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

Wang, H.

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

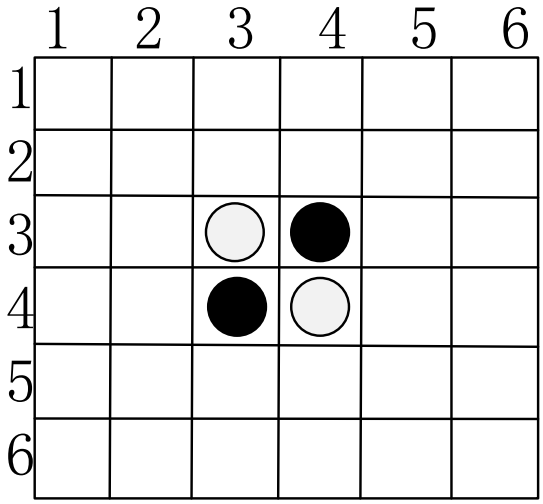


Figure 3.1: Starting position for 6×6 Othello

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:

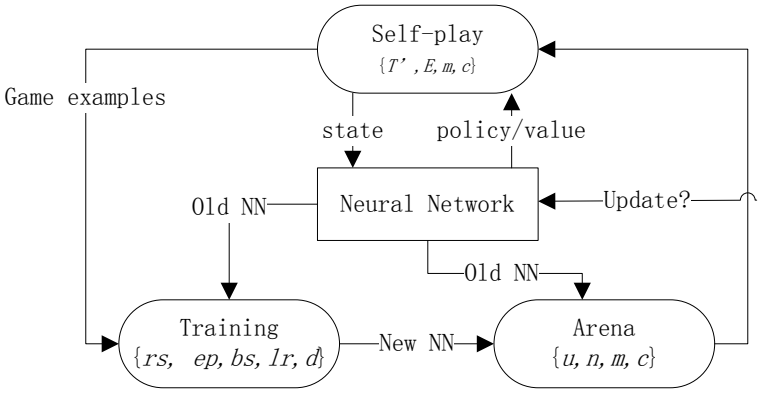


Figure 3.2: A diagram of the schema of AlphaZero-like Self-play Algorithm over 3 stages with corresponding hyper-parameters. Hyper-parameter I controls the loop over these 3 stages.

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.

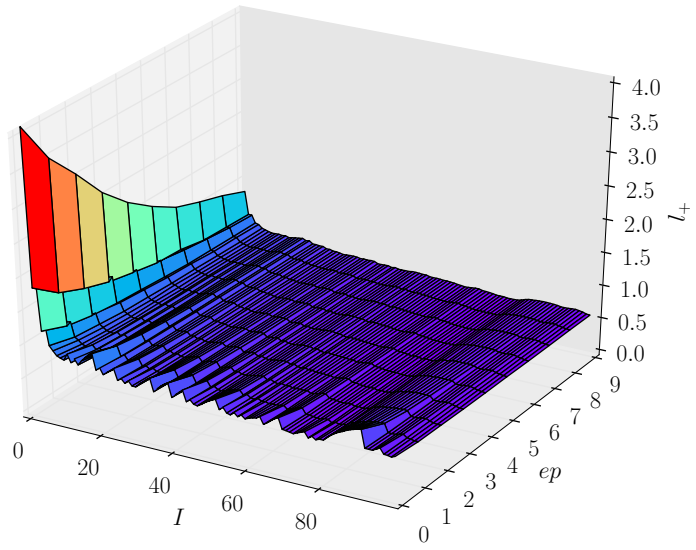


Figure 3.3: Single run training loss over iterations I and epochs ep

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.

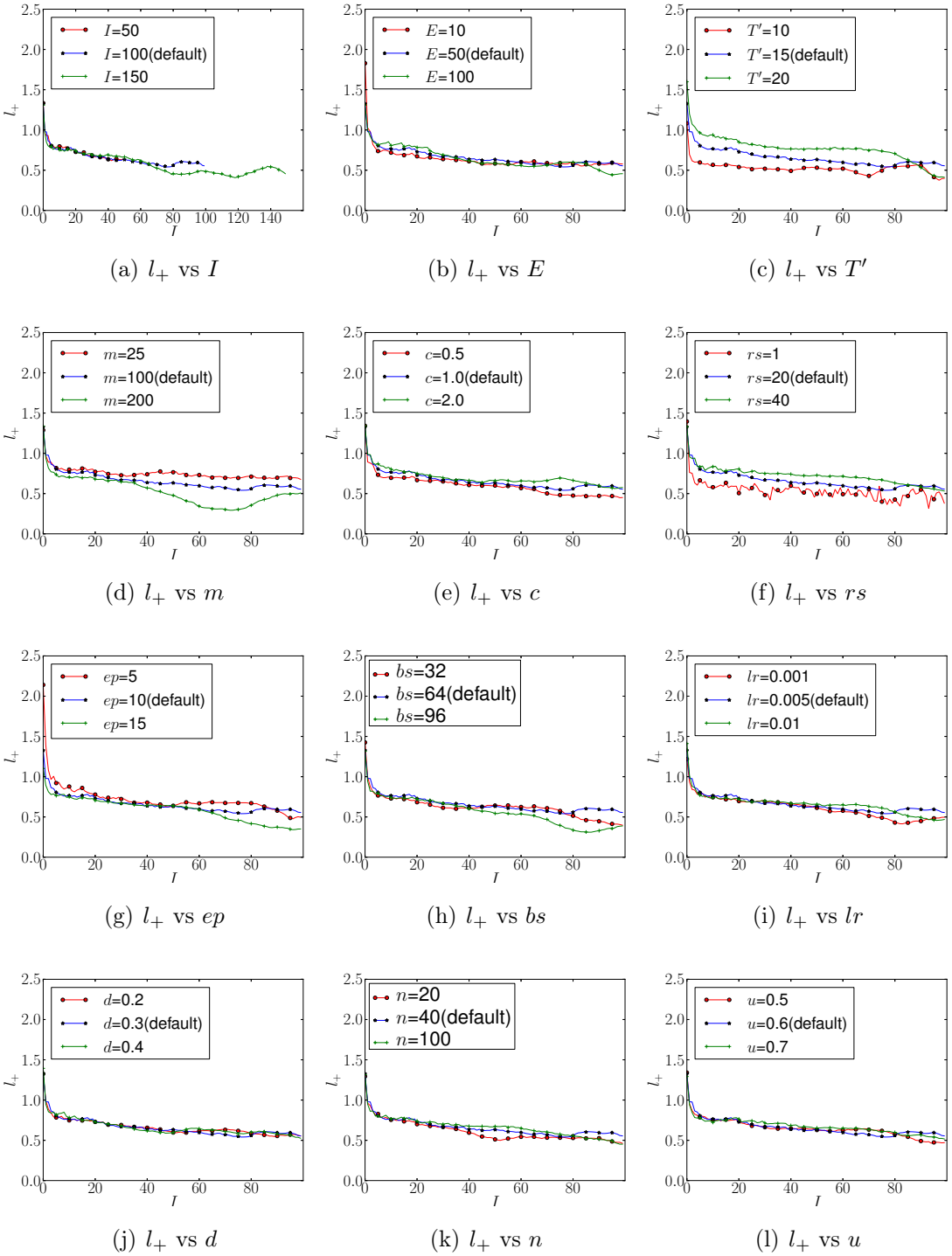
rs: In order to reduce overfitting, it is important to retrain models using previous training examples. Finding a good retrain length of historical training examples is necessary to reduce training time. In Fig. 3.4(f), we see that using training examples from the most recent single previous iteration 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

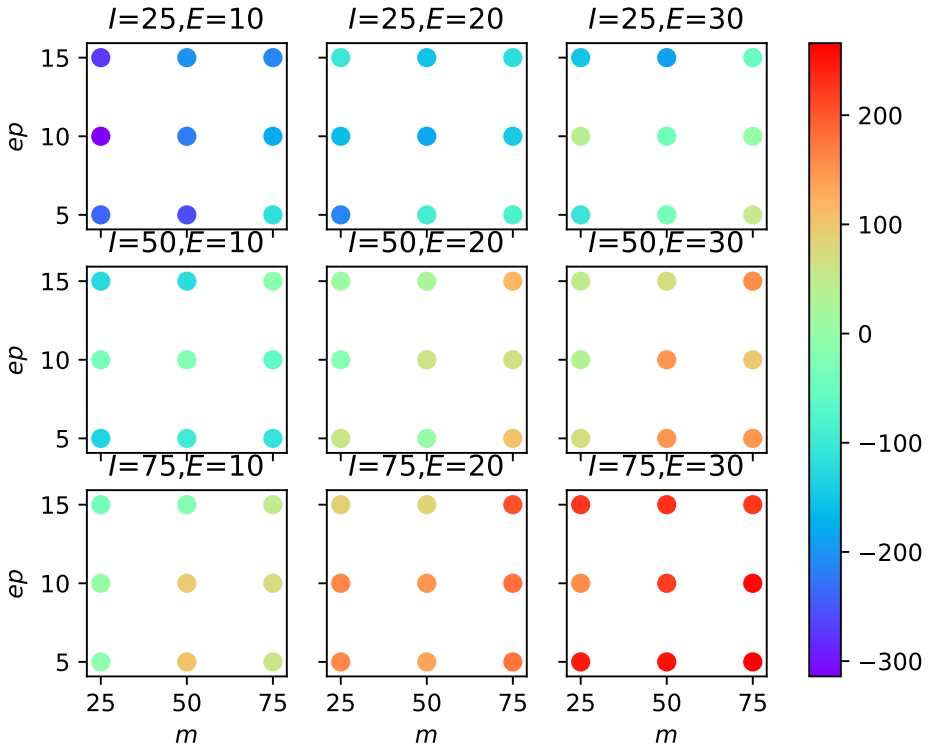


Figure 3.5: Elo ratings of the final best players of the full tournament (3 parameters, 1 target value)

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.

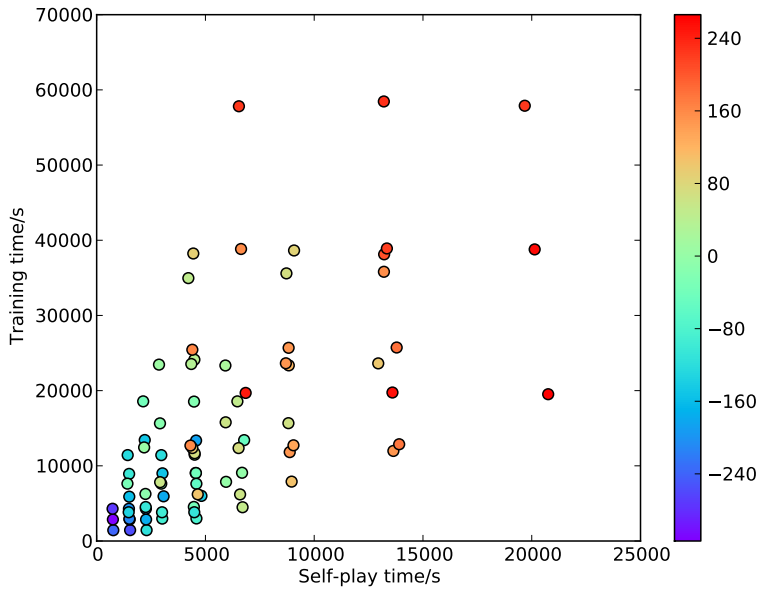


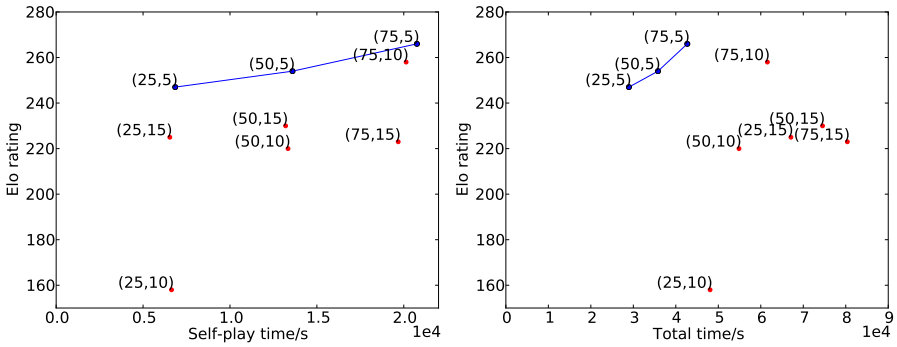
Figure 3.6: Elo ratings of the final best players with different time cost of Self-play and neural network training (same base data as in Fig. 3.5)

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.