# Warm-Start AlphaZero Self-play Search Enhancements

Hui Wang(✉), Mike Preuss, and Aske Plaat

Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
h.wang.13@liacs.leidenuniv.nl
http://www.cs.leiden.edu

**Abstract.** Recently, AlphaZero has achieved landmark results in deep reinforcement learning, by providing a single self-play architecture that learned three different games at super human level. AlphaZero is a large and complicated system with many parameters, and success requires much compute power and fine-tuning. Reproducing results in other games is a challenge, and many researchers are looking for ways to improve results while reducing computational demands. 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, namely Rollout, Rapid Action Value Estimate (RAVE) and dynamically weighted combinations of these with the neural network, and Rolling Horizon Evolutionary Algorithms (RHEA). Our experiments indicate that most of these enhancements improve the performance of their baseline player in three different (small) board games, with especially RAVE based variants playing strongly.

**Keywords:** Reinforcement learning · MCTS · Warm-start enhancements · RHEA · AlphaZero-like self-play

## 1 Introduction

The AlphaGo series of programs [1–3] 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 [4,5]. In self-play, *Monte Carlo Tree Search* (MCTS) [6] 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. Thereafter, 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


T. Bäck et al. (Eds.): PPSN 2020, LNCS 12270, pp. 528–542, 2020.
https://doi.org/10.1007/978-3-030-58115-2_37

since only a very small part of the search space is used for training, and training samples in reinforcement learning are heavily correlated [2,7].

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 [6,8], 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.

This may be because AlphaZero-like self-play is still young. Another reason could be that the original AlphaGo paper [1] remarks about AMAF and RAVE [9], two of the best known MCTS enhancements, that "AlphaGo does not employ the *all-moves-as-first* (AMAF) or *rapid action value estimation* (RAVE) 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 *Rolling Horizon Evolutionary Algorithms* (RHEA) 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 [10].

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.

The paper is structured as follows. After giving an overview of the most relevant literature in Sect. 2, we describe the test games in Sect. 3. Thereafter, we describe the AlphaZero-like self-play algorithm in Sect. 4. Before the full length experiments in Sect. 6, an orientation experiment is performed in Sect. 5. Finally, we conclude our paper and discuss future work.

## 2 Related Work

Since MCTS was created [11], many variants have been studied [6,12], especially in games [13]. In addition, enhancements such as RAVE and AMAF have been created to improve MCTS [9,14]. Specifically, [14] 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 2) are a black-box that precludes understanding. The high decision accuracy of deep learning, however, is undeniable [15], as the results in Go (and many other applications) have shown [16]. After AlphaGo Zero [2], 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.

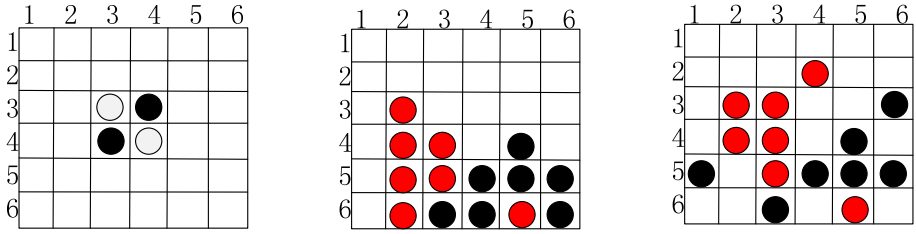
Earlier works on self-play in reinforcement learning are [17–21]. An overview is provided in [8]. For instance, [17,19] compared self-play and using an expert to play backgammon with temporal difference learning. [21] studied co-evolution versus self-play temporal difference learning for acquiring position evaluation in small board Go. All these works suggest promising results for self-play.

More recently, [22] assessed the potential of classical Q-learning by introducing Monte Carlo Search enhancement to improve training examples efficiency. [23] 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.

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.

## 3 Tested Games

In our experiments, we use the games Othello [24], Connect Four [25] and Gobang [26] with $6 \times 6$ board size. All of these are two-player games. In Othello, 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 there is no legal move (the board is full), the player who has less pieces loses the game. Figure 1(a) shows the initial state of Othello. For Connect Four, players take turns dropping their own pieces from the top into a vertically suspended grid. The pieces fall down straightly and occupy the lowest position within the column. The player who first connects a line of four pieces horizontally, vertically, or diagonally wins the game. Figure 1(b) is a game termination example for $6 \times 6$ Connect Four where the red player wins the game. As another connection game, Gobang is traditionally played on a Go board. Players



(a) 6×6 Othello (b) 6×6 Connect Four (c) 6×6 Gobang

**Fig. 1.** Starting position for Othello, example positions for Connect Four and Gobang

also alternate turns, placing a stone of their own color on an empty position. The winner is the first player to connect an unbroken horizontal, vertical, or diagonal chain of 4 stones. Figure 1(c) is a termination example for $6 \times 6$ Gobang where the black player wins the game with 4 stones in a line.

A lot of methods on implementing game-playing programs to play these three games were studied. For instance, Buro used logistic regression to create Logistello [27] to play Othello. In addition, Chong et al. described the evolution of neural networks to play Othello with learning [28]. Thill et al. employed temporal difference learning to play Connect Four [29]. Zhang et al. studied evaluation functions for Gobang [30]. Moreover, Banerjee et al. tested transfer learning in General Game Playing on small games including $4 \times 4$ Othello [31]. Wang et al. assessed the potential of classical Q-learning based on small games including $4 \times 4$ Connect Four [32]. Varying the board size allows us to reduce or increase the computational complexity of these games. In our experiments, we use AlphaZero-like learning [33].

## 4 AlphaZero-Like Self-play Algorithms

### 4.1 The Algorithm Framework

According to [3,33], the basic structure of AlphaZero-like self-play is an iterative process over three different stages (see Algorithm 1).

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 $\pi$ based on the probability $\mathbf{p}$ provided by the policy network $f_\theta$. The hyper-parameters, and the abbreviation that we use in this paper is given in Table 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 $f_\theta$. In (self-)play game episode, from *T'* steps on,

**Algorithm 1.** AlphaZero-like Self-play Algorithm

**function** AlphaZeroGeneralwithEnhancements
  Initialize $f_\theta$ 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 $\pi_t$ before $T'$ or $\arg\max_a(\pi_t)$ after $T'$ step
        executeAction($s_t$, $a_t$)
      Store every $(s_t, \pi_t, z_t)$ with game outcome $z_t$ $(t \in [1, T])$ in $D$
    Randomly sample minibatch of examples $(s_j, \pi_j, z_j)$ from $D$ ▷ stage 2
    Train $f_{\theta'} \leftarrow f_\theta$
    $f_\theta = f_{\theta'}$ if $f_{\theta'}$ is better than $f_\theta$ using **MCTS** mini-tournament ▷ stage 3
  **return** $f_\theta$;

the player always chooses the best action based on $\pi$. Before that, the player always chooses a random move according to the probability distribution of $\pi$ to obtain more diverse training examples. After game ends, the new examples are normalized as a form of $(s_t, \pi_t, z_t)$ 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 [34] based on the specific batch size ($bs$). The neural network is trained with a learning rate ($lr$) and dropout ($d$) by minimizing [35] 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 $\pi$. Dropout is a probability to randomly ignore some nodes of the hidden layer to avoid overfitting [36].

The last stage is the **arena comparison**, where a competition between the newly trained neural network model ($f'_\theta$) and the previous neural network model ($f_\theta$) 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 $f_\theta$. Otherwise, $f_{\theta'}$ is rejected and $f_\theta$ 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.

### 4.2 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 2. 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 [2,37]. After the search, the average win rate value $Q(s, a)$ and visit count $N(s, a)$ in the followed trajectory are updated correspondingly.

**Algorithm 2.** 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;
```

The P-UCT formula that is used is as follows (with $c$ as constant weight that balances exploitation and exploration):

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

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 1 is simply replaced by $\pi_t \leftarrow$ **MCTS**).

### 4.3 MCTS Enhancements

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

**Rollout.** Algorithm 2 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 2, written as *Get result v(s) by performing random rollout until the game ends.*[1]

**RAVE** is a well-studied enhancement for improving the cold-start of MCTS in games like Go (for details see [9]). The same idea can be applied to other domains

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

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 startup is adapted from AMAF in the game of Go [9]. The main pseudo code of RAVE is similar to Algorithm 2, the differences are in line 3, line 12 and line 16. For RAVE, in line 3, policy $\pi_s$ is normalized based on $Q_{rave}(s, \cdot)$. In line 12, the action $a$ with highest $UCT_{rave}$ value, which is computed based on Eq. 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 $N_{rave}$.

$$UCT_{rave}(s, a) = (1 - \beta) * U(s, a) + \beta * U_{rave}(s, a) \tag{2}$$

where

$$U_{rave}(s, a) = Q_{rave}(s, a) + c * P(s, a) \frac{\sqrt{N_{rave}(s, \cdot)}}{N_{rave}(s, a) + 1}, \tag{3}$$

and

$$\beta = \sqrt{\frac{equivalence}{3 * N(s, \cdot) + equivalence}} \tag{4}$$

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.

**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. In our experiments, $v(s)$ is computed as follows:

$$v(s) = (1 - weight) * v_{network} + weight * v_{rollout} \tag{5}$$

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

$$v(s) = (1 - weight) * v_{network} + weight * v_{rora} \tag{6}$$

where

$$weight = 1 - \frac{i}{I'} \tag{7}$$

## 5 Orientation 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. [38] 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. [39] 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. [40] that shows to be competitive in 2 player competitions [41]. The current state of RHEA is documented in [42], 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 against each other with 500 repetitions over all three games, with 100 rollouts per run for all methods. The results are shown in Table 1.

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.

**Table 1.** Comparison of random player, MCTS, Rave, and RHEA on the three games, win rates in percent (column vs. row), 500 repetitions each.

| adv | Gobang | | | | Connect Four | | | | Othello | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 | |

Besides, start sequence planning is certainly harder for Othello where a single move can change large parts of the board.

## 6 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.

### 6.1 Experiment Setup

For all 3 tested games and all experimental training runs based on Algorithm 1, we set parameters values in Table 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 2. Other parameter values are set based on [43,44].

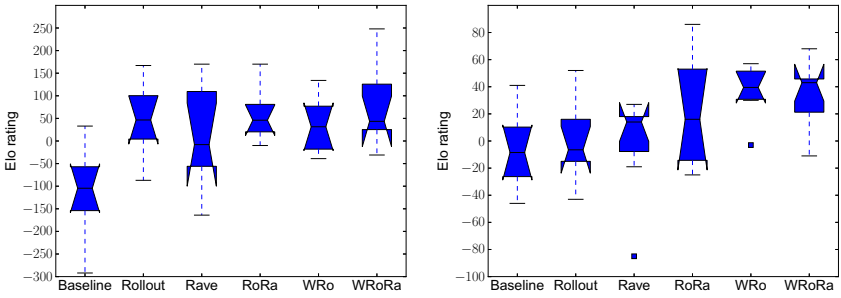
Our experiments are run on a GPU-machine with 2x Xeon Gold 6128 CPU at 2.6 GHz, 12 core, 384 GB RAM and 4x NVIDIA PNY GeForce RTX 2080TI. We use small versions of games ($6 \times 6$) in order to perform a sufficiently high number of computationally demanding experiments. Shown are graphs with errorbars of 8 runs, of 100 iterations of self-play. Each single run takes 1 to 2 days.

**Table 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 | | | |

### 6.2 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 [45] as AlphaGo papers.




(a) 6×6 Gobang (b) 6×6 Connect Four

**Fig. 2.** Tournament results for $6 \times 6$ Gobang and $6 \times 6$ Connect Four among *Baseline*, *Rollout*, *Rave*, *RoRa*, *WRo* and *WRoRa*. Training with enhancements tends to be better than baseline MCTS.

Figure 2(a) displays results for training to play the $6 \times 6$ 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 2(b) shows that all players with warm-start enhancement achieve higher Elo ratings in training to play the $6 \times 6$ 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 3 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 ($6 \times 6$ 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%.

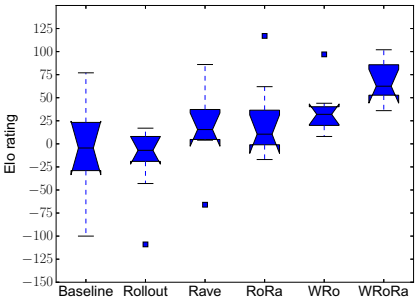



**Fig. 3.** Tournament results for $6 \times 6$ Othello among *Baseline*, *Rollout*, *Rave*, *RoRa*, *WRo* and *WRoRa*. Training with enhancements is mostly better than the baseline setting.

## 7 Discussion and Conclusion

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 performance and speed up training, in this paper, 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 $6 \times 6$ 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 [46], whereas others as Go [2] 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.

## 8 Outlook

We are not aware of other studies on warm-start enhancements of AlphaZero-like self-play. Thus, a number of interesting problems remain to be investigated.

- Which enhancements will work best on which games? Does the above hypothesis hold that games with more consistent opening plans benefit more from the warm-start?
- When (parameter $I'$) and how do we lead over from the start methods to the full AlphaZero scheme including MCTS and neural networks? If we use a weighting, how shall the weight be changed when we lead over? 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.
- Other warm-start enhancements, e.g. built on variants of RHEA's or hybrids of it, shall be explored.
- All our current test cases are relatively small games. How does this transfer to larger games or completely different applications?

In consequence, we would like to encourage other researchers to help exploring this approach and enable using its potential in future investigations.

**Acknowledgments.** Hui Wang acknowledges financial support from the China Scholarship Council (CSC), CSC No.201706990015.

## References

1. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484 (2016)
2. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
3. Silver, D., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
4. Tao, J., Lin, W., Xiaofeng, H.: Principle analysis on AlphaGo and perspective in military application of artificial intelligence. J. Command Control **2**(2), 114–120 (2016)
5. Zhang, Z.: When doctors meet with AlphaGo: potential application of machine learning to clinical medicine. Ann. Transl. Med. **4**(6) (2016)
6. Browne, C., et al.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
7. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
8. Plaat, A.: Learning to play–reinforcement learning and games (2020)
9. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th International Conference on Machine Learning, pp. 273–280 (2007)
10. Nair, S.: AlphaZero general. https://github.com/suragnair/alpha-zero-general (2018). Accessed May 2018
11. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75538-8_7
12. 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, pp. 724–731. SCITEPRESS-Science and Technology Publications, Lda (2014)
13. Chaslot, G., Bakkes, S., Szita, I., Spronck, P.: Monte-Carlo tree search: a new framework for game AI. In: AIIDE (2008)
14. Gelly, S., Silver, D.: Monte-Carlo tree search and rapid action value estimation in computer go. Artif. Intell. **175**(11), 1856–1875 (2011)
15. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
16. Clark, C., Storkey, A.: Training deep convolutional neural networks to play go. In: International Conference on Machine Learning, pp. 1766–1774 (2015)
17. Tesauro, G.: Temporal difference learning and TD-Gammon. Commun. ACM **38**(3), 58–68 (1995)
18. Heinz, E.A.: New self-play results in computer chess. In: Marsland, T., Frank, I. (eds.) CG 2000. LNCS, vol. 2063, pp. 262–276. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45579-5_18
19. Wiering, M.A., et al.: Self-play and using an expert to learn to play backgammon with temporal difference learning. J. Intell. Learn. Syst. Appl. **2**(02), 57 (2010)

20. Van Der Ree, M., Wiering, M.: Reinforcement learning in the game of Othello: learning against a fixed opponent and learning from self-play. In: IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), pp. 108–115. IEEE (2013)
21. Runarsson, T.P., Lucas, S.M.: Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. IEEE Trans. Evol. Comput. **9**(6), 628–640 (2005)
22. Wang, H., Emmerich, M., Plaat, A.: Monte Carlo Q-learning for general game playing. arXiv preprint arXiv:1802.05944 (2018)
23. Wu, D.J.: Accelerating self-play learning in go. arXiv preprint arXiv:1902.10565 (2019)
24. Iwata, S., Kasai, T.: The Othello game on an n * n board is PSPACE-complete. Theor. Comput. Sci. **123**(2), 329–340 (1994)
25. Allis, L.V.: A knowledge-based approach of connect-four. ICGA J. **11**(4), 165 (1988)
26. Reisch, S.: Gobang ist pspace-vollständig. Acta Informatica **13**(1), 59–66 (1980)
27. Buro, M.: The Othello match of the year: Takeshi Murakami vs. Logistello. ICGA J. **20**(3), 189–193 (1997)
28. Chong, S.Y., Tan, M.K., White, J.D.: Observing the evolution of neural networks learning to play the game of Othello. IEEE Trans. Evol. Comput. **9**(3), 240–251 (2005)
29. Thill, M., Bagheri, S., Koch, P., Konen, W.: Temporal difference learning with eligibility traces for the game connect four. In: IEEE Conference on Computational Intelligence and Games, pp. 1–8. IEEE (2014)
30. Zhang, M.L., Wu, J., Li, F.Z.: Design of evaluation-function for computer gobang game system. J. Comput. Appl. **7**, 051 (2012)
31. Banerjee, B., Stone, P.: General game learning using knowledge transfer. In: IJCAI, pp. 672–677 (2007)
32. Wang, H., Emmerich, M., Plaat, A.: Assessing the potential of classical Q-learning in general game playing. In: Atzmueller, M., Duivesteijn, W. (eds.) BNAIC 2018. CCIS, vol. 1021, pp. 138–150. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31978-6_11
33. Wang, H., Emmerich, M., Preuss, M., Plaat, A.: Alternative loss functions in alphazero-like self-play. In: IEEE Symposium Series on Computational Intelligence (SSCI), pp. 155–162. IEEE (2019)
34. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
35. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
36. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)
37. Rosin, C.D.: Multi-armed bandits with episode context. Ann. Mathe. Artif. Intell. **61**(3), 203–230 (2011). https://doi.org/10.1007/s10472-011-9258-6
38. Justesen, N., Mahlmann, T., Risi, S., Togelius, J.: Playing multi-action adversarial games: online evolutionary planning versus tree search. IEEE Trans. Comput. Intell. AI Games **10**, 281–291 (2017)
39. 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 2013, pp. 351–358. New York (2013). Association for Computing Machinery

40. Liu, J., Liebana, D.P., Lucas, S.M.: Rolling horizon coevolutionary planning for two-player video games. In: 8th Computer Science and Electronic Engineering Conference, CEEC 2016, Colchester, UK, 28–30 September 2016, pp. 174–179. IEEE (2016)
41. Gaina, R.D., et al.: The 2016 two-player GVGAI competition. IEEE Trans. Games **10**(2), 209–220 (2018)
42. Gaina, R.D., Devlin, S., Lucas, S.M., Perez-Liebana, D.: Rolling horizon evolutionary algorithms for general video game playing (2020)
43. Wang, H., Emmerich, M., Preuss, M., Plaat, A.: Hyper-parameter sweep on AlphaZero general. arXiv preprint arXiv:1903.08129 (2019)
44. 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)
45. Coulom, R.: Whole-history rating: a Bayesian rating system for players of time-varying strength. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 113–124. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87608-3_11
46. Vinyals, O., et al.: Grandmaster level in StarCraft ii using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)