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

Wang, H.

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

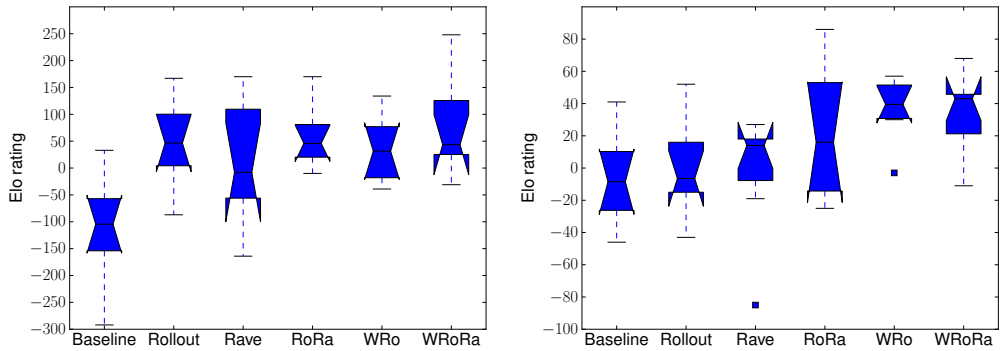
of computationally demanding experiments. Shown are graphs with errorbars of 8 runs, of 100 iterations of self-play. Each single run takes 1 to 2 days.

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

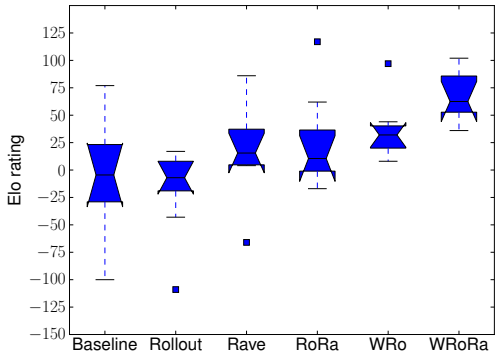


Figure 5.2: Tournament results for 6×6 Othello among *Baseline*, *Rollout*, *Rave*, *RoRa*, *WRo* and *WRoRa*. Training with enhancements is mostly better than the baseline setting.

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.