Policy or Value? Loss Function and Playing Strength in AlphaZero-like Self-play

Hui Wang,* Michael Emmerich, Mike Preuss, Aske Plaat
Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands
*email: h.wang.13@liacs.leidenuniv.nl

***Abstract*—Recently, AlphaZero has achieved outstanding performance in playing Go, Chess, and Shogi. Players in AlphaZero consist of a combination of Monte Carlo Tree Search and a Deep Q-network, that is trained using self-play. The unified Deep Q-network has a policy-head and a value-head. In AlphaZero, during training, the optimization minimizes the *sum* of the policy loss and the value loss. However, it is not clear if and under which circumstances other formulations of the objective function are better. Therefore, in this paper, we perform experiments with combinations of these two optimization targets. Self-play is a computationally intensive method. By using small games, we are able to perform multiple test cases. We use a light-weight open source reimplementation of AlphaZero on two different games. We investigate optimizing the two targets independently, and also try different combinations (sum and product).**

Our results indicate that, at least for relatively simple games such as 6x6 Othello and Connect Four, optimizing the sum, as AlphaZero does, performs consistently worse than other objectives, in particular by optimizing only the value loss. Moreover, we find that care must be taken in computing the playing strength. Tournament Elo ratings differ from training Elo ratings—training Elo ratings, though cheap to compute and frequently reported, can be misleading and may lead to bias.

It is currently not clear how these results transfer to more complex games and if there is a phase transition between our setting and the AlphaZero application to Go where the sum is seemingly the better choice.

***Index Terms*—AlphaZero, Loss Optimization, Loss Combination, Elo Evaluation**

I. Introduction

The AlphaGo series of papers [1]–[3] have sparked an enormous interest of researchers and the general public alike into deep reinforcement learning. AlphaGo Zero [2], the successor of AlphaGo, even masters the game of Go without human knowledge. It generates game playing data purely by an elegant form of self-play, training a single unified neural network with a policy head and a value head, in a Monte Carlo Tree Search (MCTS) searcher. AlphaZero [3] uses a single architecture for playing three different games (Go, Chess and Shogi) without human knowledge. Many surveys, applications and optimization methods [4]–[7] have been published and transformed the research field into one of the most active of current computer science.

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

Despite the success of AlphaGo and related methods in various application areas [8], [9], there are unexplored and unsolved puzzles in the design and parameterization of the algorithms. The neural network in AlphaZero is represented as $f_\theta = (pi, v)$ (a unified deep network with a policy head and a value head). Policy pi is a probability distribution of choosing the best move. A lower policy loss (loss_pi) indicates a more accurate selection of the best move. Value function v is the prediction of the final outcome. A lower value loss (loss_v) indicates a more accurate prediction of the final outcome. The use of a double-headed network by Alpha(Go) Zero is innovative, and we know of no in-depth study of how the two losses ($loss\_pi$ and $loss\_v$) contribute to the playing strength of the final player. In Alpha(Go) Zero the sum of the two losses is used. Other studies based on the AlphaGo series algorithms just use it that way. In order to increase our understanding of the inner workings of the optimization of the double-headed network we study different combinations of policy and value loss in this paper. Therefore, in this work, we investigate:
a) what will happen if we only optimize a single target?
b) is a product combination a good alternative to summation?

We perform our experiments using a light-weight AlphaZero implementation named AlphaZeroGeneral [10] and focus on smaller games, namely 5×5 and 6×6 Othello [11], 5×5 and 6×6 Connect Four games [12]. The smaller size of these games allows us to do more experiments, and they also provide us largely uncharted territory where we hope to find effects that cannot be seen in Go or Chess.

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

Our contributions can be summarized as follows:

1) In our smaller games, minimizing the value loss gives consistently better results than the summation of the value and the policy loss, contradicting both the default setting of AlphaZeroGeneral and Alpha(Go) Zero, that use the sum of the two losses.
2) In self-play training, it is easy to compute a running Elo rating for the two players being trained. This relative training Elo is misleading and can easily lead to wrong conclusions. A (more expensive) full tournament Elo rating should be used when comparing playing strengths.

The paper is structured as follows. Part II presents related work. Part III presents games tested in the experiments. Part IV introduces the AlphaZero algorithm (with important parameters and default loss function) and Bayesian Elo system. Part V sets up the experiments. Part VI presents the experimental results. Part VII concludes the paper. Part VIII discusses prospective directions for future work.

II. Related work

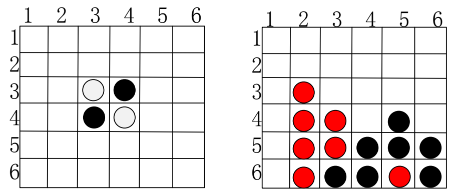
Deep reinforcement learning [13] is currently one of the most active research areas in artificial intelligence, reaching human level performance for difficult games such as Go [14], which was almost unthinkable 10 years ago. Since Mnih et al. reported human-level control for playing Atari 2600 games by means of deep reinforcement learning [15] in 2015, the performance of Deep Q-networks (DQN) improved dramatically.

We have also observed a shift in DQN from imitating and learning from expert human players [1] to relying more on self-play. This has been advocated in the area of reinforcement learning [16]–[18] for quite some time already. Silver et al. [2] turned to self-play to generate training data instead of training from human data (AlphaGo Zero), which not only saves a lot of work of collecting and labeling data from human experts, but also shifts the constraining factor for learning from available data to computing power, and achieves a form of efficient curriculum learning [19]. This approach was generalized to a framework (AlphaZero) that showed that the same approach that worked in Go, also worked in Shogi and Chess, demonstrating how to transfer the learning process to other tasks [3].

Reinforcement learning is a very active field. We see a move away from human data to self-play. After many years of active research in MCTS [20], currently most research effort is in improving DQN variants. AlphaGo is a complex system with many tunable hyper-parameters. It is unclear if the many choices concerning parameters and methods that have been made in the AlphaGo series are close to optimal or if the choices can be improved by, e.g., choosing a different parameter set [21]. This includes the choice of optimization tasks (loss functions) used for measuring training success. Even if the choices were very good for Go and other complex games, this does not necessarily transfer well to less complex tasks.

III. Test Games: Othello/Connect Four

In our experiments, we use game Othello and Connect Four on 5×5 and 6×6 board size respectively. Othello (also known as Reversi) is a two-player game. Players take turns placing their own color pieces. During a game, 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 1(a) is the start configuration for 6×6 Othello. Connect Four is a two-player connection game. Players take turns dropping their own pieces from the top into a vertically suspended grid. The pieces fall straight down and occupy the lowest position within the column. The player who first forms a horizontal, vertical, or diagonal line of four pieces wins the game. Fig 1(b) is a game termination example for 6×6 Connect Four where the red player wins the game.



(a) 6×6 Othello (b) 6×6 Connect Four

Fig. 1. Test Games Examples

There is a wealth of research on finding playing strategies for these two games by means of different methods. For example, De Jong et al. applied experience-based learning to playing Othello [22]. Chong et al. described the evolution of neural networks for learning to play Othello [23]. Edelkamp et al. used Connect Four as a case study for state space search [24]. Thill et al. applied temporal difference learning to play Connect Four [25], and Banerjee et al. tested knowledge transfer in General Game Playing on small games including 4×4 Othello [26]. Wang et al. assessed the potential of classical Q-learning based on small games including 4×4 Connect Four [27]. Obviously, these two games are commonly tested in game playing.

IV. AlphaZero

A. The Base Algorithm

According to [2], [3], the fundamental structure of AlphaZero algorithm is an *iteration* over three different stages (see Algorithm 1).

The first stage is a **self-play** tournament. In the following description the hyper-parameters are shown in italics. 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 an enhanced policy $\overrightarrow{\pi}$ based on the playing policy ($\overrightarrow{pi}$) provided by the current best neural network model (*nnm*). In MCTS, *Cpuct* is used to balance exploration and exploitation of game tree search. *mctssimulation* is the number of times to run down from the root for building the game tree, where the *nnm* provides the value (v) of the states for MCTS. For actual (self-)play of the game, from *tempThreshold* steps on, the player always chooses the best move according to $\overrightarrow{\pi}$. Before that, up to *tempThreshold* steps, the player always chooses a random move based on the probability distribution from $\overrightarrow{\pi}$. After finishing the games, these new examples are normalized as a form of $(s_t, \overrightarrow{\pi}_t, z_t)$ and appended to the queue (*trainingExamplesList*). If the iteration count surpasses the *retrainlength*, the first elements in the queue will be removed.

The second stage consists of **neural network training**, using data from self-play tournament. During training, there are

Algorithm 1 AlphaZero Algorithm

```
1: function AlphaZero(initial neural network model: nnm, parameter setting: ps)
2:   while current iteration<ps.iteration do                                  ▷ stage 1
3:     while current episode<ps.episode do
4:       while !game terminates do
5:         $\overrightarrow{\pi}$ ← MCTS(nnm, ps.Cpuct, ps.mctssimulation, $s$);
6:         if current game step<ps.tempThreshold then
7:           action∼ $\overrightarrow{\pi}$;
8:         else
9:           action← $\arg\max_a \overrightarrow{\pi}$;
10:    trainingExamplesList← append examples;
11:    if iteration of trainingExamplesList>ps.retrainlength then
12:      trainingExamplesList.pop(0);
13:    while current epoch<ps.epoch do                                        ▷ stage 2
14:      batch← trainingExamples.size/ps.batchsize;
15:      while current batch<batch do
16:        loss_pi, loss_v, nnnw← trainNNT( ps.learningrate, ps.dropout, training examples of current batch) by optimizing the loss function;
17:    while current arenacompare<ps.arenacompare do                          ▷ stage 3
18:      while !game terminates do
19:        if player==player1 then
20:          $\overrightarrow{\pi_1}$ ← MCTS(nnnm, ps.Cpuct, ps.mctssimulation, $s$);
21:          action← $\arg\max_a \overrightarrow{\pi_1}$;
22:        else
23:          $\overrightarrow{\pi_2}$ ← MCTS(pnnm, ps.Cpuct, ps.mctssimulation, $s$);
24:          action← $\arg\max_a \overrightarrow{\pi_2}$;
25:    if player1.win/ps.arenacompare≥ps.updateThreshold then
26:      nnm←nnnm;
27:  return nnm;
```

several *epochs*. In each *epoch*, training examples are divided into several small batches [28] according to the specific *batchsize*. The neural network is trained to optimize (minimize) [29] the value of the *loss function* which (see Part IV-B) sums up the mean-squared error between predicted outcome and real outcome and the cross-entropy losses between $\overrightarrow{pi}$ and $\overrightarrow{\pi}$ with a *learningrate* and *dropout*[1].

The last stage is **arena comparison**[2], which is comparing the newly trained neural network model with the previous neural network model. The player will adopt the better model for the next iteration. In order to achieve this, the newly trained neural network (*nnnw*) and the previous best neural network (*pnnw*) are compared by playing against each other for *arenacompare* games. If the *nnnw* wins more than a fraction of *updateThreshold* games, it is replacing the previous best *pnnw*. Otherwise, the *nnnw* is rejected and the *pnnw* is kept as current best model. In order to present this process intuitively, we present all 12 parameters in their corresponding positions in the pseudo code (Algorithm 1).

B. Loss Function

The ***training loss function*** consists of *loss_pi* and *loss_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 $\overrightarrow{pi}$ over all legal actions as outputs. $\overrightarrow{pi}_\theta$ is the policy provided by f_θ to guide MCTS for playing games. After performing MCTS, we obtain an improvement estimate $\overrightarrow{\pi}$. It is an aim of the training to make $\overrightarrow{\pi}$ more similar to $\overrightarrow{pi}$. This can be achieved by optimizing the cross entropy of two distributions. Therefore, the *loss_pi* can be defined as $-\overrightarrow{\pi}\log(\overrightarrow{pi}_\theta(s_t))$. The other aim is to minimize the difference between the output value ($v_\theta(s_t)$) of the s according to f_θ and the real outcome ($z_t \in \{-1, 1\}$) of the game. Therefore, *loss_v* can be defined as $(v_\theta(s_t) - z_t)^2$. Summarizing, the total loss function of AlphaZero can be defined as Equation 1.

$$loss\_pi + loss\_v = -\overrightarrow{\pi}\log(\overrightarrow{pi}_\theta(s_t)) + (v_\theta(s_t) - z_t)^2 \quad (1)$$

C. Bayesian Elo System

The ***Elo rating function*** has been developed as a method for calculating the relative skill levels of players in games [31]. Usually, in zero-sum games, there are two players, player A and B. If player A has an Elo rating of R_A and B has an Elo rating of R_B, then the expectation of that player A wins the next game can be calculated by $E_A = \frac{1}{1+10^{(R_B-R_A)/400}}$. If the real outcome of the next game is S_A, then the updated Elo rating of player A can be calculated 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 set as a bigger value for weaker players but a smaller value for stronger players. Following [3], in our design, we adopt the Bayesian Elo system [32] to show the improvement curve of the learning player during the whole training process. We furthermore also employ this method to assess the playing strength of the final models.

V. Experimental Setup

Our experiments are performed on a GPU server with 128G RAM, 3TB local storage, 20 Intel Xeon E5-2650v3 CPUs (2.30GHz, 40 threads), 2 NVIDIA Titanium GPUs (each with 12GB memory) and 6 NVIDIA GTX 980 Ti GPUs (each with 6GB memory). On these GPUs, every algorithm training run takes several days.

A. Parameter Values

In this work, all neural network models share the same structure, which consists of 4 convolutional neural networks and 2 fully connected layers [10]. The parameter values for Algorithm 1 used in our experiments are given in Table I. The values are based on work reported in [21].

B. Optimization Targets

As we want to assess the effect of optimizing different loss functions, we employ a weighted sum loss function based on Equation 1:

$$\lambda(-\overrightarrow{\pi}\log(\overrightarrow{pi}_\theta(s_t))) + (1-\lambda)(v_\theta(s_t) - z_t)^2 \quad (2)$$

where λ is a weight parameter. This provides some flexibility to gradually change the nature of the function. In our experiments, we first set λ=0 and λ=1 in order to assess the two targets independently. Then we fix λ at 0.5, resulting in the

[1] *dropout* is used as probability to randomly ignore some nodes of the hidden layer. This mechanism is used to reduce overfitting [30].

[2] Note that 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.

TABLE I
DEFAULT PARAMETER SETTING

Parameter	Default Value
iteration	200
episode	50
tempThreshold	15
mctssimulation	100
Cpuct	1.0
retrainlength	20
epoch	10
batchsize	64
learningrate	0.005
dropout	0.3
arenacompare	40
updateThreshold	0.6

equivalent of Equation 1. Furthermore, we employ a product combination loss function as follows:

$$-\overrightarrow{\pi}\log(\overrightarrow{pi}_\theta(s_t)) \times (v_\theta(s_t) - z_t)^2 \tag{3}$$

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

C. Measurements

The chosen loss function is used to guide each training process, with the expectation that smaller loss is supposed to mean a stronger model. However, in practise, we have found that this is not always the case and another measure is needed to check. Therefore, following Deep Mind's work, we also employ Bayesian Elo ratings to describe the playing strength of the model in every iteration. In addition, for each game, we use all best players trained from the four different targets (2 single losses, sum, and product) and 8 repetitions plus a random player to play the game with each other for 20 times. From this, we calculate the Elo ratings of these 33 players to show the real playing strength of a player, rather than the playing strength only based on its own whole training history.

VI. EXPERIMENT RESULTS

In the following, we present the experimental results from 3 aspects according to the measurements introduced above (i.e. training loss, the whole history training Elo rating and the

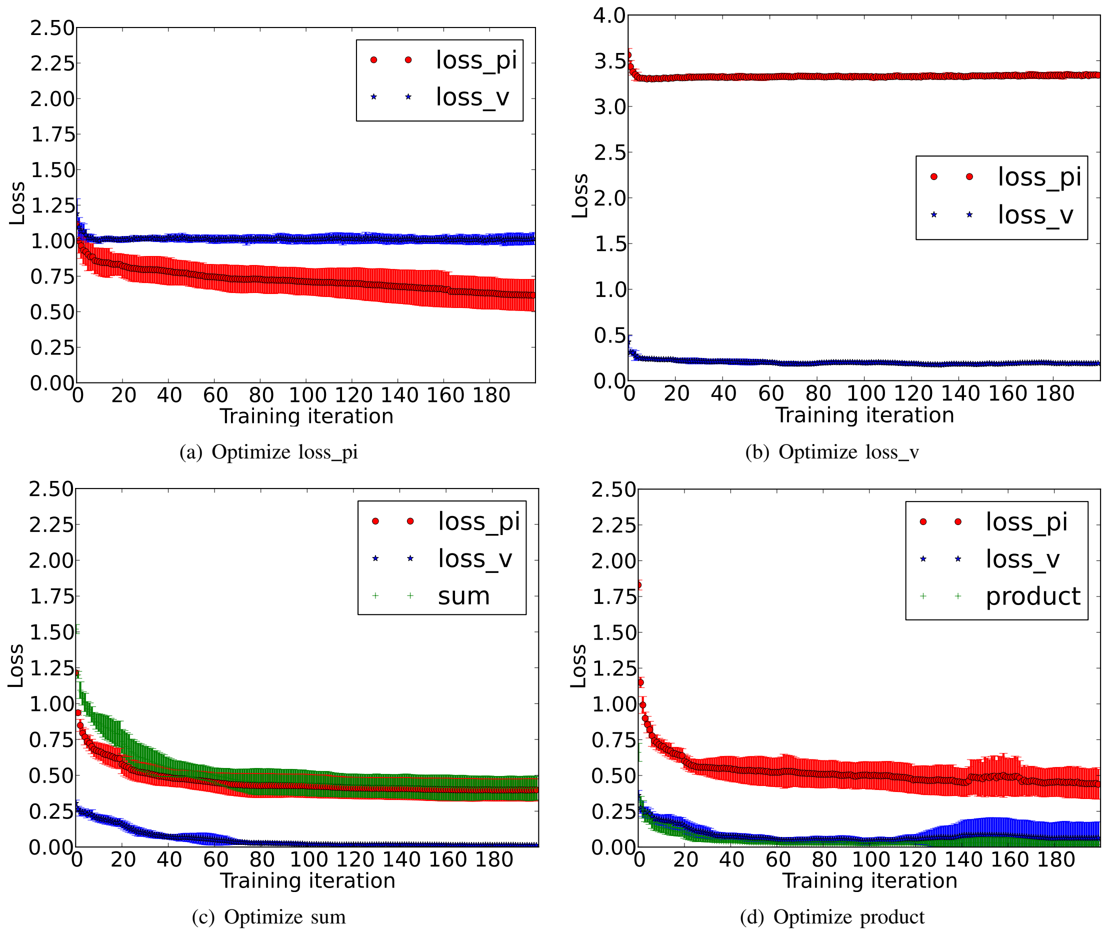


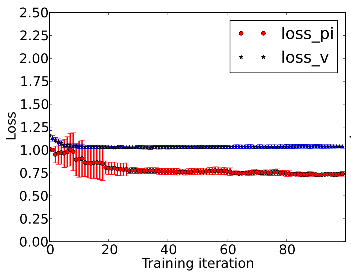
Fig. 2. Training losses for optimizing different targets in 5×5 Othello, averaged from 8 runs. All measured losses are shown, but only one of these is optimized for. Except for the sum, the target that is optimized for is also the lowest

tournament Elo rating of the final best player[3]). Error bars indicate standard deviation of the 8 runs.

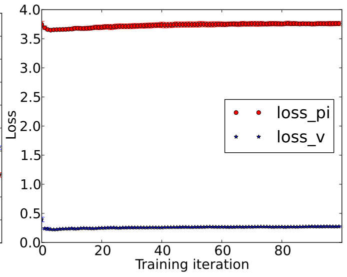
A. Training Loss

We first show the training losses in every iteration during the training phase, with different loss measures, but only one optimization task per diagram, which means we need four of these per game. This means that we can see what optimizing for a specific target actually means for the other loss types.

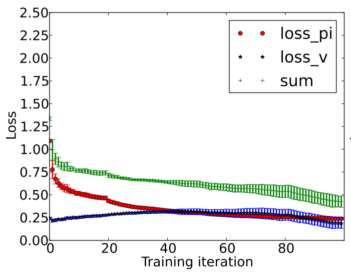
For 5×5 Othello, from Fig 2(a), we find that by only optimizing loss_pi, loss_pi is significantly minimized to about 0.6 at the end of each training, where loss_v is minimized to a level at 1.0 after 10 iterations. From Fig 2(b), the results show that when only optimizing loss_v, loss_v is minimized from more than 0.5 to about 0.2 the end of each training, where loss_pi is minimized from more than 3.5 to about 3.3. In Fig 2(c), we see that when the sum of loss_pi and loss_v is optimized, both losses are reduced significantly. The loss_pi decreases from about 1.2 to 0.5, loss_v surprisingly decreases to 0. Fig 2(d), it is similar to Fig 2(c), while the product of loss_pi and loss_v is optimized, the loss_pi and loss_v are both reduced as well. The loss_pi decreases to 0.5, the loss_v also surprisingly decreases to about 0.



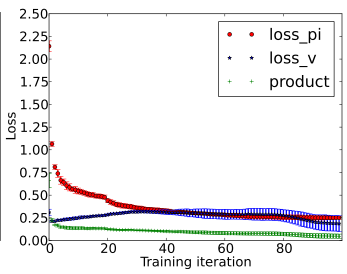
(a) Optimize loss_pi



(b) Optimize loss_v



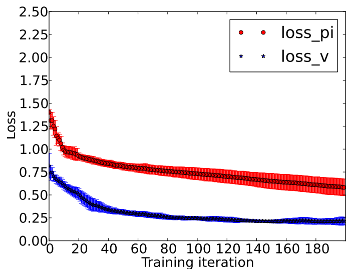
(c) Optimize sum



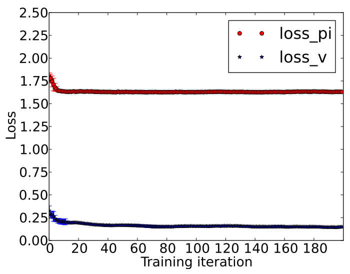
(d) Optimize product

Fig. 3. Training losses for optimizing different targets in 6×6 Othello, averaged from 8 runs. All measured losses are shown, but only one of these is optimized for (similar to the smaller Othello game in Fig 2). Note the different scaling for subfigure (b). Except for sum, the target that is optimized for is the lowest

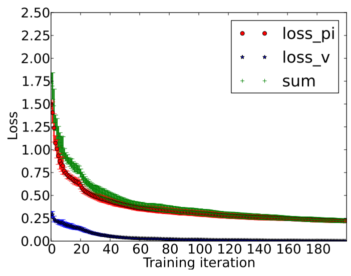
For the larger 6×6 Othello, we find that when only optimizing loss_pi, loss_pi is significantly minimized to about 0.75 at the end of each training, where loss_v is minimized to a level at 1.05 after about 10 iterations (Fig 3(a)). For optimizing loss_v (Fig 3(b)), the results show that loss_v is reduced from more than 0.5 to about 0.25 at the end of each training, but loss_pi seems to stay at the similar level. For optimizing the sum (Fig 3(c)), we find in contrast to 5×5 Othello that loss_pi decreases from about 1.1 to 0.4, whereas loss_v increases slightly from about 0.2 and then decreases to about 0.2 again. We also find a similar behavior of loss_v when optimizing the product of losses (Fig 3(d)), with the difference that the final computed loss is much lower as the values are usually smaller than one. However, the similarity of the single losses is striking.



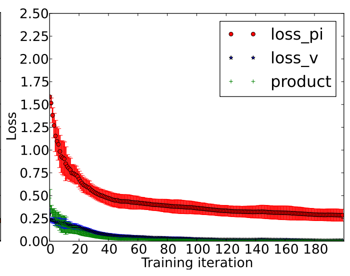
(a) Optimize loss_pi



(b) Optimize loss_v



(c) Optimize sum



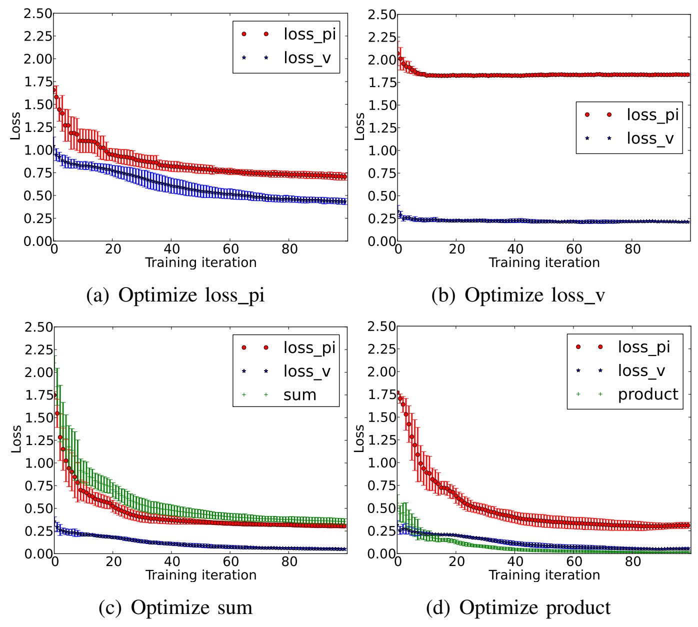
(d) Optimize product

Fig. 4. Training losses for optimizing the four different targets in 5×5 Connect Four, aggregated from 8 runs. Loss_v is always the lowest

For 5×5 Connect Four (displayed in Fig 4(a)), we find that when only optimizing loss_pi, it is significantly minimized from 1.4 to about 0.6 at the end of each training, whereas loss_v is minimized much quicker from 1.0 to about 0.2, where it is almost stationary. Optimizing loss_v (Fig 4(b)) leads to some reduction from more than 0.5 to about 0.15, but loss_pi is not moving much after an initial slight decrease to about 1.6. For minimizing the sum (Fig 4(c)) and the product (Fig 4(d)), the behavior of loss_pi and loss_v is very similar, they both decrease steadily, until loss_v surprisingly reaches 0. Of course the sum and the product arrive at different values, but in terms of both loss_pi and loss_v they are not different.

The training process of the larger 6×6 Connect Four is investigated in Fig 5(a). We find that optimizing loss_pi reduces it significantly from 1.7 to about 0.7 at the end of each training, where loss_v is minimized from 1.2 to about 0.4. For the scenario with optimizing loss_v (Fig 5(b)), we find a similar behavior than for the smaller Connect Four. After some initial progress, there is only stagnation. Again, for optimizing the sum and the product, the target value changes, but the single loss values loss_pi and loss_v behave similarly (Figs 5(c) and 5(d)). Thus we see that both targets lead to very

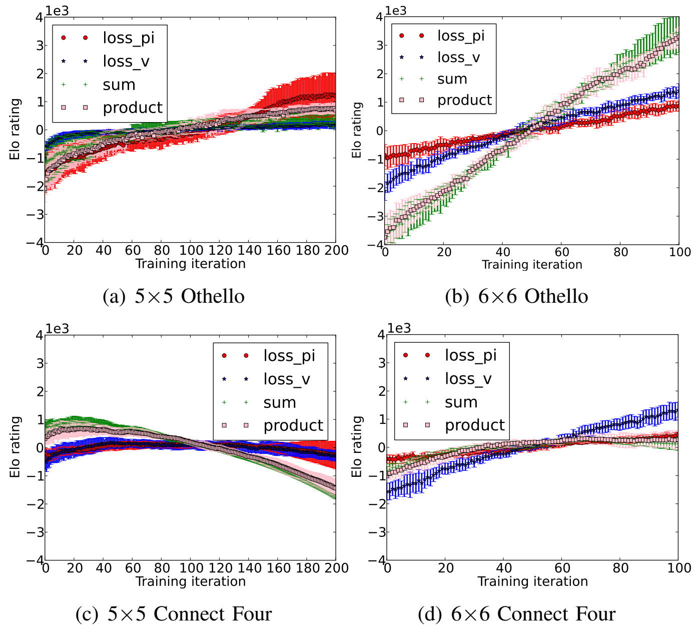
[3]Source code of measuring loss, training Elo rating and tournament Elo rating is released on github: https://github.com/wh1992v/lossfunctionfora0g



(a) Optimize loss_pi (b) Optimize loss_v

(c) Optimize sum (d) Optimize product

Fig. 5. Training losses for optimizing different targets in 6×6 Connect Four, averaged from 8 Runs. Loss_v is the lowest except for the product target



(a) 5×5 Othello (b) 6×6 Othello

(c) 5×5 Connect Four (d) 6×6 Connect Four

Fig. 6. The whole history Elo rating at each iteration during training for different games, aggregated from 8 runs. The training Elo for sum and product in panel b and c shows inconsistent results

similar training processes.

B. Whole History Training Elo Rating

Following the AlphaGo series papers, we also investigate the whole history training Elo rating of every iteration during the training. However, these papers present results of a single training run, whereas we provide means and variances for 8 runs for each targets, categorized by different games in Fig 6. We display the Elo progression obtained from the different optimization targets for one game together. However, one shall be aware that their numbers are not directly comparable as they stem from players who have never seen each other. Nevertheless, the trends are important, and it is especially interesting to see if the Elo values correlate with the progression of the losses.

From Fig 6(a) (small 5×5 Othello) we see that for all optimization tasks, Elo values steadily improve, while they raise fastest for loss_pi. In Fig 6(b), we find that for the bigger 6×6 Othello version, Elo values also always improve, but much faster for the sum and product target, compared to the single loss targets.

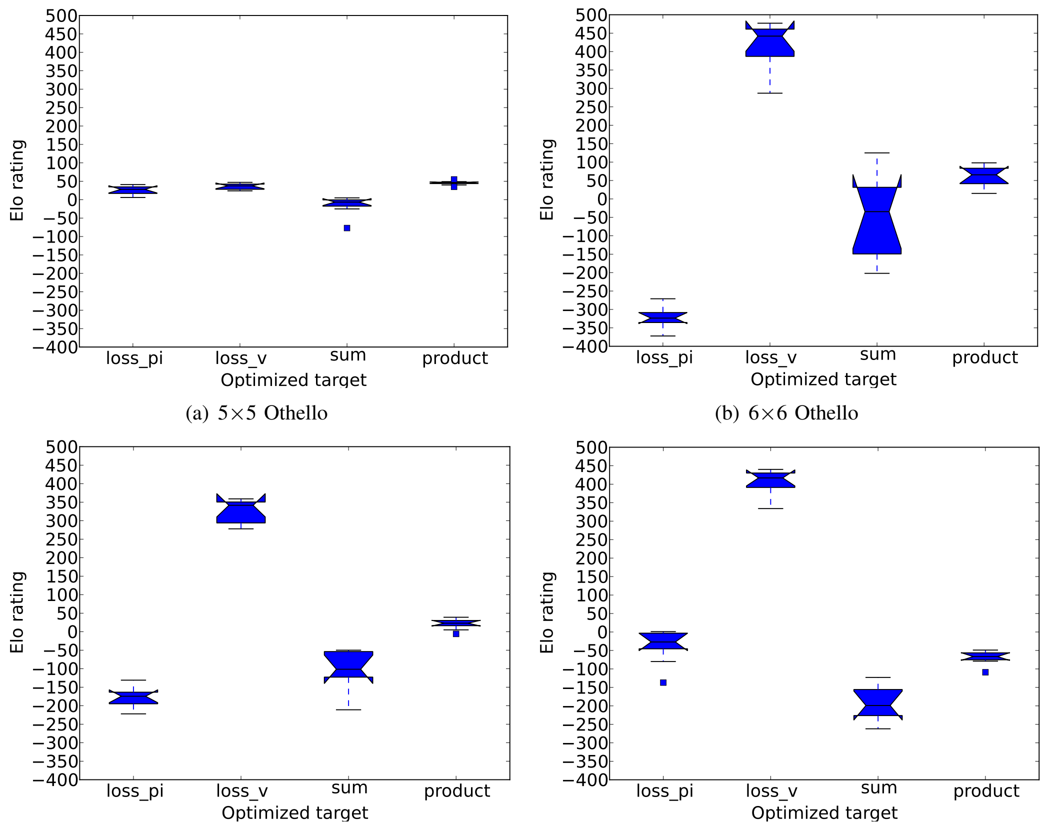
Figures 6(c) and 6(d) show the Elo rate progression for training players with the four different targets on the small and larger Connect Four setting. This looks a bit different from the Othello results, as we find stagnation (for 6×6 Connect Four) as well as even degeneration (for 5×5 Connect Four).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

Overall, we can conclude that the whole history Elo rating is certainly good for assessing if training actually works, whereas the losses alone do not always show that. We may even experience contradicting outcomes as stagnating losses and rising Elo rates (for the big Othello setting and loss_v) or completely counterintuitive results as for the small Connect Four setting where Elo rates and losses are partly anti-correlated. We seemingly have experimental evidence for the fact that training losses and Elo rates are by no means exchangeable as they can provide very different impressions of what is actually happening.

C. The Final Best Player Elo Rating

In order to measure which target can achieve better playing strength, we take all the final models trained from 8 runs and 4 targets plus a random player to pit against each other for 20 times in a full round robin tournament. This enables a direct comparison of the final outcomes of the different training processes with different targets. It is thus more informative than the whole history training Elo, but provides no information during the training process. In principle, we could of course do that also during the training at certain iterations, but this is a computationally very expensive process that would slow down learning a lot.

The results are presented in Fig 7. and show that optimizing loss_v achieves the highest Elo rating with small variance for 6×6 Othello, 5×5 Connect Four and 6×6 Connect Four. For 5×5 Othello, with 200 training iterations, the difference between the results is small. We therefore presume that optimizing loss_v is the best choice for the games we focus on. This is somewhat surprising because we expected the sum to perform best as documented in the literature. However, it may



(a) 5×5 Othello

(b) 6×6 Othello

(c) 5×5 Connect Four

(d) 6×6 Connect Four

Fig. 7. Round-robin tournament of all final models from optimizing different targets in different games over 8 runs. For each game we run the 8 final models from the 4 different optimization targets plus a random player (i.e. 33 players in total) against each other. In panel (a) the difference is small. In panel b, c, and d, the Elo rating of loss_v optimized players clearly dominates.

be the case that this applies to smaller games only, and 5×5 Othello already seems to be a border case where overfitting levels out all differences.

In conclusion, we find that optimizing for the loss_v function only is an alternative to the sum target for certain cases. We also report exceptions, especially in relation to the Elo rating as calculated during training. The relation between Elo and loss during training is sometimes inconsistent (5×5 Connect Four training shows Elo decreasing while the losses are actually minimized). A combination achieves lowest loss, but loss_v achieves the highest Elo. If we optimize the combination, optimizing the product can result to a higher Elo rating model in these games.

VII. Conclusion

Most function approximators in supervised learning and reinforcement learning use a single neural network with a single input and output. In reinforcement learning, the network either is a policy network, or a value network. Alpha(Go) Zero introduces the innovation of optimizing for *both* policy and value, using a single unified network, with two heads, a policy head and a value head. Alpha(Go) Zero and other works optimize this network using an unweighted sum of policy loss and value loss. In this paper, we study four different loss function combinations: (1) loss_pi, (2) loss_v, (3) loss_pi + loss_v, (4) loss_pi× loss_v. We use the open source AlphaZeroGeneral system for light-weight self-play experiments, using two small games, Connect Four and Othello. Surprisingly, for our games, we find that loss_v achieves the highest tournament Elo rating, in contrast to what AlphaZero uses and in contrast to the defaults of AlphaZeroGeneral, showing that default hyper-parameter settings may be non-optimal, especially for the smaller games we investigate here.

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

VIII. Outlook

The running training Elo is computed based on the two active players in the self-play training. We find that tournament results, that are computed using many more diverse players, are more reliable. The narrow basis may be more susceptible to bias, and a broader basis during training may be able to improve training results.

The self-play results by Alpha(Go) Zero are very impressive. However, self-play training uses huge computational resources. We used small games in order be able to run many variations and experiments in a reasonable time (weeks), and in order to be able to do the kind of loss function experimentation that we wanted to do. Self-play research will benefit greatly from efficiency improvements and further development of (open source) systems such as AlphaZeroGeneral.

For future work, scaling studies between large and small games are interesting. Furthermore, the weight value λ in Equation 2 should be further optimized. It remains an open question whether the sum function needs to be revised also for games on larger game boards. Moreover, based on the findings of this paper, we encourage a deeper analysis of Elo rating based performance analysis with the goal to obtain more objective scoring methods.

ACKNOWLEDGMENT

Hui Wang acknowledges financial support from the China Scholarship Council (CSC), CSC No.201706990015. We thank the ADA Research Group in LIACS (especially Holger Hoos and Marie Anastacio) for providing computational resources.

REFERENCES

[1] Silver D, Huang A, Maddison C J, Guez A, Sifre L, van den Driessche G, et al, ”Mastering the game of Go with deep neural networks and tree search”, Nature. vol. 529(7587), pp. 484–489, 2016.

[2] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, et al, ”Mastering the game of go without human knowledge”, Nature. vol. 550(7676), pp. 354–359, 2017.

[3] Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, et al, ”A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”, Science. vol. 362(6419), pp. 1140–1144, 2018.

[4] Granter S R, Beck A H and Papke Jr D J, ”AlphaGo, deep learning, and the future of the human microscopist”, Archives of pathology & laboratory medicine. vol. 141(5), pp. 619–621, 2017.

[5] Wang F Y, Zhang J J, Zheng X, Wang X, Yuan Y, Dai X, et al, ”Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond”, IEEE/CAA Journal of Automatica Sinica. vol. 3(2), pp. 113–120, 2016.

[6] Fu M C, ”AlphaGo and Monte Carlo tree search: the simulation optimization perspective”, Proceedings of the 2016 Winter Simulation Conference. IEEE Press pp. 659–670, 2016.

[7] Segler M, Preuss M and Waller M, Planning chemical syntheses with deep neural networks and symbolic AI. Nature. vol. 555(7698), pp. 604-610, 2018.

[8] Tao J, Wu L and Hu X, ”Principle Analysis on AlphaGo and Perspective in Military Application of Artificial Intelligence”, Journal of Command and Control. vol. 2(2), pp. 114–120, 2016.

[9] Zhang Z, ”When doctors meet with AlphaGo: potential application of machine learning to clinical medicine”, Annals of translational medicine. vol. 4(6), 2016.

[10] Surag Nair: https://github.com/suragnair/alpha-zero-general. 2018.

[11] Iwata S and Kasai T, ”The Othello game on an n×n board is PSPACE-complete”, Theoretical Computer Science. vol. 123(2), pp. 329–340, 1994.

[12] Allis V, ”A knowledge-based approach of Connect-Four-the game is solved: White wins”. 1988.

[13] Schmidhuber J, ”Deep learning in neural networks: An overview”, Neural networks. vol. 61, pp. 85–117, 2015.

[14] Clark C and Storkey A, ”Training deep convolutional neural networks to play go”, International Conference on Machine Learning. pp. 1766–1774, 2015.

[15] Mnih V, Kavukcuoglu K, Silver D, Rusu A, Veness J, Bellemare M, et al, ”Human-level control through deep reinforcement learning”, Nature. vol. 518(7540), pp. 529–533, 2015.

[16] Heinz E A, ”New self-play results in computer chess”, International Conference on Computers and Games. Springer, Berlin, Heidelberg. pp. 262–276, 2000.

[17] Wiering M A, ”Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning”, Journal of Intelligent Learning Systems and Applications. vol. 2(2), pp. 57–68, 2010.

[18] Van Der Ree M and Wiering M, ”Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play”, In Adaptive Dynamic Programming And Reinforcement Learning. pp. 108–115, 2013.

[19] Bengio Y, Louradour J, Collobert R and Weston J, ”Curriculum learning”, Proceedings of the 26th annual international conference on machine learning. ACM, pp. 41–482009.

[20] Browne C B, Powley E, Whitehouse D, Lucas S, Cowling P, Rohlfshagen P, et al, ”A survey of monte carlo tree search methods”, IEEE Transactions on Computational Intelligence and AI in games. vol. 4(1), pp. 1–43, 2012.

[21] Wang H, Emmerich M, Preuss M and Plaat A, ”Hyper-Parameter Sweep on AlphaZero General”, arXiv preprint arXiv:1903.08129, 2019.

[22] De Jong K A and Schultz A C, ”Using experience-based learning in game playing”, Machine Learning Proceedings 1988. Morgan Kaufmann, pp. 284–290, 1988.

[23] Chong S Y, Tan M K and White J D, ”Observing the evolution of neural networks learning to play the game of Othello”, IEEE Transactions on Evolutionary Computation. vol. 9(3), pp. 240–251, 2005.

[24] Edelkamp S and Kissmann P, ”On the complexity of BDDs for state space search: A case study in Connect Four”, Twenty-Fifth AAAI Conference on Artificial Intelligence. 2011.

[25] Thill M, Bagheri S, Koch P and Konen W, ”Temporal difference learning with eligibility traces for the game connect four”, 2014 IEEE Conference on Computational Intelligence and Games. IEEE, pp. 1–8, 2014.

[26] Banerjee B and Stone P, ”General Game Learning Using Knowledge Transfer”, International Joint Conference on Artificial Intelligence. pp. 672–677, 2007.

[27] Wang H, Emmerich M and Plaat A, ”Assessing the Potential of Classical Q-learning in General Game Playing”, arXiv preprint arXiv:1810.06078, 2018.

[28] Ioffe S and Szegedy C, ”Batch normalization: accelerating deep network training by reducing internal covariate shift”, Proceedings of the 32nd International Conference on International Conference on Machine Learning. vol 37, pp. 448–456, 2015.

[29] Kingma D P and Ba J, ”Adam: A method for stochastic optimization”, arXiv preprint arXiv:1412.6980, 2014.

[30] Srivastava N, Hinton G, Krizhevsky A, Sutskever I and Salakhutdinov R, ”Dropout: a simple way to prevent neural networks from overfitting”, The Journal of Machine Learning Research. vol. 15(1), pp. 1929–1958, 2014.

[31] Albers P C H and Vries H, ”Elo-rating as a tool in the sequential estimation of dominance strengths”, Animal Behaviour. pp. 489–495, 2001.

[32] Coulom R, ”Whole-history rating: A Bayesian rating system for players of time-varying strength”, International Conference on Computers and Games. Springer, Berlin, Heidelberg, pp. 113–124, 2008.