Multiple Policy Value Monte Carlo Tree Search

Li-Cheng Lan[1] , **Wei Li**[1] , **Ting-Han Wei**[1] and **I-Chen Wu**[1,2]
[1]Department of Computer Science, National Chiao Tung University, Taiwan
[2]Pervasive Artificial Intelligence Research (PAIR) Labs, Taiwan
{sb710031, fm.bigballon, tinghan.wei}@gmail.com, icwu@csie.nctu.edu.tw

Abstract

Many of the strongest game playing programs use a combination of Monte Carlo tree search (MCTS) and deep neural networks (DNN), where the DNNs are used as policy or value evaluators. Given a limited budget, such as online playing or during the self-play phase of AlphaZero (AZ) training, a balance needs to be reached between accurate state estimation and more MCTS simulations, both of which are critical for a strong game playing agent. Typically, larger DNNs are better at generalization and accurate evaluation, while smaller DNNs are less costly, and therefore can lead to more MCTS simulations and bigger search trees with the same budget. This paper introduces a new method called the multiple policy value MCTS (MPV-MCTS), which combines multiple policy value neural networks (PV-NNs) of various sizes to retain advantages of each network, where two PV-NNs f_S and f_L are used in this paper. We show through experiments on the game NoGo that a combined f_S and f_L MPV-MCTS outperforms single PV-NN with policy value MCTS, called PV-MCTS. Additionally, MPV-MCTS also outperforms PV-MCTS for AZ training.

1 Introduction

Many of the state-of-the-art game playing programs in games such as Go, chess, shogi, and Hex used a combination of Monte Carlo tree search (MCTS) and deep neural networks (DNN) [Silver *et al.*, 2016; Silver *et al.*, 2018; Gao *et al.*, 2018]. MCTS is a heuristic best-first search algorithm and had been a major breakthrough for many games since 2006 [Kocsis and Szepesvári, 2006; Coulom, 2006; Browne *et al.*, 2012], especially for Go [Gelly and Silver, 2007; Enzenberger *et al.*, 2010]. Starting in 2015, DNNs were investigated for move prediction [Clark and Storkey, 2015; Tian and Zhu, 2015]. For a problem with large state spaces like Go, the power of DNNs to generalize for previously unseen states is critical. In addition, DNNs were found to be more accurate at evaluating positions than Monte Carlo rollout [Silver *et al.*, 2016]. MCTS, however, remains a critical component of strong game playing programs due to its ability to search ahead while balancing exploration and exploitation.

The choice of network size in an MCTS and DNN combined algorithm is a non-trivial decision. Common current practices tend to settle on a network size based on empirical experience; for the example of Go, most teams tend to settle on 256 filters with varying number of layers [Tian *et al.*, 2019; Pascutto, 2017], following the first example by AlphaGo [Silver *et al.*, 2016]. Empirical data shows that overall, the larger the network, the stronger the program tends to be under the same number of MCTS simulations [Pascutto, 2017]. Similarly, outside the context of Go, there is abundant evidence showing that in general, the larger the network, the better it will be at generalization, and the increased capacity of the network also leads to better learned representations [Hinton *et al.*, 2015; He *et al.*, 2016].

However, when creating a program that combines MCTS and DNN, given the same amount of computing resources, it is not a simple decision of training using the largest allowable network, since the number of MCTS simulations depends on the size of the network. A smaller network can be much faster and therefore search more states; given the asymptotic convergence guarantee of MCTS, it may be more favorable to spend the finite budget on performing more simulations, rather than using a larger DNN for a more accurate evaluation. When considering training following the AZ paradigm, this problem is critical, since millions of self-play records need to be generated.

To solve this dilemma, we consider taking advantage of both large and small networks by combining them together with MCTS, which we call multiple policy value MCTS (MPV-MCTS). Two pre-trained DNNs are used in this paper, typically of significantly different network sizes (i.e., consisting of different numbers of filters and layers). MPV-MCTS is a general method that does not depend on how the two DNNs are trained. In the method, each network grows its own best-first search tree. The two trees share the same action value, so intuitively, the small net helps avoid blind spots via its lookahead from a higher simulation count, while the large net provides more accurate values and policies.

We use a simplified variant of Go called NoGo [Müller, 2015] to demonstrate the idea. Experiments show that by combining supervised learning networks of different sizes, MPV-MCTS can reach an average of 418 Elo rating against

the baseline (HaHaNoGo), whereas a small network and a large network alone can only reach 277 and 309 Elo ratings, respectively. Compared with intermediate-sized single networks, MPV-MCTS is also stronger against the common baseline. MPV-MCTS can also improve the playing strength of separately trained AZ networks. Lastly, using equivalent training budgets, MPV-MCTS can accelerate AZ training by at least a factor of 2. Matchups between the MPV-MCTS method and those without achieves win rates of 56.6% and 51.2% for 2 times and 2.5 times the training budget, respectively. With the same training budget, AZ training with MPV-MCTS can be up to 252 Elo ratings stronger than those trained without.

We list two major contributions of this paper as follows:

1. The MPV-MCTS search tree generated with two differently sized DNNs f_S and f_L is stronger in playing strength than either net alone, given the same amount of computing resource usage.
2. With MPV-MCTS, AZ training can be performed more efficiently.

2 Background

2.1 Monte Carlo Tree Search

MCTS is a best-first tree search algorithm, typically consisting of the iterative process of: 1) traversing the search tree according to a specified selection algorithm to *select* a yet-to-be-evaluated leaf state 2) *evaluating* the leaf, and 3) *updating* the tree correspondingly with the evaluation result. Instead of following the minimax paradigm, MCTS averages the evaluation results in the subtree rooted at state s to decide on the state value $V(s)$. Monte Carlo sampling (often referred to as rollout) is a standard evaluation method; modern programs may use DNNs as state value approximators instead of rollout, or combine the two evaluation methods in some way. Most selection algorithms are designed to minimize the expected regret of sampling the state s by balancing exploration and exploitation. Examples of selection algorithms include UCB1 [Kocsis and Szepesvári, 2006] and PUCT [Rosin, 2011]. Both of these algorithms follow the general form:

$$a_t = \arg\max_a (Q(s,a) + u(s,a)), \tag{1}$$

where $Q(s,a)$ is the state action value of taking action a at s, and $u(s,a)$ is a bonus value to regulate exploration. For example, the bonus value of PUCT (used by AlphaGo) is:

$$u_{\text{PUCT}}(s,a) \propto P(s,a) \times \frac{\sqrt{N(s)}}{N(s,a)}, \tag{2}$$

where $N(s)$ and $N(s,a)$ are the simulation counts of s and taking the action a at s respectively; and $P(s,a)$ is the probability of a being the best action of s, which is referred to as the *prior probability*.

2.2 Policy Value MCTS

Policy Value MCTS (PV-MCTS) uses DNNs to provide a policy $p(a|s)$ and a value $v(s)$ for any given state s. The policy p is used for PUCT as $P(s,a)$ in Equation 2 during the selection phase, while $v(s)$ is used as the evaluation result to update the state value V of ancestor states of s. The particular implementation of PV-MCTS as used by AlphaGo consists of two separate networks, the policy networks (be used when a node become a branch node of MCTS) and value networks (be used when a node be selected). AlphaGo Zero combines the two as a policy value network (PV-NN) that outputs the prior probabilities and state value simultaneously.

Data from Silver et al. [2017] hinted that the advantage of search cannot be easily replaced by spending more effort on training better policy networks. More specifically, their experiments show that the output policy of PV-MCTS is about 2000 Elo ratings [Elo, 1978] stronger than simply using the policy output of the same PV-NN without search. This indicates that even with a strong DNN policy function, performance can be further improved significantly through search.

2.3 Combining Multiple Strategies in MCTS

There are several methods that have been proposed to combine multiple strategies during MCTS action value evaluation, such as Rapid Action Value Evaluation (RAVE) [Gelly and Silver, 2011], implicit minimax backups [Lanctot *et al.*, 2014], and asynchronous policy value MCTS (APV-MCTS) [Silver *et al.*, 2016]. Both RAVE and implicit minimax backup combine MCTS evaluation (rollout) with another evaluation; for the scope of this paper, we omit the details. Of these three algorithms, MPV-MCTS is most related to APV-MCTS. APV-MCTS is designed to combine both MCTS evaluations of value network and rollout. The combined strength of the value network and rollout can achieve more than 95% winning rate against using either one alone [Silver *et al.*, 2016], demonstrating that there is potential for better performance by combining MCTS evaluations of different strategies.

3 Method

3.1 Multiple Policy Value MCTS

In this subsection, we focus on explaining the scenario where the overall system consists of two PV-NNs f_S and f_L (small and large networks, respectively). Let b_S (b_L) be an assigned number of simulations, or budget, for which our method uses the network f_S (f_L) to grow its own search tree T_S (T_L). Our problem is then: Given s, f_S, f_L, b_S, and b_L, find a stronger policy:

$$\pi(s, (f_S, b_S), (f_L, b_L)),$$

such that $b_S \geq b_L$.

When a state is in both search trees, the two networks collaborate by sharing the same state value $V(s)$ and the same prior probability $P(s,a)$. For the scope of this paper, we use the following method, similar to APV-MCTS:

$$V(s) = \alpha V_S(s) + (1-\alpha) V_L(s), \text{ and} \tag{3}$$

$$P(s,a) = \beta p_S(a|s) + (1-\beta) p_L(a|s), \tag{4}$$

where $\alpha, \beta \in [0,1]$ are weight coefficients. Note that α and β can be set according to the accuracy of the values and prior probabilities. For example, the more accurate V_L and p_L are,

the smaller α, β should be. In the experiments, we set $\alpha = 0.5$, $\beta = 0$, following settings by APV-MCTS.

For each simulation during the search, we first choose either f_S or f_L, say f_S, then select a leaf state of T_S to evaluate, and then use the results to update the search tree. It is up to the user to design how the two networks take turns, as long as the budgets b_S and b_L are satisfied. We suggest several ways to take turn in section 5. In our experiments, for a given set of b_S and b_L, we simply randomly select b_S numbers between 1 and $b_S + b_L$, and perform a small net simulation for those iterations; for all other iterations, we use the larger net. This corresponds to line 1 in Algorithm 1.

We now consider how f_S and f_L contribute to the overall MPV-MCTS method, starting with f_S. Conceptually, the goal is to allow T_S to provide the overall MPV-MCTS with the benefits of lookahead search, where the tree balances between exploration and exploitation as it grows. This is the role of f_S because it is the faster of the two networks, and can therefore perform more simulations with the same amount of resource usage as f_L. This is also the reason why we assign b_S to be larger than b_L. During MPV-MCTS, for each simulation using f_S, a leaf state is selected following the PUCT algorithm using the simulation count of f_S as N. This corresponds to lines 4-6 in Algorithm 1.

Now we consider the role of f_L. For every simulation of f_L, we wish to identify and simulate the most critical states. While there are many ways to do so, we simply assume for simplicity that in the larger tree the yet-to-be-evaluated leaf with the higher visit count in T_S should be more important, and thus the leaf with the highest visit count in T_S is selected in each simulation. This corresponds to line 8 in Algorithm 1. There is a very rare special case, wherein the selected leaf may not yet be visited by f_S. In this case (line 9), we reselect a unevaluated leaf state (line 10) using PUCT for f_L instead (following Equation 2, but with $N_L(s)$ and $N_L(s,a)$).

Algorithm 1: MPV-MCTS Algorithm

Input: state s, networks f_S and f_L, budgets b_S and b_L
1 $list = \text{RandomlySelect}(b_S, b_S + b_L)$;
2 **for** $i \leftarrow 1$ **to** $b_S + b_L$ **do**
3 **if** i *in* $list$ **then**
4 $s_{leaf} =$ SelectUnevaluatedLeafStateByPUCT(T_S) ;
5 $(p, v) = f_S(s_{leaf})$;
6 Update$(T_S, s_{leaf}, (p, v))$;
7 **else**
8 $s_{leaf} =$ SelectUnevaluatedLeafStateByPriority(T_L) ;
9 **if** $N_S(s_{leaf}) = 0$ **then**
10 $s_{leaf} =$ SelectUnevaluatedLeafStateByPUCT(T_L) ;
11 $(p, v) = f_L(s_{leaf})$;
12 Update$(T_L, s_{leaf}, (p, v))$;
13 **end**
14 **end**

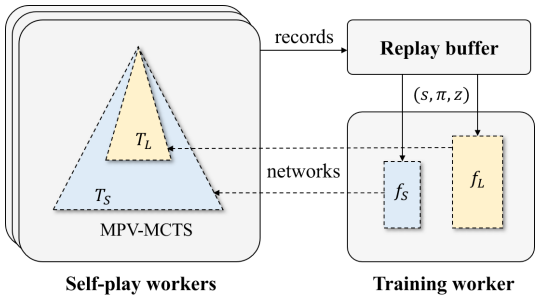


Figure 1: Workflow of training AZ with MPV-MCTS.

3.2 Training AlphaZero with MPV-MCTS

We first briefly review the AZ training method, in which an agent improves by playing against itself repeatedly. The agent generates its playing policy by using a PV-NN in a PV-MCTS algorithm. The weights in the PV-NN are first initialized randomly. We now describe the training process by listing three major components.

- By using the PV-NN in a PV-MCTS algorithm, *self-play workers* generate game records by letting two same instances of itself play against each other. At each turn in a game, a worker computes PV-MCTS and follows the playing policy as follows. The probability of a being played at s is $\pi_a \propto N(s,a)^{1/\tau}$, where $N(s,a)$ is the action's simulation count and τ is a temperature parameter. When a game ends, the self-play record is saved to the replay buffer.
- The *replay buffer* is a fixed-size queue containing a collection of self-play game records. Each record (s, π, z) includes a state s (of which there will be many in a single game), the playing policy π used to select the action at state s, and the outcome z of the game.
- The *training worker* continually samples records from the replay buffer and trains the current PV-NN.

Following the above AZ training and from the intuition that larger DNNs tend to learn better, it is reasonable to expect the trained agent using f_L should outperform that with f_S. That said, if our hypothesis for MPV-MCTS holds, it is also possible that replacing PV-MCTS with MPV-MCTS in AZ training will lead to better performance.

Figure 1 shows the work-flow of how MPV-MCTS can be applied to the AZ training algorithm. We start from two PV-NNs of different sizes with random weights. The replay buffer does not need to be modified. The self-play workers use the latest PV-NNs of both sizes with MPV-MCTS to generate the self-play records. After performing MPV-MCTS, we simply use the simulation count of the small net tree T_S as our playing policy π to select actions. When a game ends, we store the records to the replay buffer. The training worker repeatedly use records sampled from the replay buffer to train both f_S and f_L using the same loss function as AlphaGo Zero [Silver *et al.*, 2017].

4 Experiments

In this section, we empirically demonstrate our method on NoGo, a two-player game on a 9×9 Go board. NoGo was selected as the feature game of the 2011 Combinatorial Games Workshop at BIRS, and subsequently became a tournament item in the Computer Olympiad [Müller, 2015]. The game has the same rules as Go, except where moves that lead to capturing and suicide are not allowed. The player who cannot make any moves during her turn loses. NoGo is chosen for our experiments since the complexity of the game is relatively low when compared with Go, while maintaining many similar characteristics [Chou *et al.*, 2011].

4.1 Experiment Settings

In our experiments, the state-of-the-art NoGo program HaHaNoGo [Lan, 2016] (MCTS-based) with 100,000 simulations served as the baseline. HaHaNoGo defeated the reigning champion HappyNoGo (which placed first in the 2013 and 2015 Computer Olympiad) in TAAI 2016. In all experiments, each version played 1,000 games against the baseline. We use the Elo rating [Elo, 1978] instead of win rate to judge the playing strength of AI by setting the Elo rating of HaHaNoGo to 0. In this paper, all experiments are performed on eight Intel Xeon(R) Gold 6154 CPUs and 64 Nvidia Tesla V100 GPUs.

The architecture of our PV-NNs is the same as that used in AlphaGo Zero [Silver *et al.*, 2017], except for filter sizes, residual blocks [He *et al.*, 2016] and inputs. Let $f(x, y)$ denote a PV-NN with x filters and y residual blocks. The input to the network is a $9 \times 9 \times 4$ image stack comprised of four binary feature planes for the board information (player's stones, opponent's s, player's legal moves, opponent's moves) representing game states.

For fairness of comparison, let one unit of normalized budget indicate the amount of computing resources consumed for one single forward pass on network $f_{128,10}$ on the environment of the above setting. Note that one forward pass on $f(a \times x, b \times y)$ ideally runs about $a^2 \times b$ times slower than $f(x, y)$. Thus, for computing resource analysis, one single forward pass on $f_{64,5}$ is said to consume $1/8$ normalized budget (ignoring the cost of selection and update). Thus, given the same amount of normalized budget B, B forward passes can be performed on $f_{128,10}$, and $8B$ on $f_{64,5}$.

4.2 Evaluating MPV-MCTS

Combining Supervised Learning Networks

First, we investigate using PV-NNs that were trained with supervised learning in MPV-MCTS. We trained both $f_{64,5}$ and $f_{128,10}$ from a dataset of 200,000 games (about 10^7 positions) generated by HaHaNoGo with 50,000 simulations for each move via self-play. We will present the performance of MPV-MCTS with different budget allocation schemes, compared to single networks using PV-MCTS.

With a normalized budget of B, we combine $f_{64,5}$ and $f_{128,10}$ with the MPV-MCTS algorithm by different allocation schemes according to a budget ratio $r \in [0, 1]$, where the normalized budget is rB for $f_{128,10}$ and $(1 - r)B$ for $f_{64,5}$.

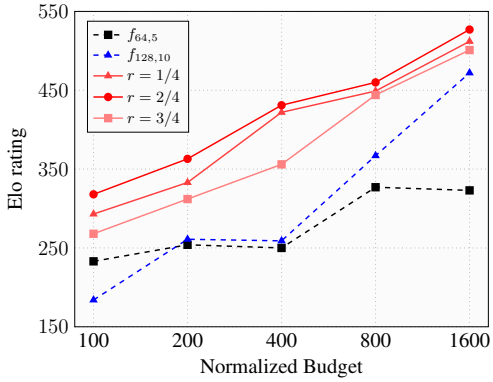


Figure 2: The Elo ratings for different kinds of (normalized) budget allocation. For a given budget B, rB is allocated to $f_{128,10}$, while the remaining $(1 - r)B$ is allocated to network $f_{64,5}$. When $r = 1$, it is equivalent to PV-MCTS with $f_{128,10}$ with simulation count B; when $r = 0$, it is equivalent to $f_{64,5}$ with simulation count $8B$.

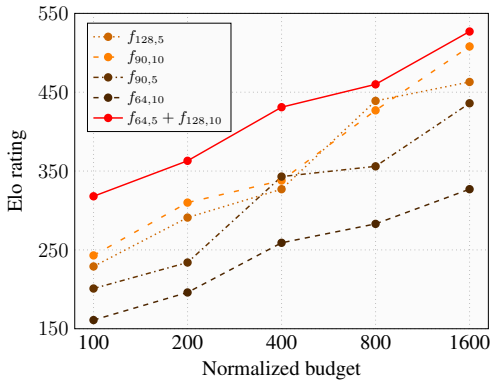
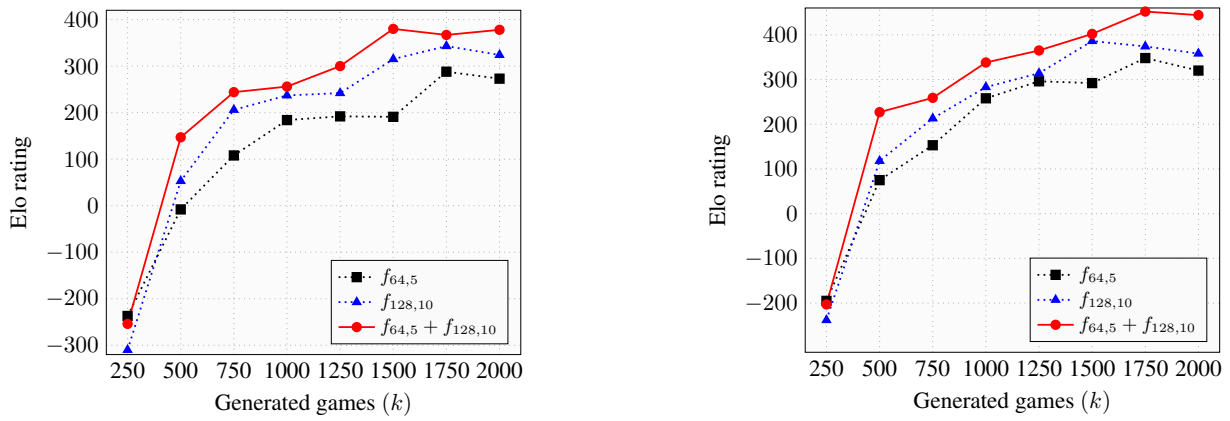


Figure 3: The Elo ratings of intermediate-sized networks with PV-MCTS vs. $f_{64,5} + f_{128,10}$ with MPV-MCTS.

Figure 2 shows the result of different resource allocation schemes, showing that combining two PV-NNs leads to better performance. The strongest version in these experiments is $r = 2/4$, which achieved 527 Elo rating (about 95.40% win rate against the baseline) with a normalized budget of 1600. While the version that only uses either the small or the large network can only achieve 323 and 472 Elo rating at best, respectively. With the same normalized budget, both PV-NNs $f_{64,5}$ and $f_{128,10}$ alone are weaker than all three allocation ratios of MPV-MCTS. In the rest of the experiment, we set $r = 2/4$.

We also trained intermediate-sized PV-NNs, $f_{128,5}$, $f_{90,10}$, $f_{90,5}$, and $f_{64,10}$, which are expected to be more accurate than smaller PV-NNs (e.g., $f_{64,5}$) and faster than larger PV-NNs (e.g., $f_{128,10}$). We compared these mid-sized PV-NNs using PV-MCTS with the best performing set in Figure 2 (i.e., MPV-MCTS with $f_{64,5} + f_{128,10}, r = 2/4$) with the same normalized budget. Figure 3 shows that MPV-MCTS outperforms all of the mid-sized PV-NNs with PV-MCTS.



(a) Testing with a normalized budget of 200.

(b) Testing with a normalized budget of 800.

Figure 4: Combining PV-NNs trained by AZ with MPV-MCTS, where each red dot is the combination of the blue and black dots in the same column.

Combining AlphaZero Trained Networks

Since MPV-MCTS is comprised of pre-trained PV-NNs, we now investigate using various AZ trained PV-NNs in pairs. We trained both $f_{64,5}$ and $f_{128,10}$ using the following settings: simulation count: 800, PUCT constant: 1.5, learning rate: 0.05, batch size: 1024.

After training, $f_{64,5}$ and $f_{128,10}$ will each have a progression of PV-NNs as the accumulated number of self-play games approaches 2,000,000. We select eight pairs of large and small PV-NNs, along the following total number of accumulated self-play games: 250k, 500k, ... , 2000k. That is, in Figure 4, the red data points refer to the MPV-MCTS consisting of $f_{64,5}$ and $f_{128,10}$, each trained following AZ with 250k accumuluated self-play game records. Figure 4 also presents the strength of each pair of large and small PV-NNs using the same normalized budget for testing. For Figure 4a, a normalized budget of 200 was used, while for Figure 4b, a normalized budget of 800 was used. Both results show that MPV-MCTS outperforms both the large and small nets, at all stages throughout the AZ training process. This also shows the robustness of MPV-MCTS.

4.3 AlphaZero Training with MPV-MCTS

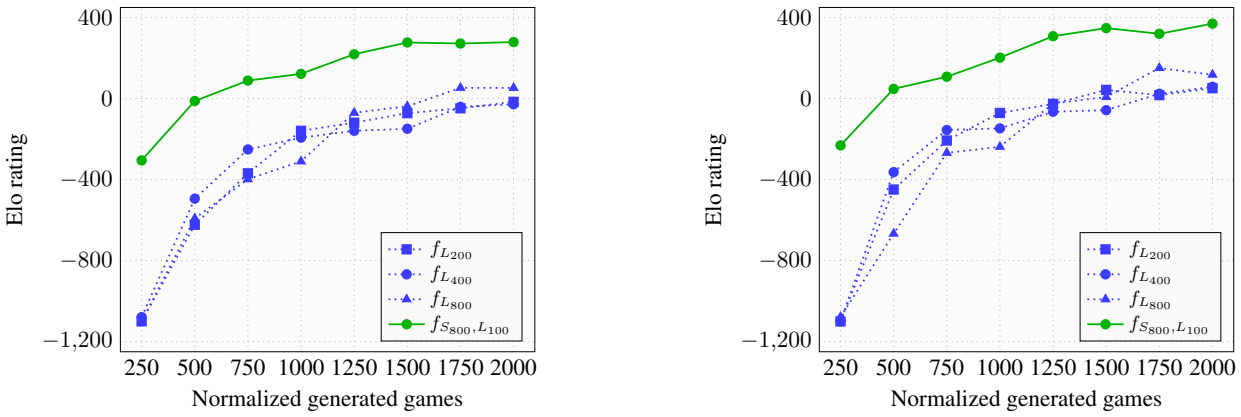
In this subsection, we investigate performing AZ training with MPV-MCTS, as illustrated in Figure 1. Let f_S be $f_{64,5}$, and f_L be $f_{128,10}$ for clarity of presentation. Following the workflow in Figure 1, both f_L and f_S are trained together; self-play workers use both PV-NNs to generate self-play games, using the playing policy $\pi(s, (f_S, 800)), (f_L, 100)$.

Next, we follow AZ training for three separate large networks, where the difference is that during self-play, each move is generated with simulation counts of 200, 400, and 800. These three are denoted as $f_{L_{200}}$, $f_{L_{400}}$, and $f_{L_{800}}$, respectively. Since we wish to fix the total training resource usage, we will need to define a *normalized game generation count*, similar to the way we defined a normalized budget. Since the version with 800 simulations per self-play move theoretically spends four times approximately as much resources as the version with 200 simulations per self-play, the former will generate $1/4$ as many self-play game records given the same training budget. For this experiment, we define 1 normalized generated game to use the same amount of training budget as generating 1 self-play game record for $f_{128,10}$ using 200 simulations per move. Note that in the MPV-MCTS case (where f_S and f_L are trained with 800 and 100 simulations per move respectively), generating 1 actual self-play game record is equivalent in terms of cost as 1 normalized generated game.

Figure 5 presents the results of training AZ with MPV-MCTS compared with PV-MCTS. Since our main goal is to demonstrate the benefit of using MPV-MCTS for training, the testing conditions should be as equal across all sets versions as possible. Therefore, although trained with different simulations, each agent with the trained large net uses 200 (Figure 5a) or 800 (Figure 5b) simulations per move for testing. For the MPV-MCTS case, we use only f_L during testing for comparison; f_S was omitted from testing so that the comparison is strictly between large PV-NNs.

The results show that our method outperforms the best large net following AZ training with PV-MCTS (i.e., $f_{L_{800}}$) by average 238 Elo ratings with a total of 2000k normalized generated self-play games. Our interpretation is that the quality (how good is the policy during self-play) and quantity of the self-play game records are two key factors during AlphaZero training. Comparing with $f_{L_{800}}$, MPV-MCTS will have more game records (more training steps). On the other hand, against $f_{L_{200}}$, MPV-MCTS will have higher quality game records.

The results for $f_{L_{800}}$ in Figure 5 corresponds to the blue dataset in Figure 4, where the self-play simulation (for training) was also set to be 800. If we look at the MPV-MCTS training case (the green dataset in Figure 5), the result at 2000k normalized generated games performs 279 Elo ratings and 370 Elo ratings stronger than the baseline, for 200 and 800 simulations respectively (Figures 5a and 5b). Comparing between the data for MPV-MCTS in Figure 5b and $f_{128,10}$ in Figure 4b, the performance for MPV-MCTS still exceeds that of $f_{128,10}$ for 1000k (4000k normalized) and 1250k gen-



(a) Testing with a normalized budget of 200.

(b) Testing with a normalized budget of 800.

Figure 5: AZ training with MPV-MCTS. For a fair comparison, the Elo ratings were obtained by performing PV-MCTS using equally sized large networks f_L for all cases. That is, f_S was only used for the green dataset during training, and was omitted for testing.

erated games (5000k normalized). In fact, matches between $f_{S_{800},L_{100}}$ at 2000k normalized generated games and $f_{L_{800}}$ at 1000k generated games yields a win rate of 56.6%, while playing against $f_{L_{800}}$ at 1250k generated games yields a win rate of 51.2%. This implies that MPV-MCTS accelerates AZ training by at least factor of 2.

5 Discussion and Future Work

In this section, we discuss some settings that we only performed partial experiments on, and share preliminary results.

First, instead of two separate trees, we can think of our method as having just one single tree, with the small net's tree as the "main" tree, because it is usually the larger one. Nonetheless, we describe MPV-MCTS as two trees for generality (thereby allowing the algorithm to work with more than two nets) and simplicity.

Second, in line 8 of Algorithm 1, we used the simulation count to select the most important yet-to-be-evaluated states for f_L. Other types of priority functions can be used instead when selecting states for f_L, such as following its own PUCT, or adding a discounted bonus to states whose parent has higher visit counts. For situations with limited budgets, the former tends to be weaker than our current priority function (but still stronger than a single net), while the latter is complicated but yields no improvement. For multiple simultaneous networks, only the smallest network (with the largest tree) uses PUCT for selection. All other networks would use a user-defined priority function instead.

Third, instead of randomly selecting the order to take turns simulating with f_S and f_L, as described in subsection 3.1 and line 1 of Algorithm 1, we have also tried some other settings. Assume that the budget allocation is $b_S = 800, b_L = 100$; one alternative method is to force MPV-MCTS to start with the larger net (say, the first 50 simulations all use f_L) so that the small net has more information that guides the search when it begins. Results show that this alternative method of taking turns does not yield improvement in testing, but seem to have beneficial effects when used in AZ training (as in subsection 3.2). We also tried round-robin, but no significant difference was observed.

Fourth, our method provides an opportunity for ensemble learning. For example, we tried to increase the coefficient of the mean square error in the small net's loss function, with the aim of improving the accuracy of its value function. Results show that training AZ with MPV-MCTS in this way accelerates the process of training.

Fifth, we trained a mini-net (even weaker than the small net) to replace the small net. The results of 800 mini network simulations + 100 large network simulations were comparable to 800 small network simulations + 100 large network, despite the mini network being obviously worse than the small network. This seems to imply that the MPV-MCTS benefits from the look-ahead search provided by the smaller "partner" more than the quality of the smaller partner.

Finally, we believe that a better way to train the AZ algorithm with MPV-MCTS is to train with the help of several sizes of networks. In this scenario, the largest network is the primary network. The training begins by using the smallest network as the support network, following the training process as the one described in subsection 3.2. As the large network improves and the representation learned by the small network is no longer sufficient to master the training data, we can replace it with a larger support network. This process is iterative; while the primary network is persistent, the support network should increase in capacity whenever it is unable to keep up. The decision of using which support networks and the parameters such as α, β, and simulation count can be controlled by meta-learning, which is left as an open problem.

Acknowledgments

This research is partially supported by the Ministry of Science and Technology (MOST) under Grant Number MOST 107-2634-F-009-011 and MOST 108-2634-F-009-011 through Pervasive Artificial Intelligence Research (PAIR) Labs, Taiwan. The computing resource is partially supported by National Center for High-performance Computing (NCHC).

References

[Browne *et al.*, 2012] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[Chou *et al.*, 2011] C-W Chou, Olivier Teytaud, and S-J Yen. Revisiting monte-carlo tree search on a normal form game: Nogo. In *European Conference on the Applications of Evolutionary Computation*, pages 73–82. Springer, 2011.

[Clark and Storkey, 2015] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pages 1766–1774, 2015.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[Elo, 1978] Arpad E Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978.

[Enzenberger *et al.*, 2010] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[Gao *et al.*, 2018] Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for monte carlo tree search. In *IJCAI*, pages 3762–3768, 2018.

[Gelly and Silver, 2007] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[Gelly and Silver, 2011] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[He *et al.*, 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Hinton *et al.*, 2015] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[Lan, 2016] Li-Cheng Lan. Hahanogo: An open source nogo program. https://github.com/lclan1024/HaHaNoGo, 2016.

[Lanctot *et al.*, 2014] Marc Lanctot, Mark HM Winands, Tom Pepels, and Nathan R Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[Müller, 2015] Martin Müller. Nogo history and competitions. https://webdocs.cs.ualberta.ca/~mmueller/nogo/history.html, 2015.

[Pascutto, 2017] G.-C. Pascutto. Leela-zero. https://github.com/gcp/leela-zero, 2017.

[Rosin, 2011] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[Silver *et al.*, 2017] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[Silver *et al.*, 2018] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Tian and Zhu, 2015] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015.

[Tian *et al.*, 2019] Yuandong Tian, Jerry Ma*, Qucheng Gong*, Shubho Sengupta*, Zhuoyuan Chen, James Pinkerton, and C. Lawrence Zitnick. Elf opengo: An analysis and open reimplementation of alphazero. *CoRR*, abs/1902.04522, 2019.