2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)

Learning of Evaluation Functions via Self-Play Enhanced by Checkmate Search

Taichi Nakayashiki
Graduate School of Arts and Sciences
the University of Tokyo
Tokyo, Japan
tnakayashiki@g.ecc.u-tokyo.ac.jp

Tomoyuki Kaneko
Graduate School of Interdisciplinary Information Studies
the University of Tokyo
Tokyo, Japan
kaneko@acm.org

***Abstract*—As shown in *AlphaGo*, *AlphaGo Zero*, and *AlphaZero*, reinforcement learning is effective in learning of evaluation functions (or value networks) in Go, Chess and Shogi. In their training, two procedures are repeated in parallel; self-play with a current evaluation function and improvement of the evaluation function by using game records yielded by recent self-play. Although *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* have achieved super human performance, the method requires enormous computation resources. To alleviate the problem, this paper proposes to incorporate a checkmate solver in self-play. We show that this small enhancement dramatically improves the efficiency of our experiments in Minishogi, via the quality of game records in self-play. It should be noted that our method is still free from human knowledge about a target domain, though the implementation of checkmate solvers is domain dependent.**

***Index Terms*—Machine Learning, Neural Network, Minishogi**

I. INTRODUCTION

Recently, *AlphaGo* outperformed human experts in the game of Go [1]. *AlphaGo Zero* improved its playing strength and established a sophisticated reinforcement learning without any human data or domain knowledge [2], and *AlphaZero* defeated one of the strongest computer programs in the game of Chess and Shogi (Japanese chess) [3] with almost the same method as *AlphaGo Zero*. The performance of *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* was supported by deep neural networks with many convolutional layers, trained by millions of self-play records. However, it is not easy for ordinary researchers to train such neural networks with their methods because it needs enormous computation resources to produce a large number of game records.

To accelerate the training of such neural networks within ordinary computational resources, we present to incorporate checkmate solvers in self-play. For successful training, the quality of game records in self-play is also crucial. To obtain reliable game records, 800 simulations (playouts) were performed in Monte-Carlo Tree Search (MCTS) for each move in self-play in *AlphaZero*. Our incorporation of checkmate solvers contributes to the quality of game records via a different way from MCTS. By incorporation of checkmate solvers, the best move is immediately played if the solver finds the checkmate sequence in a given position in self-play. This enhancement is effective especially for the beginning of the training because the parameters of neural networks are initialized with random values and gradually improved so that the network predicts the winner of a position as well as the move played in game records yielded by self-play. Therefore, without our enhancement, games played by self-play consist of almost random moves at the beginning of the training, due to the randomized initialization. It should be noted that this enhancement can be implemented only by the rule of the game, without human knowledge or playing experiences.

We demonstrated the effectiveness of our enhancement in experiments in Minishogi. Shogi is a major Chess variant in Japan [4], and was chosen as one of the three challenges tackled by *AlphaZero* project [3], in addition to Chess and Go. Minishogi is a small variant of Shogi. Its board size is 5×5 and only six type of pieces are used. We chose Minishogi for our experiments because our computational resources were not sufficient for *AlphaZero*'s style training of Shogi. Although the game tree complexity of Minishogi is smaller than that of Shogi, it still keeps sufficient complexity by inheriting promotion and dropping rules from Shogi. A notable property in Minishogi and Shogi is that endgame databases are not available, because the number of active pieces never decreases by the dropping rule. Instead, a specialized checkmate solver is effective to identify a checkmate sequence.

II. RELATED WORK

A. Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) is a general tree search algorithm [5], which is effective in game-tree search in Go as well as Chess and Shogi [1], [3].

MCTS iteratively conducts playouts and the confidence in the best action is gradually improved as the number of playouts increases. Each playout consists of four steps; traversal, expansion, evaluation, and backpropagation.

1) Traversal: it traverses the search tree from the root to a leaf, by repeatedly selecting a child having maximum indicator described later. We denote a leaf reached by this step s_L.
2) Expansion: the leaf s_L is expanded and their child nodes are connected to the search tree.

A part of this work was supported by JSPS KAKENHI Grant Number 16H02927 and by JST, PRESTO.

2376-6824/18/$31.00 ©2018 IEEE
DOI 10.1109/TAAI.2018.00036

3) Evaluation: the leaf s_L is evaluated by a rollout (with random moves) in traditional MCTS, or by value networks in *AlphaGo Zero* and *AlphaZero*.
4) Backpropagation: the evaluation is propagated from the leaf node s_L to the root node through the parents, so that each node has the average of all evaluations measured for its sub-tree.

B. Application of Neural Networks into Monte-Carlo Tree Search

We briefly introduce how neural networks are utilized in MCTS in *AlphaGo Zero* [2] and *AlphaZero* [3]. Let s be a position and a be a legal move. We denote a neural network with parameter θ by $f_\theta(s) = (\boldsymbol{p}_s, v_s)$, that computes probability distribution over moves $\boldsymbol{p}$ and scalar evaluation v_s that estimates the current player winning probability, for given position s. Each node in the search tree in MCTS stores $\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$, where $N(s,a)$ is the visit count, $W(s,a)$ is the total action-value, $Q(s,a)$ is the mean action-value, and $P(s,a)$ is the prior probability of selecting that edge that are obtained in the output of neural networks $\boldsymbol{p}$. In the traversal step, action $a_t = \arg\max_a(Q(s_t,a) + U(s_t,a))$ is selected, where

$$Q(s,a) = \frac{\sum_{s'|s,a\to s'} v_{s'}}{N(s,a)},$$
$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)},$$

$s, a \to s'$ indicates that s' is a reached leaf position in the traversal step with move a in the position s, and c_{puct} is a constant determining the level of exploration and was 5 in our experiments. In the evaluation step, the output of neural networks v_{s_L} is used for the evaluation of the leaf s_L.

In addition to the best move, the probability for each move at the root, $\boldsymbol{\pi}$, can be obtained with the visit counts in MCTS, by setting each element of $\boldsymbol{\pi}$ be proportional to $N^{1/\tau}$ where τ is a parameter controlling temperature. Probability $\boldsymbol{\pi}$ is stored and utilized in training of neural networks.

III. METHOD

We present to incorporate checkmate solvers into the training framework of *AlphaZero* [3]. For completeness, we briefly introduce the training framework of *AlphaZero*, as well as our enhancement. The framework consists of two components; self-play and optimization of the parameters in neural networks. Both of them run in parallel.

A. Self-Play

Game records are continuously produced by self-play between agents with the latest parameters of neural networks. In *AlphaZero*, MCTS with a fixed number of playouts determines a move in self-play. In our method, the agent invokes a checkmate solver at a given position, and if a checkmate sequence is found, the agent immediately selects the move to play. Otherwise, the agent conducts MCTS as in *AlphaZero*. The checkmate solver conducts depth first search with iterative deepening.

In self-play, the probability distribution over legal moves $\boldsymbol{\pi}$ is computed by visit counts and stored for each state. For a state that a checkmate sequence was found, the probability of the winning move and other moves were set to 1 and 0, respectively.

The details are as follows: For each position, the agent searches 1 600 simulations in MCTS to determine a move to play. For concurrency in MCTS, 32 leaves are evaluated at once in a mini-batch. To keep variation in leaves in a mini-batch, virtual loss [6] is introduced. To keep sufficient variation in game records, Dirichlet noise is added to the prior probabilities in the root node in advance; $\epsilon = 0.25$ and $\boldsymbol{\eta} \sim \mathrm{Dir}(0.03)$ [1]. The search tree in MCTS is not inherited in the search at the next position after a move played. The temperature parameter in MCTS τ is 1 for the first 20 plies in a game. We chose 20 instead of 30 to handle a relatively short game length in Minishogi. After that, $\tau = 0$ is used for the rest of the game.

B. Optimization

The parameters in neural networks, θ, are trained by using the same loss function $l(\theta)$ as *AlphaGo Zero* and *AlphaZero*:

$$(\boldsymbol{p}_s, v_s) = f_\theta(s),$$
$$l(\theta) = \sum_s \left\{(z_s - v_s)^2 - \boldsymbol{\pi}_s^{\mathrm{T}} \log \boldsymbol{p}_{\boldsymbol{s}}\right\} + c\|\theta\|^2$$

where $\boldsymbol{p}_s$ represents a probability distribution over moves, v_s is a scalar evaluation estimating the winning probability of current player, s is a position, $f_\theta(s)$ is a deep neural network with parameters θ, $z \in \{-1, 1\}$ and $\boldsymbol{p}$ are the winner and the probability distribution for legal moves, respectively, recorded in the game records, and c is a parameter controlling the strength of L2 weight regularization.

C. Neural Network Architecture

We describe how we apply the training framework in Minishogi. We basically followed the architecture of neural networks *AlphaZero* in Shogi. The differences are the input to the neural network, the number of residual blocks, and a full connection layer we added before the policy head described later.

Specifically, the input to the neural network is a $5 \times 5 \times 31$ image stack. The first 20 planes are binary feature planes indicating each type of pieces on the board, where the first 10 planes are for white pieces and the others are for black pieces. The next 10 planes indicate the number of captured pieces of each type, where the first 5 planes are for white's prisoners and the others are for black's prisoners. Finally, the last one plane indicates the player to move in a given position. Although *AlphaGo Zero*'s input of Shogi was oriented to the perspective of the current player, we fixed white and black pieces of input to train a neural network from raw positions. Table I summarizes the input layer.

TABLE I
INPUT LAYER OF NEURAL NETWORK IN MINISHOGI

Feature	Planes
White pieces	10
Black pieces	10
White's prisoners	5
Black's prisoners	5
The current player	1
Total	31

The output of the neural network consists of a vector of $1\,500$ elements (policy) and scalar in the range $[-1, 1]$ (value). The first $(5 \times 5) \times (5 \times 5) \times 2 = 1\,250$ elements of policy represent an ordinary move; all combination of the origin and destination of a move, and promotion. Additional $10 \times 5 \times 5 = 250$ elements of policy represent a dropping move; all combination of piece type and destination.

The architecture of neural networks after the input layer as follows:

The input features are processed by a residual tower that consists of a single convolutional block followed by 10 residual blocks, though *AlphaGo Zero* used 19 or 39 residual blocks. This difference is because of computational resources.

The convolutional block applies the following modules:

1. A convolution of 256 filters of kernel size 3×3
2. Batch normalization
3. ReLU

Each residual block applies the following modules sequentially to its input:

1. A convolution of 256 filters of kernel size 3×3
2. Batch normalization
3. ReLU
4. A convolution of 256 filters of kernel size 3×3
5. Batch normalization
6. A skip connection that adds the input to the block
7. ReLU

The output of the residual tower is passed into two separate heads for computing the policy and value respectively. The policy head applies the following modules:

1. A convolution of 2 filters of kernel size 1×1
2. Batch normalization
3. ReLU
4. A fully connected layer to a hidden layer of size 256
5. ReLU
6. A fully connected layer outputs a vector of size $1,500$

The value head applies the following modules:

1. A convolution of 1 filter of kernel size 1×1
2. Batch normalization
3. ReLU
4. A fully connected layer to a hidden layer of size 256
5. ReLU
6. A fully connected layer to a scalar
7. A tanh layer outputting a scalar in the range $[-1, 1]$

Each convolution layer was processed with stride 1.

As a result, the number of parameters in our neural networks in Minishogi was about 1.2% of that of *AlphaGo Zero*.

IV. EXPERIMENTS

We conducted the training of deep neural networks in Minishogi, by following the procedures in *AlphaZero*, with or without our enhancement of checkmate solvers. To compare the effect of checkmate solvers, we trained two neural networks, one was trained by self-play with checkmate solvers of depth 7 (agent A), and the other was trained by self-play without checkmate solvers (agent B).

A checkmate solver was executed in parallel at the root node of MCTS while MCTS was waiting for a GPU to get policy and value output. The checkmate solver had little effect on thinking time of MCTS, and MCTS took about 1.5 seconds of thinking time per move with and without the checkmate solver.

A. Training

At the beginning, the parameters of neural networks were randomly initialized, and $6\,000$ game records were generated by self-play with those networks. Self-play was conducted in parallel by using multiple GPUs. It took about 9 hours for agent A (with checkmate solvers) with six Geforce GTX 1080s, and about 17 hours for agent B (without checkmate solvers) with six Geforce GTX 1080s. This large difference in elapsed time can be explained by the difference in the average game length for the two agents:

- Agent A: 22.4 plies / record, and
- Agent B: 36.1 plies / record.

It is natural that the average game length is reduced by incorporation of checkmate solvers.

After that, we conducted $3\,000$ minutes of experiments, where self-play and optimization of network parameters were conducted in parallel. In $3\,000$ minutes, neural network parameters were updated from about $25\,100$ mini-batches for agent A, and about $22\,500$ mini-batches for agent B, and the number of games played by agent A and agent B was $4\,797$, $2\,361$, respectively, in self-play.

The parameters in neural networks were optimized by stochastic gradient descent with momentum. The learning rate was 0.01, the momentum parameter was 0.9 and the L2 weight regularization parameter c was 10^{-4}. Each mini-batch consists of $1\,024$ positions, sampled uniformly at random from all positions from the most recent $6\,000$ games of self-play, while in *AlphaGo Zero*, they were sampled from the most recent $500\,000$ games.

We used a single PC with two GPUs (GeForce GTX 1080) for the training of each agent and assigned one GPU for each task of self-play and training of its neural network.

We used the standard Minishogi rule [7], except for repetition of check moves. In the rule, the repetition of check moves is not allowed more than three times, but in our experiments, it was regarded as just repetition, which is also known as *sennitite*, Black's win.

If a game goes more than 256 plies, the game was terminated and treated as a draw. Such games were not frequently occurred (0% for agent A and 0.2% for agent B) and we believe that our results were not affected by these draws.

B. Results

Fig 1 shows the loss and the accuracy during the training. For given position s, the accuracy was calculated with the methods as follows:

1) For policy: The policy output is correct if $\arg\max_a P(s,a) = \arg\max_a \pi_{s,a}$.
2) For value: The value output is correct if $v_s \cdot z > 0$.

The policy accuracy was higher on agent A, though the value accuracy was about the same at the training time of 3 000 minutes. The difference can be explained by the quality of game records and also by a possibility that the neural network learned moves in checkmate sequences.

We adopted a fixed training time for both agents. Therefore, the number of training iterations was different between agent A and agent B, because of the difference in the average game length. In our experiments, although the update of neural network parameters and self-play run in parallel, the average game length affected the frequency of updates. We measured statistics of game length for each agent, after training started (i.e., excluding the initial 6 000 game records). The minimum, maximum, and average game length were 5, 94, 24.6, respectively for agent A, and 7, 256, and 43.5 for agent B.

Table II shows the result of games between agent A and agent B. The number of wins and losses was counted for agent A. Every 500 minutes of training time, 50 games were played between agent A and agent B. Each agent played 25 games with White and 25 games with Black. The same parameters of MCTS as self-play in training were used to incorporate randomness, except for both agents (including agent B) used checkmate solvers of depth 7 to compare the quality of neural networks in a fair manner. We can see that agent A with our enhancement outperformed agent B (baseline) in all time steps.

Table III shows the result of games between agent A with training time of every 500 minutes and agent B with training time of 3 000 minutes. The number of wins and losses was counted for agent A. We can see that agent A with our enhancement outperformed agent B after training of 3 000 minutes, even agent A was trained in only 1 000 minutes.

Fig 2 shows the win ratio shown in Tables II and III.

For further understanding of the learning efficiency, we also compared the strength between agent A and agent B for every 5 000 iterations. Table IV summarizes the results. We can see that agent A still outperformed agent B in this configuration.

TABLE II
RELATIVE STRENGTH OF AGENT A AGAINST AGENT B AT EACH TIME STEP

Elapsed (minutes)	Win	Loss
500	32	18
1000	30	20
1500	35	15
2000	35	15
2500	33	17
3000	36	14

TABLE III
RELATIVE STRENGTH OF AGENT A (EVERY 500 MINUTES) AGAINST AGENT B (AFTER 3000 MINUTES)

Elapsed (minutes)	Win	Loss
500	22	28
1000	35	15
1500	35	15
2000	27	23
2500	34	16
3000	36	14

TABLE IV
RELATIVE STRENGTH OF AGENT A AGAINST AGENT B AT EVERY 5000 ITERATIONS

Iterations	Win	Loss
5000	42	8
10000	27	23
15000	37	13
20000	35	15

To investigate whether the neural network learned the moves to checkmate, we investigated a position as an example. Fig 3 is the position which is white to move, and white can win by checkmate Black by one ply with dropping a white's prisoner of gold piece to e4. We investigated the policy output of the neural network. Agent A with training time 3 000 showed the third highest policy to the correct move, $P(s,a) \simeq 0.09$. On the other hand, agent B with training time 3 000 minutes showed the worst policy to the correct move, $P(s,a) \simeq 0.003$. Note that the checkmate move was not recommended by value networks in both agents. It is because checkmated positions were not included in training positions of neural networks.

V. CONCLUSION AND FUTURE WORK

We applied the procedures of *AlphaZero* into Minishogi, and successfully trained neural networks via self-play without human knowledge. Then, we presented to incorporate checkmate solvers in self-play in the training procedure, with which we can get absolutely correct labels in certain positions that contribute to a more efficient training of neural networks, without introducing additional human knowledge about a target game, at least in Minishogi.

In our experiments, we showed that checkmate solvers were extremely useful to accelerate the training of neural networks. The playing strength of neural networks trained with our enhancement outperformed those without the enhancement in self-play experiments.

This improvement, incorporating checkmate solvers, can be applicable to other finite two-person perfect-information games.

As a future work, it would be interesting to measure the effect of sophisticated checkmate solvers, by incorporating df-pn [8].

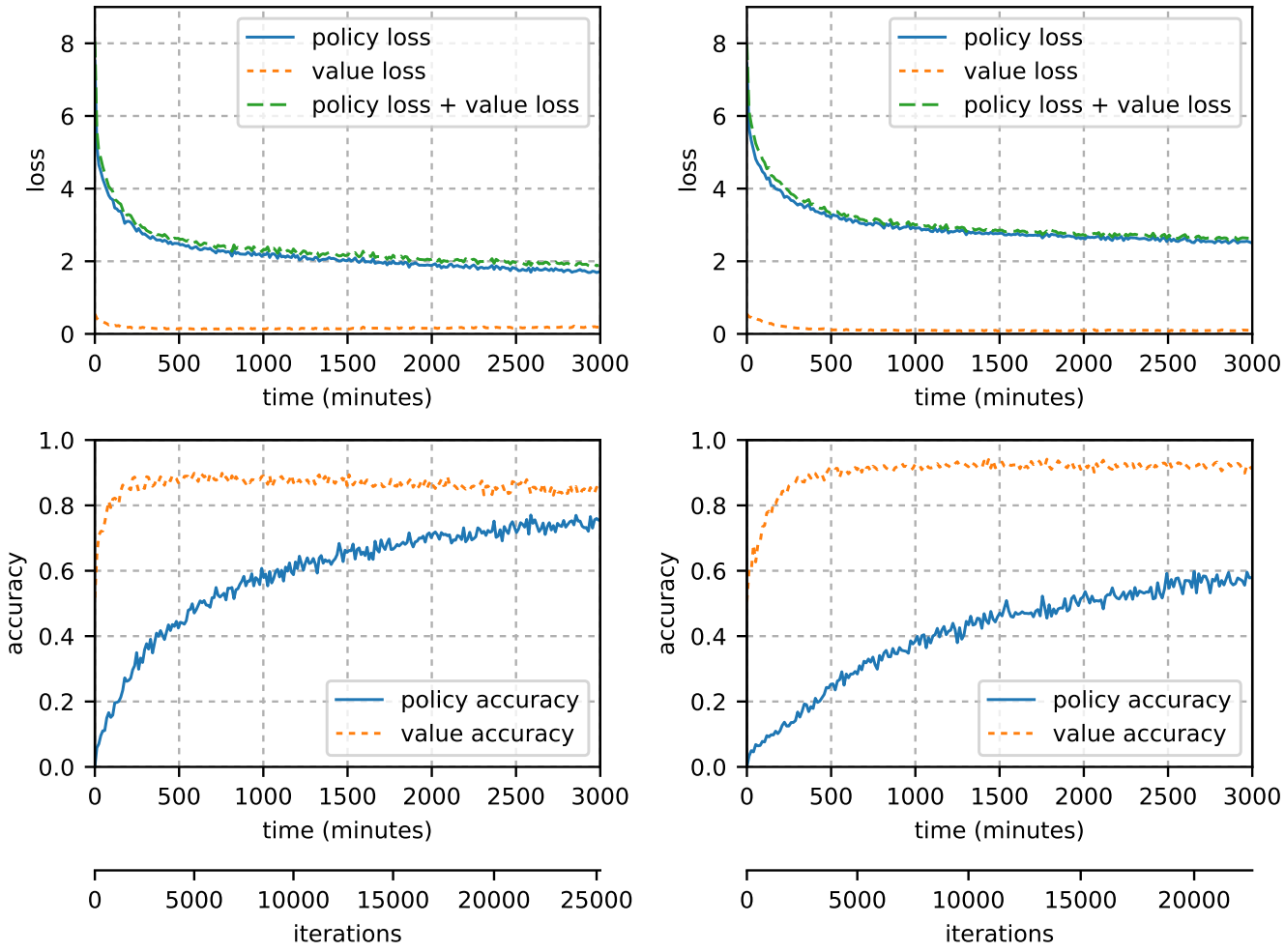

Fig. 1. The left two figures are for agent A (with checkmate solvers), and the right two are for agent B. The top two figures show the policy loss and value loss, and the bottom two figures show the accuracy in training data generated by self-play. The horizontal axes are time elapsed (minutes) and the number of mini-batches processed (iterations).

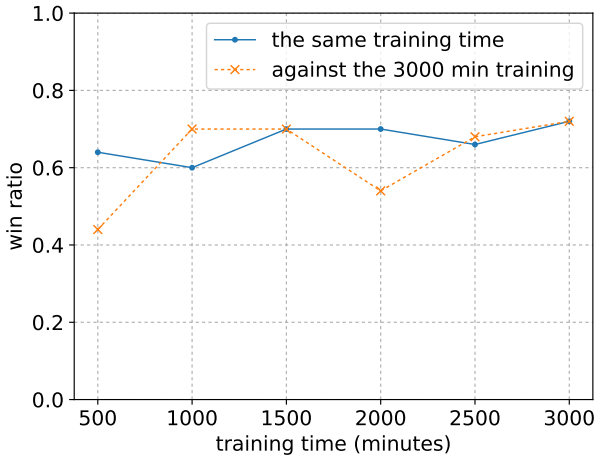

Fig. 2. Win ratio of agent A against agent B.

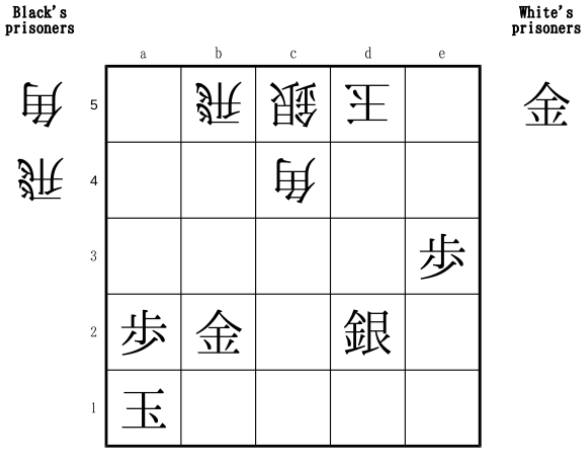

Fig. 3. An example position: White to move and White can immediately win by checkmate, dropping a gold piece to e4.

REFERENCES

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. [Online]. Available: http://arxiv.org/abs/1712.01815

[4] H. Iida, M. Sakuta, and J. Rollason, "Computer shogi," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 121–144, Jan. 2002.

[5] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, March 2012.

[6] G. M. Chaslot, M. H. Winands, and H. J. Herik, "Parallel monte-carlo tree search," in *CG '08: Proceedings of the 6th international conference on Computers and Games*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 60–71.

[7] K. Wu, J. Chen, S. Yen, and S. Hsu, "Design of knowledge-based opening database for minishogi," in *Conference on Technologies and Applications of Artificial Intelligence, TAAI 2012, Tainan, Taiwan, November 16-18, 2012*. IEEE, 2012, pp. 290–293. [Online]. Available: https://doi.org/10.1109/TAAI.2012.22

[8] A. Kishimoto, M. H. Winands, M. Müller, and J.-T. Saito, "Game-tree search using proof numbers: The first twenty years," *ICGA Journal*, vol. 35, no. 3, pp. 131–156, 2012.