, M., Preuss, M., Plaat, A.: Alternative loss functions in AlphaZero-like self-play. In: 2019 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE (2019) 155–162

[35] Wang, H., Preuss, M., Plaat, A.: Warm-start AlphaZero self-play search enhancements. In: International Conference on Parallel Problem Solving from Nature, Springer (2020) 528–542

[36] Wang, H., Preuss, M., Plaat, A.: Adaptive warm-start MCTS in AlphaZero-like deep reinforcement learning. arXiv preprint arXiv:2105.06136 (2021)

[37] Wang, H., Preuss, M., Emmerich, M., Plaat, A.: Tackling Morpion Solitaire with AlphaZero-like ranked reward reinforcement learning. In: 2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE (2020) 149–152

[38] Wang, H., Tang, Y., Liu, J., Chen, W.: A search optimization method for rule learning in board games. In: Pacific Rim International Conference on Artificial Intelligence, Springer (2018) 174–181

[39] Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI Competition. AI Magazine **26** (2005) 62–62

[40] Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Technical report, Stanford Logic Group Computer Science Department Stanford University (2008)

[41] Kaiser, D.M.: The design and implementation of a successful general game playing agent. In: FLAIRS Conference. (2007) 110–115

[42] Genesereth, M., Thielscher, M.: General game playing. Synthesis Lectures on Artificial Intelligence and Machine Learning **8** (2014) 1–229

[43] Świechowski, M., Mańdziuk, J.: Fast interpreter for logical reasoning in general game playing. Journal of Logic and Computation **26** (2016) 1697–1727

[44] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518** (2015) 529–533

[45] Mehat, J., Cazenave, T.: Monte-Carlo Tree Search for general game playing. Univ. Paris **8** (2008)

[46] Banerjee, B., Stone, P.: General game learning using knowledge transfer. In: IJCAI Proceedings-International Joint Conference on Artificial Intelligence. (2007) 672–677

[47] Hammersley, J.: Monte Carlo methods. Springer Science & Business Media (2013)

[48] Thielscher, M.: The general game playing description language is universal. In: IJCAI Proceedings-International Joint Conference on Artificial Intelligence. Volume 22. (2011) 1107

[49] Watkins, C.J.C.H.: Learning from delayed rewards. PhD thesis, King's College, Cambridge (1989)

[50] Even-Dar, E., Mansour, Y.: Convergence of optimistic and incremental Q-learning. Advances in Neural Information Processing Systems **14** (2001) 1499–1506

[51] Hu, J., Wellman, M.P.: Nash Q-learning for general-sum stochastic games. Journal of Machine Learning Research **4** (2003) 1039–1069

[52] Wiering, M., Van Otterlo, M.: Reinforcement learning. Adaptation, Learning, and Optimization **12** (2012)

[53] Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research **4** (1996) 237–285

[54] Torrey, L., Shavlik, J.: Transfer learning. In: Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques. IGI Global (2010) 242–264

[55] Vodopivec, T., Samothrakis, S., Ster, B.: On Monte Carlo tree search and reinforcement learning. Journal of Artificial Intelligence Research **60** (2017) 881–936

[56] Méhat, J., Cazenave, T.: Combining UCT and nested Monte Carlo Search for single-player general game playing. IEEE Transactions on Computational Intelligence and AI in Games **2** (2010) 271–277

[57] Cazenave, T., Saffidine, A., Schofield, M.J., Thielscher, M.: Nested Monte Carlo Search for two-player games. In: AAAI. Volume 16. (2016) 687–693

[58] Ruijl, B., Vermaseren, J., Plaat, A., Herik, J.v.d.: Combining simulated annealing and Monte Carlo Tree Search for expression simplification. arXiv preprint arXiv:1312.0841 (2013)

[59] Soemers, D.J., Mella, V., Browne, C., Teytaud, O.: Deep learning for general game playing with Ludii and Polygames. arXiv preprint arXiv:2101.09562 (2021)

[60] Goldwaser, A., Thielscher, M.: Deep reinforcement learning for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence. Volume 34. (2020) 1701–1708

[61] Tao, J., Wu, L., Hu, X.: Principle analysis on AlphaGo and perspective in military application of artificial intelligence. Journal of Command and Control **2** (2016) 114–120

[62] Zhang, Z.: When doctors meet with AlphaGo: potential application of machine learning to clinical medicine. Annals of Translational Medicine **4** (2016)

[63] Silver, A.: Leela Chess Zero: AlphaZero for the PC (2019)

[64] Tian, Y., Ma, J., Gong, Q., Sengupta, S., Chen, Z., Pinkerton, J., Zitnick, C.L.: Elf opengo: An analysis and open reimplementation of AlphaZero. arXiv preprint arXiv:1902.04522 (2019)

[65] Nair, S.: AlphaZero General. `https://github.com/suragnair/alpha-zero-general` (2018) Accessed May, 2018.

[66] Chaslot, G.M.J., Winands, M.H., HERIK, H.J.V.D., Uiterwijk, J.W., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation **4** (2008) 343–357

[67] Chaslot, G.M.B., Winands, M.H., van Den Herik, H.J.: Parallel Monte-Carlo Tree Search. In: International Conference on Computers and Games, Springer (2008) 60–71

[68] Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks **61** (2015) 85–117

[69] Clark, C., Storkey, A.: Training deep convolutional neural networks to play Go. In: International Conference on Machine Learning. (2015) 1766–1774

[70] Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM **38** (1995) 58–68

[71] Heinz, E.A.: New self-play results in computer Chess. In: International Conference on Computers and Games, Springer (2000) 262–276

[72] Wiering, M.A., et al.: Self-play and using an expert to learn to play Backgammon with temporal difference learning. Journal of Intelligent Learning Systems and Applications **2** (2010) 57

[73] Van Der Ree, M., Wiering, M.: Reinforcement learning in the game of Othello: learning against a fixed opponent and learning from self-play. In: 2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), IEEE (2013) 108–115

[74] Plaat, A.: Learning to Play. Springer (2020)

[75] Chen, Y., Huang, A., Wang, Z., Antonoglou, I., Schrittwieser, J., Silver, D., de Freitas, N.: Bayesian optimization in AlphaGo. arXiv preprint arXiv:1812.06855 (2018)

[76] Iwata, S., Kasai, T.: The Othello game on an n× n board is PSPACE-complete. Theoretical Computer Science **123** (1994) 329–340

[77] Buro, M.: The Othello match of the year: Takeshi Murakami vs. Logistello. ICGA Journal **20** (1997) 189–193

[78] Chong, S.Y., Tan, M.K., White, J.D.: Observing the evolution of neural networks learning to play the game of Othello. IEEE Transactions on Evolutionary Computation **9** (2005) 240–251

[79] Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)

[80] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[81] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research **15** (2014) 1929–1958

[82] Coulom, R.: Whole-history rating: A Bayesian rating system for players of time-varying strength. In: International Conference on Computers and Games, Springer (2008) 113–124

[83] Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K., et al.: A racing algorithm for configuring metaheuristics. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation. Volume 2. (2002)

[84] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, Springer (2011) 507–523

[85] Moerland, T.M., Broekens, J., Jonker, C.M.: Model-based reinforcement learning: A survey. arXiv preprint arXiv:2006.16712 (2020)

[86] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep reinforcement learning that matters. In: Proceedings of the AAAI Conference on Artificial Intelligence. Volume 32. (2018)

[87] Igami, M.: Artificial intelligence as structural estimation: Deep Blue, Bonanza, and AlphaGo. The Econometrics Journal **23** (2020) S1–S24

[88] Li, Y., Fang, Y., Akhtar, Z.: Accelerating deep reinforcement learning model for game strategy. Neurocomputing **408** (2020) 157–168

[89] Segler, M.H., Preuss, M., Waller, M.P.: Planning chemical syntheses with deep neural networks and symbolic AI. Nature **555** (2018) 604–610

[90] Fu, M.C.: AlphaGo and Monte Carlo Tree Search: the simulation optimization perspective. In: 2016 Winter Simulation Conference (WSC), IEEE (2016) 659–670

[91] Matsuzaki, K., Kitamura, N.: Do evaluation functions really improve Monte-Carlo Tree Search? ICGA Journal **40** (2018) 294–304

[92] Allis, L.V.: A knowledge-based approach of Connect-Four. J. Int. Comput. Games Assoc. **11** (1988) 165

[93] Reisch, S.: Gobang ist PSPACE-vollständig. Acta Informatica **13** (1980) 59–66

[94] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)

[95] Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: Proceedings of the 26th Annual International Conference on Machine Learning. (2009) 41–48

[96] Mandai, Y., Kaneko, T.: Alternative multitask training for evaluation functions in game of Go. In: 2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI), IEEE (2018) 132–135

[97] Caruana, R.: Multitask learning. Machine Learning **28** (1997) 41–75

[98] Matsuzaki, K.: Empirical analysis of PUCT algorithm with evaluation functions of different quality. In: 2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI), IEEE (2018) 142–147

[99] Thill, M., Bagheri, S., Koch, P., Konen, W.: Temporal difference learning with eligibility traces for the game Connect Four. In: 2014 IEEE Conference on Computational Intelligence and Games, IEEE (2014) 1–8

[100] Zhang, M., Wu, J., Li, F.: Design of evaluation-function for computer Gobang game system. Journal of Computer Applications **7** (2012) 051

[101] Emmerich, M.T., Deutz, A.H.: A tutorial on multiobjective optimization: fundamentals and evolutionary methods. Natural Computing **17** (2018) 585–609

[102] Justesen, N., Mahlmann, T., Risi, S., Togelius, J.: Playing multi-action adversarial games: Online evolutionary planning versus tree search. IEEE Transactions on Computational Intelligence and AI in Games (2017) 281–291

[103] Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo Tree Search. In: International Conference on Computers and Games, Springer (2006) 72–83

[104] Ruijl, B., Vermaseren, J., Plaat, A., van den Herik, J.: Combining simulated annealing and Monte Carlo Tree Search for expression simplification. In: Proceedings of the 6th International Conference on Agents and Artificial Intelligence-Volume 1, SCITEPRESS-Science and Technology Publications, LDA (2014) 724–731

[105] Chaslot, G., Bakkes, S., Szita, I., Spronck, P.: Monte-Carlo Tree Search: A new framework for game AI. In: AIIDE. (2008)

[106] Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., Lew, M.S.: Deep learning for visual understanding: A review. Neurocomputing **187** (2016) 27–48

[107] Runarsson, T.P., Lucas, S.M.: Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. IEEE Transactions on Evolutionary Computation **9** (2005) 628–640

[108] Wu, D.J.: Accelerating self-play learning in Go. arXiv preprint arXiv:1902.10565 (2019)

[109] Cazenave, T., Chen, Y.C., Chen, G.W., Chen, S.Y., Chiu, X.D., Dehos, J., Elsa, M., Gong, Q., Hu, H., Khalidov, V., et al.: Polygames: Improved zero learning. ICGA Journal (2020) 1–13

[110] Silver, A.: Fat Fritz 2: The best of both worlds. `https://en.chessbase.com/post/fat-fritz-2-best-of-both-worlds` (2021) Accessed June, 2021.

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

[112] Perez, D., Samothrakis, S., Lucas, S., Rohlfshagen, P.: Rolling horizon evolution versus tree search for navigation in single-player real-time games. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. GECCO '13, New York, NY, USA, Association for Computing Machinery (2013) 351–358

[113] Liu, J., Liebana, D.P., Lucas, S.M.: Rolling horizon coevolutionary planning for two-player video games. In: 2016 8th Computer Science and Electronic Engineering Conference, CEEC 2016, Colchester, UK, September 28-30, 2016, IEEE (2016) 174–179

[114] Gaina, R.D., Couëtoux, A., Soemers, D.J.N.J., Winands, M.H.M., Vodopivec, T., Kirchgeßner, F., Liu, J., Lucas, S.M., Pérez-Liébana, D.: The 2016 two-player GVGAI Competition. IEEE Transactions on Games **10** (2018) 209–220

[115] Gaina, R.D., Devlin, S., Lucas, S.M., Perez, D.: Rolling horizon evolutionary algorithms for general video game playing. IEEE Transactions on Games (2021)

[116] Boyer, C.: Morpion Solitaire. `http://www.morpionsolitaire.com/` (2020) Accessed May, 2020.

[117] Ma, Q., Ge, S., He, D., Thaker, D., Drori, I.: Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. arXiv preprint arXiv:1911.04936 (2019)

[118] Zhang, W., Dietterich, T.G.: Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. Journal of Artificial Intelligence Reseach **1** (2000) 1–38

[119] James, J., Yu, W., Gu, J.: Online vehicle routing with neural combinatorial optimization and deep reinforcement learning. IEEE Transactions on Intelligent Transportation Systems **20** (2019) 3806–3817

[120] Rego, C., Gamboa, D., Glover, F., Osterman, C.: Traveling salesman problem heuristics: Leading methods, implementations and latest advances. European Journal of Operational Research **211** (2011) 427–441

[121] Hu, H., Duan, L., Zhang, X., Xu, Y., Wei, J.: A multi-task selected learning approach for solving new type 3D bin packing problem. arXiv preprint arXiv:1804.06896 (2018)

[122] Laterre, A., Fu, Y., Jabri, M.K., Cohen, A.S., Kas, D., Hajjar, K., Dahl, T.S., Kerkeni, A., Beguir, K.: Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. arXiv preprint arXiv:1807.01672 (2018)

[123] Wang, H., Schwab, I., Emmerich, M.: Comparing knowledge representation forms in empirical model building. INTELLI 2015 (2015) 184

[124] Moerland, T.M., Broekens, J., Plaat, A., Jonker, C.M.: A0C: AlphaZero in continuous action space. arXiv preprint arXiv:1805.09613 (2018)

[125] Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems. (2015) 2692–2700

[126] Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940 (2016)

[127] Feng, D., Gomes, C.P., Selman, B.: Solving hard AI planning instances using curriculum-driven deep reinforcement learning. arXiv preprint arXiv:2006.02689 (2020)

[128] Cazenave, T.: Nested Monte-Carlo Search. In: Twenty-First International Joint Conference on Artificial Intelligence. (2009)

[129] Cazenave, T., Teytaud, F.: Beam nested rollout policy adaptation. In: ECAI. (2012)

[130] Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artificial intelligence review **18** (2002) 77–95

[131] Coulom, R.: Bayesian Elo computation system. `https://github.com/ddugovic/BayesianElo` (2018) Accessed May, 2018.

[132] Edwards, S.J., et al.: Standard portable game notation specification and implementation guide. http://www. saremba. de/chessgml/standards/pgn/pgncomplete (1994)

English Summary

In deep reinforcement learning, searching and learning techniques are two important components. They can be used independently and in combination to solve different problems in AI, and have achieved impressive results in game playing and robotics. These results have inspired research into artificial general intelligence (AGI), using these methods. Two general frameworks—General Game Playing (GGP) and AlphaZero—have been built as the testbed to explore different aspects of AGI. Both frameworks combine searching and learning methods.

The purpose of this dissertation is to assess the potential of these methods. We study table based classic Q-learning on the GGP system, showing that classic Q-learning works on GGP, although convergence is slow, and it is computationally expensive to learn complex games. For larger games deep neural networks may work better, which we study next.

Previous work shows that combining searching and learning can achieve better performance. In this approach search is based on the learned neural network model and this model is trained by examples that are played by the search algorithm. This dissertation uses an AlphaZero-like self-play framework to explore AGI on small games. By tuning different hyper-parameters, the role, effects and contributions of searching and learning are studied. In order to understand the relative importance of searching and learning, a correlation experiment is performed, suggesting that within a limited budget, a higher number of the outer self-play iterations is more promising than inner training epochs, search simulations, and game episodes.

A further experiment shows that search techniques can contribute as experts to generate better training examples to speed up the start phase of training. This idea is called *warm-start* in the dissertation. We find that in AlphaZero-like self-play, a combination of Rollout and Rave enhancements can improve the start iterations of self-play training, especially with an adaptive iteration length. The

warm-start method is a promising methods to improve the training by embedding searching techniques in self-play based learning.

Based on the successes of AlphaZero-like self-play in two-player games, we explore the possibility in single-player games. In order to extend the AlphaZero-like self-play approach to single player complex games, the Morpion Solitaire game is implemented by combining Ranked Reward method. Morpion Solitaire is a highly challenging combinatorial puzzle. Our first AlphaZero-based approach is able to achieve a near human best record. This result indicates that AlphaZero-like self-play approach is a promising method to explore AGI in single player games.

Overall, in this thesis, both searching and learning techniques are studied (by themselves and in combination) in GGP and AlphaZero-like self-play systems. We do so for the purpose of making steps towards artificial general intelligence, towards systems that exhibit intelligent behavior in more than one domain. Our results are promising, and propose alternative ways in which search enhancements can be embedded as experts to generate better training examples for the start phase of training.

Nederlandse Samenvatting

Twee belangrijke componenten in het vakgebied Deep Reinforcement Learning zijn technieken om te zoeken en te leren. Ze kunnen los van elkaar en in combinatie worden gebruikt om uiteenlopende problemen in de kunstmatige intelligentie op te lossen, en hebben hierin indrukwekkende resultaten bereikt, zowel in robotica als in het spelen van spellen. Deze resultaten hebben onderzoek naar kunstmatige *brede* intelligentie met behulp van zoek- en leertechnieken gestimuleerd (artificial general intelligence: AGI). Twee benaderingen—General Game Playing (GGP) en AlphaZero—worden veel gebruikt om verschillende aspecten van AGI te onderzoeken. Deze twee benaderingen gebruiken combinaties van zoek- en leertechnieken.

Het doel van dit proefschrift is om te kijken hoe ver we kunnen komen met deze technieken. We bestuderen het klassieke Q-learning algoritme (dat nog een tabel als basis-datastructuur gebruikt) in een GGP systeem. We tonen hiermee aan dat klassiek Q-learning werkt in GGP, al convergeert het langzaam, en is er erg veel rekenkracht nodig om ingewikkelder spellen te leren spelen. Voor grotere spellen zouden diepe neurale netwerken wel eens beter kunnen werken, hetgeen ons volgende experiment is. Bestaand onderzoek geeft aan dat de combinatie van zoeken en leren beter zal presteren. In deze werken maken zoektechnieken gebruik van neurale netwerken die getraind worden met voorbeelden die het zoekalgoritme dan weer aanlevert—de zogeheten *self-play* aanpak. In dit proefschrift gebruiken we een op AlphaZero gebaseerde benadering om AGI in kleine spellen te onderzoeken. We onderzoeken met verschillende waarden voor hyperparameters de rol, het effect, en de bijdrage van zoek- en leertechnieken. Teneinde het relatieve belang van zoeken en leren te bekijken, hebben we een experiment gedaan met als uitkomst dat het aantal zoek-iteraties van groter belang is dan training-iteraties, simulaties, of spel-episoden, dan tot nog toe gedacht.

Een ander experiment laat zien dat zoektechnieken gebruikt kunnen worden als expert om de start van het leerproces te verbeteren. Dit idee noemen we de *warme*

start in dit proefschrift. In de AlphaZero benadering kan een combinatie van Rollout en RAVE technieken de eerste iteraties van self-play-training verbeteren, en helemaal met een adaptieve iteratie-lengte. De warme-start techniek is een veelbelovende trainingsmethode om toe te voegen aan de self-play aanpak.

Na het succes van AlphaZero self-play methoden in spellen voor twee personen, onderzoeken we of dit succes ook gebruikt kan worden in spellen voor één persoon, ofwel puzzels. Hiertoe implementeren we het spel Morpion Solitaire met de Ranked Reward methode. Morpion Solitaire is een heel ingewikkelde combinatorische puzzel. Onze eerste AlphaZero-achtige aanpak bleek al in staat om het beste menselijke record te benaderen. Dit geeft aan dat dit een veelbelovende AGI benadering is voor eenpersoons spellen.

In dit proefschrift worden zoek- en leertechnieken bestudeerd, zowel apart als in combinatie, in GGP en in AlphaZero-achtige self-play systemen. Ons doel is om stappen te zetten om kunstmatige intelligentie breder toepasbaar te maken, naar systemen die intelligent gedrag vertonen in meer dan een domein. Onze resultaten zijn veelbelovend, en laten zien in welke richting zoektechnieken kunnen worden toegepast om leertechnieken te verbeteren.

Acknowledgements

The foremost and deepest gratitude is offered to my three supervisors, Prof.dr. Aske Plaat, Dr. Michael Emmerich and Dr. Mike Preuss, who have introduced me to the fascinating research fields of deep reinforcement learning, game playing and multi-objective optimization, and who have guided me to develop a better understanding of the research fields and the scientific spirit of academia. Prof. Plaat, thank you for your trust and encouragement. And thank you for your help and support of finding computation resources and connecting to other researchers in the field. I was always deeply touched when I received your kind response and valuable feedback after asking for your help. Dr. Emmerich, thank you for your guidance, your trust and encouragement. You always help me in time. And I am grateful that you offered me enough trust and freedom to focus on games. Dr. Preuss, thank you for your supervision, especially during the Corona quarantine, also thank you for introducing me to your research community. I am so lucky since all of you are so kind and always willing to help me, both in academic research and in my personal life.

Besides, I would like to thank my thesis committee: Prof.dr. Thomas Bäck, Prof.dr. Marcello Bonsangue, Prof.dr. Joost Batenburg, Prof.dr. Mark Winands, Dr. Mitra Baratchi, Dr. Thomas Moerland and Dr. Ingo Schwab for your insightful comments and suggestions.

To my colleagues in Reinforcement Learning group and Natural Computing group, I sincerely thank you for your help and cooperation. I am so honoured to work with you. I also want to thank LIACS staff, especially Marloes van der Nat and Abdeljalil El Boujadayni for your kind help with daily issues.

I deeply thank my parents, Jiangshan Wang and Yuezhi Song, and my elder sister, Wei Wang, for your consistent trust, support, encouragement and unconditional love. To my niece, Jiaxin Wu, my sister's lovely daughter, currently the youngest

member in my family, your birth is a switch-point of my life since it teaches me the meaning of love and responsibility.

To all of my teachers, classmates and friends during the last 23 years of my study career and my relatives, I would like to thank you for your selfless help, especially Prof.dr. Wu Chen, who led me to the academia and Guoqiang Zhang, my best friend, who consistently supported me.

Last but not the least, thanks to China Scholarship Council for the financial support that enables me to conduct my research and make many good friends in Leiden.

Curriculum Vitae

Hui Wang was born on February 13th, 1992 in Taihu, Anhui, People's Republic of China. He received his Bachelor degree in Computer Science and Technology at Southwest University in June 2015. Then he studied as a Master student and obtained his Master degree in Software Engineering under the supervision of Prof. Wu Chen at the same university in June 2017.

In September 2017, he joined the reinforcement learning research group headed by Prof. Aske Plaat and started his PhD study at Leiden Institute of Advanced Computer Science (LIACS), Leiden University, under the supervision of Prof. Aske Plaat, Dr. Michael Emmerich and Dr. Mike Preuss. His research was funded by China Scholarship Council (CSC).

His research primarily focuses on assessing/developing searching and learning techniques in (deep) reinforcement learning on board games. Besides his research, he also acted as a teaching assistant for two postgraduate courses every year, namely Reinforcement Learning and Multi-Objective Decision Analysis. What's more, he also participated in several academic seminars, such as Dagstuhl seminar (Artificial and Computational Intelligence in Games: Revolutions in Computational Game AI). And he helped to organize 2020 Benelux AI conference (BNAIC) as the multimedia chair.