Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

Developing Value Networks for Game 2048 with Reinforcement Learning

Kiminori Matsuzaki[1,a)]

Received: July 12, 2020, Accepted: January 12, 2021

Abstract: The game 2048 is a stochastic single-player game and several computer players have been developed in not only research work but also student projects. Among them, the most successful approach is based on N-tuple networks trained by reinforcement learning methods. Though there have been several works on computer players with deep neural networks, their performance were not as good in most cases. In our previous work, we designed policy networks and applied supervised learning, which resulted in an average score of 215,802. In this study, we tackle the problem with value networks and reinforcement learning methods, since value networks are important to combine with game-tree search methods. We investigate the training methods in several aspects, including batches of training, use of symmetry, network structures, and use of game-specific tricks. We then conduct a training for 240 hours with the best configuration. With the best value network obtained, we achieved an average score of 228,100 with the greedy (1-ply search) play, and furthermore an average score of 406,927 by combining it with the 3-ply expectimax search.

Keywords: game 2048, neural network, reinforcement learning, stochastic game, single-player game

1. Introduction

Deep neural networks (DNNs) now play an important role in the development of computer game players. Several master-level computer players have been developed with DNNs trained by reinforcement learning methods for several games: not only two-player perfect-information games like Go (AlphaGo Zero [23]), Chess (Giraffe [13] and DeepChess [5]) and Shogi (AlphaZero [22]) but also other type of games such as Poker (Poker-CNN [29] and DeepStack [19]), Atari games [18], and Mahjong (Suphx [14]).

The target of this study is the game "2048" [4], a stochastic single-player game. 2048 is a slide-and-merge game and its "easy to learn but hard to master" characteristics have attracted quite a few people. According to its author, during the first three weeks after its release, people spent a total time of over 3,000 years playing the game.

Several computer players have been developed for 2048. Among them, the most successful approach is to design N-tuple networks (NTNs) as evaluation functions and apply reinforcement learning methods. This approach was first introduced to 2048 by Szubert and Jaśkowski [24]. The state-of-the-art player developed by Jaśkowski [10] combined several techniques to improve NTN-based players, and achieved an average score of 609,104 within a time limit of 1 second per move. DNN-based computer players, however, have not achieved a big success yet.

There are two approaches to design deep neural networks for computer players: *policy networks* and *value networks*. A policy network takes a game position and returns probabilities of possible moves. A value network takes a game position (and a move) and return the evaluation value. A dual-head network used in AlphaGo Zero [23] is a network that shares some layers of a policy network and a value network. Though we can develop computer players with either network, a value network is more useful when we want to combine it with game-tree search techniques.

Most of the existing DNN-based players for 2048 have been developed taking the value-network approach. We should, however, say that the performance of these players with value networks (especially trained with reinforcement learning) was not good. As far as the author knows, the first study in this direction was by Guei et al. [8]: they developed convolutional neural networks (CNNs) and trained them with Q-learning and TD-learning methods, but the average score achieved was just 11,400 even in the best case. The most successful value network for 2048 was by tjwei [25]: the value network included two convolution layers with 1×2 filters, and the average score was 85,351 with a supervised learning method and about 33,000 with the reinforcement learning method. There are other implementations of computer players with value networks [1], [7], [11], [20], [21], [26], [27] or dual-head networks [2] but the scores of these players were not as good or were not reported.

Not many studies took the policy-network approach [6], [12], [16]. The author focused on policy networks trained with supervised learning in the previous studies. We [12] first designed CNNs with 2×2 filters by changing the numbers of convolution layers and applied supervised learning using play logs of existing NTN-based players [15]. The best network with five convolution layers achieved an average score of 93,830. The author [16] then developed a policy network with 1×2 convolution filters and

[1] School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] matsuzaki.kiminori@kochi-tech.ac.jp

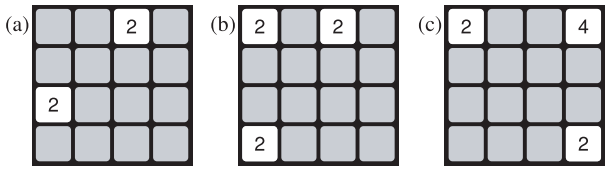
extended it so that it also took the states after the moves as the input. The best computer player with this extended policy network achieved an average score of 215,802. In fact, we also tried to apply these ideas to value networks and dual-head networks, but they performed as poorly as random players.

In this study, we focus on reinforcement learning for value networks for 2048. Starting with the TD-learning that worked well for NTNs [24], we investigate several options of the training details. We also perform a longer training up to 240 hours with the best configuration, and evaluate the developed players in combination with expectimax tree search [3]. Programs and trained parameters are available at http://ipl.info.kochi-tech.ac.jp/matsuzaki-lab/repos/JIP2020Supplements/index.html.

Here are the experiments and results in this paper.

- We first change the batches of training (Section 3.1). Due to the randomness in 2048, the training works well with a batch large enough (N = 1,024), and shuffling of training data is not necessary.
- Symmetry often plays an important role in game playing and training. We adopt the board symmetry in game-plays and in data augmentation (Section 3.2). The experiment results suggest that the advantage of utilizing symmetry does not outweigh the disadvantage.
- We test three more network architectures developed in the previous study [16] (Section 3.3). Though 2048 has a very small board, we confirm that CNNs performs better than MLP and a CNN with 2×2 filters seems to be a good choice in terms of the computational cost and the performance. We also test CNNs with different size (Section 3.4).
- We apply two game-specific tricks in the generation of training data (Section 3.5). The restart strategy [15] works well for DNNs too and it improves the players' performance when combined with expectimax search.
- We perform the training for 240 hours using the best configuration (Section 4). The best computer player obtained in this study performs much better than any other existing DNN-based player.
  - With the greedy (1-ply search) play, the average score is 228,100, 2,048 achievement ratio 98.1%, 4,096 achievement ratio 94.1%, and 8,192 achievement ratio 82.9%.
  - With the expectimax 2-ply search, the average score is 367,024, 2,048 achievement ratio 99.8%, and 4,096 achievement ratio 99.3%, and 8,192 achievement ratio 97.2%.
  - With the expectimax 3-ply search, the average score is 406,927, 4,096 achievement ratio 99.6%, 8,192 achievement ratio 98,8%, and 32,768 achievement ratio 26.0%.

The rest of the paper is organized as follows. Section 2 briefly introduces the rules of 2048 (most of this section comes from our previous paper [12]). Section 3 designs the experiments that evaluate several training options and reports the results. Section 4 reports the final experiments for the best training configuration with longer training time. We review existing DNN-based players for 2048 in Section 5. We summarize the findings in this paper and show future directions in Section 6.



(a) An initial state. Two tiles are placed randomly.
(b) After the first move: *up*. A new 2-tile appears at the lower-left corner.
(c) After the second move: *right*. Two 2-tiles are merged to a 4-tile, and score 4 is given. A new tile appears at the upper-left corner.

Fig. 1 Process of game 2048 [12].

Table 1 Score and number of moves when a tile is first created

tile	score	moves
1,024	9,000	480
2,048	20,000	950
4,096	44,000	1,900
8,192	97,000	3,800
16,384	210,000	7,500
32,768	450,000	15,000

2. Game 2048

2048 is played on a 4×4 grid. The objective of the original 2048 game is to reach a 2,048-tile by moving and merging the tiles on the board according to the rules below. In an initial state (**Fig. 1**), two tiles are placed randomly with numbers 2 ($p_2 = 0.9$) or 4 ($p_4 = 0.1$). The player selects a direction (either up, right, down, or left), and then all the tiles will move in the selected direction. When two tiles of the same number collide, they create a tile with the sum value and the player gets the sum as the score. Here, the merges occur from the far side and newly created tiles do not merge again on the same move: move to the right from `222␣`, `␣422` and `2222` results in `␣␣24`, `␣␣44`, and `␣␣44`, respectively. Note that the player cannot select a direction in which no tiles move nor merge. After each move, a new tile appears randomly at an empty cell with number 2 ($p_2 = 0.9$) or 4 ($p_4 = 0.1$). If the player cannot move the tiles in any direction, the game ends.

In this study, we evaluate the computer players in terms of their average score as well as the achievement ratio of 2,048-, 4,096-, 8,192-, 16,384- and 32,768-tiles (the ratio of games in which such a tile is successfully created). **Table 1** shows the required score and the number of moves when such a tile is first created. For example, a player should perform about 950 moves and obtain a score of about 20,000 or larger before creating the first 2,048-tile.

3. Investigating Training Options

Szubert and Jaśkowski [24] examined three reinforcement learning methods for 2048 using N-tuple networks (NTNs). They reported that TD learning applied to *afterstate* (**Fig. 2**; the state after the tiles slide&merge and before a new tile appears), called *TD-afterstate*, outperformed the other two methods introduced in the paper (called *Q-learning* and *TD-state*). Guei et al. [8] evaluated common Q-learning method and TD-afterstate for deep neural networks (DNNs), and again TD-afterstate achieved the better performance. Following these results, we design our reinforcement learning methods based on the TD-afterstate method.

First we review the idea of TD-afterstate for 2048 using Fig. 2. Given a game position s_t at time t, let the player select a move

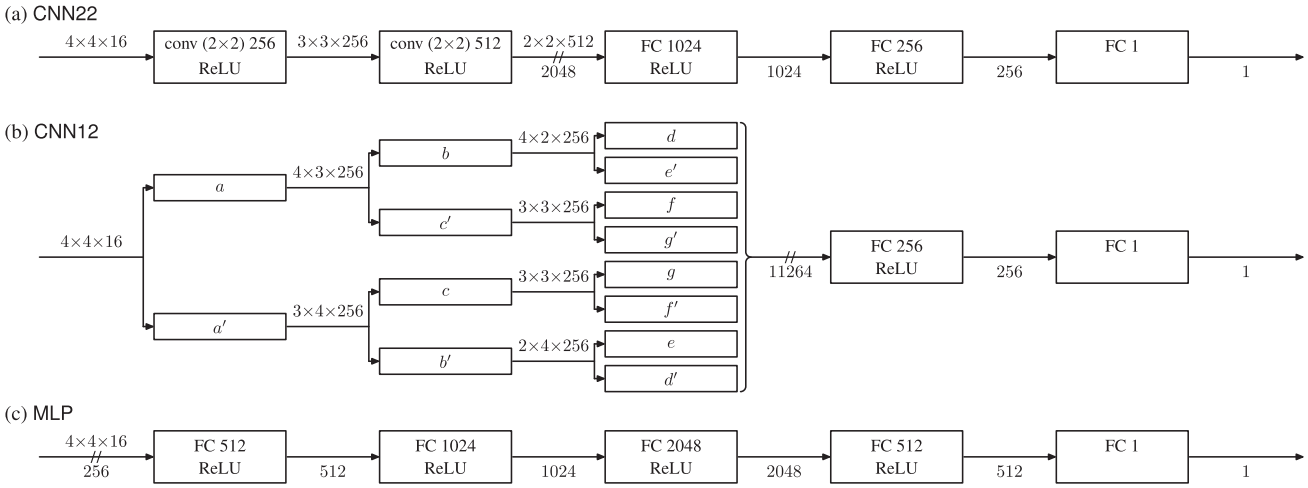


Fig. 3 Structures of (a) CNN22, (b) CNN12, and (c) MLP. The arrows with // denote reshaping of tensors. Each block in the first three layers in CNN12 is either "conv (1×2) 256" or "conv (2×1) 256" where labels x and x' ($x = a, b, \ldots, g$) show that these filters share their weights (transposed).

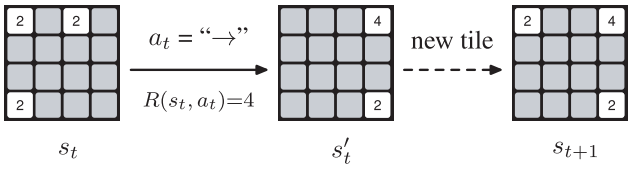


Fig. 2 Definition of afterstate

Table 2 Neural networks used in the paper.

Network	description	number of weights
CNN22	2 × conv (2 × 2) / ReLU [256, 512], 2 × FC / ReLU [1024, 256], FC [1]	2,902,273
CNN12	3 × conv (dual 1 × 2) / ReLU [256, 256×2, 256×4], FC / ReLU [256], FC [1]	3,680,513
CNN22ACG	2 × conv (2 × 2) / ReLU [160, 320], 2 × FC / ReLU [2048, 256], FC [1]	3,363,809
MLP	4 × FC / ReLU [512, 1024, 2048, 512], FC [1]	3,280,897
CNN22S	2 × conv (2 × 2) / ReLU [192, 384], 2 × FC / ReLU [768, 192], FC [1]	1,636,033
CNN22SS	2 × conv (2 × 2) / ReLU [128, 256], 2 × FC / ReLU [512, 128], FC [1]	730,241
CNN22SSS	2 × conv (2 × 2) / ReLU [64, 128], 2 × FC / ReLU [256, 64], FC [1]	184,897

Abbreviations: conv = convolution layer(s), FC = full-connect layer(s)
The numbers in square brackets show the number of filters in each layer.

a_t (either up, right, down, or left). Then, all the tiles slide (and merge) resulting afterstate s'_t with score $R(s_t, a_t)$. The next position s_{t+1} is given by adding a random tile to the afterstate s'_t. In the TD-afterstate method, we apply a TD learning algorithm to the consecutive afterstates using $R(s_t, a_t)$ as the reward. With the basic TD(0) algorithm, we update the evaluation value of an afterstate to approach $V(s'_t)$:

$$V'(s'_t) = R(s_{t+1}, a_{t+1}) + V(s'_{t+1}). \quad (1)$$

Note that we cannot define the afterstate s'_e of the end game s_e, but let the reward and the evaluation value be $R(s_e, \cdot) = 0$ and $V(s'_e) = 0$ in that case. After the training steps, the evaluation value $V(s)$ will approach the expected score obtained from the hypothetical afterstate s to the end of the game.

Table 2 summarizes the neural networks used in this study and **Fig. 3** illustrates the network structures of CNN22, CNN12, and MLP. Following our previous work [12], [16], we used the same input encoding method as Guei et al. [8]. As shown in **Fig. 4**, the input is a $4 \times 4 \times 16$ binary array: the first 4×4 shows the places of the empty cells, the second does those of 2-tiles, and so on, up to 32,768-tiles. Except for Sections 3.3 and 3.4, we use CNN22 for our experiments. It consists of five layers (first two layers are convolution layers with 2×2 filters and the last three layers are full-connect layers) and the first four layers are followed by ReLU activation function. The numbers of filters are 256, 512, 1,024, 256, and 1, in order. The initial values of the weights are given randomly from the normal distribution (mean $\mu = 0$, variance $\sigma^2 = (0.1)^2$) truncated between $\pm 2\sigma$. The loss function we used was mean squared error between the evaluation values $V(s'_t)$ and the desired value $V'(s'_t)$ in Eq. (1). We used those values directly without any scaling. We use the optimization algorithm *Adam* with the default hyper parameters in TensorFlow (learning rate $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$) [*1].

The computers we used equipped an Intel Core i7-9800X CPU (3.8 GHz, 8 cores/16 threads), 128 GB Memory, and two NVIDIA GeForce RTX 2080Ti GPUs (11 GB GPU Memory). The OS and software were Ubuntu 18.04.3 LTS, Python 3.7.3, and TensorFlow 1.13.1. Though the computers were equipped two GPUs, all the programs in this study used a single GPU. We executed two training programs on a computer in such a way that each program used a different GPU, except for the last training phase in Section 4. Since there was only one instance of the weights of a network, the batches were carried out one by one on a GPU. We used a GPU to speed up the computation (much faster than using a CPU) but still the dominant part in the computation was for computing evaluation values and updating the weights.

For statistical stability, we executed each training and evaluation for five times and we show their mean in the graphs and tables unless otherwise stated.

[*1] We also tested the stochastic gradient descent (SGD) algorithm but it was too sensitive to the learning rate and the results were worse than those with Adam).

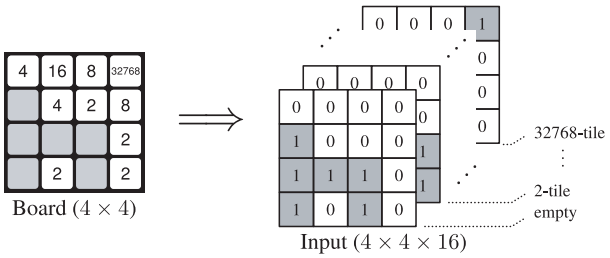


Fig. 4 Input encoding.

3.1 Batch of Training

TD learning is often implemented in an online manner, i.e., we perform a training process concurrently with game-plays. For example, in the implementation by Szubert and Jaśkowski [24], the training function was called every time a new move was made [*2]. Such a straightforward implementation, however, is not good in the case of DNNs. The first reason is that calling the training function on a GPU has a certain overhead. The second reason is that the correlation between consecutive positions would have bad effects.

In our implementation, we developed two functions, *generator* and *trainer*, each executed on different threads that shared the weights of the DNN on a single GPU. A generator performed game-plays and put the positions s'_t and the target evaluation values $V'(s'_t)$ to the queue. At each step, a generator ran a batch of feed-forward computation to obtain the evaluation values of four possible afterstates and selected a move corresponding to the maximum value. *trainer* took positions from the queue and called the training function. As mentioned earlier, all the programs used a single GPU that executed batches one by one. Therefore, these generator and trainer threads ran asynchronously on a CPU (for preparing batches), but they were blocked at the call of a batch on a GPU.

We conducted an experiment with different implementations of the generator and trainer functions (Experiment 1).

game The generator put a list of positions after each game. The trainer took a list of positions from the queue and called the training function once for the whole positions. (variable-size batch training)

batch1K, batch64 The generator put a position after each move. The trainer took N = 1,024 or 64 positions from the queue and called the training function. (fixed-size batch training)

shuffle1K, shuffle64 The generator put a position after each move. The trainer took $5N$ = 5,120 or 320 positions from the queue, shuffled them, and called the training function five times with batch size N = 1,024 or 64. (training with replay buffer)

position The generator put a position after each move. The trainer called the training function for each position in a queue. (online TD learning)

In each implementation except for position, we used five generator threads and one trainer thread. In the implementation of position, the generator and trainer functions were called alternately.

[*2] The program by tjwei [25] also called the training function for each new move. It was not described how Guei et al. [8] performed the training of TD learning.

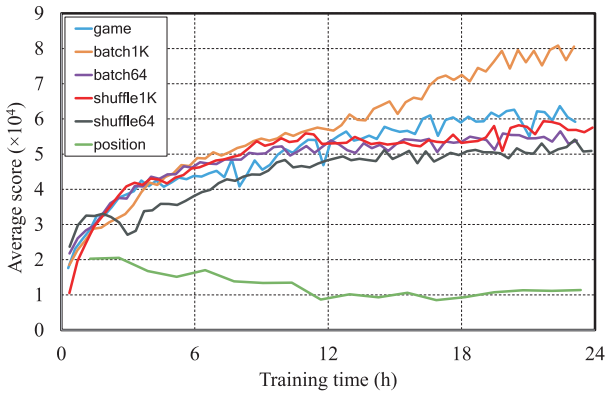


Fig. 5 Experiment 1. Average scores.

Table 3 Results of Experiment 1. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	speed (pos./h)
game	61,941 ± 3,128	96.8%	79.2%	3.4%	2.94×10^6
batch1K	80,851 ± 2,023	92.6%	77.4%	32.4%	2.78×10^6
batch64	56,486 ± 5,534	90.4%	66.8%	4.4%	2.72×10^6
shuffle1K	58,536 ± 3,149	94.6%	73.6%	1.4%	2.72×10^6
shuffle64	54,120 ± 1,585	93.4%	64.2%	1.0%	2.68×10^6
position	11,373 ± 7,310	20.8%	0.0%	0.0%	0.77×10^6

For each setting, we executed the training program for 24 hours. To monitor the progress of the training, we took a snapshot after each increment of 10^6 positions, and calculated the average score of the latest 100 games that the generators played [*3]. **Figure 5** shows the progress of the training. **Table 3** summarizes the best results from the last five snapshots for each setting, in terms of mean and standard deviation of the average scores, the achievement ratios of 2,048-, 4,096-, and 8,192-tiles, and the training speed.

Results and Discussion

The best average score was achieved by batch1K under the same training time.

Small batch sizes are not good. In the extreme cases, position started with an average score of around 20,000 but decreased it down to 11,000. With batch64 the training was fast in a very early stage (before 6 hours), but the improvement slowed down. The result of the game was better than batch64 but worse than batch1K. Note that the batch size changed over the training with game, and they were about 2,800 in average.

Additional experiment was performed to investigate the effects of batch size N for N = 32 to 4,096. **Table 4** shows the average score and achievement ratios after 24-hour training. The results suggest that too small or too large batch size is not good. The author considers the reason of poor performance for the latter is that large batches make the training slow.

The results of shuffle1K and shuffle64 were poorer than those of batch1K and batch64. With shuffling, we have the advantage of less correlation among the training data but also a disadvantage with delays of updating networks. This advantage seems to be small for 2048. I consider the reason is that we already have wide variety of positions without shuffling since new tiles appear

[*3] The average score was of greedy (1-ply search) play.

Table 4 Additional experiment by changing the batch size. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

batch size	average score	achievement ratio 2,048	4,096	8,192
32	44,993 ± 3,163	86.2	51.8	0.6
64	56,486 ± 5,534	90.4	66.8	4.4
128	73,577 ± 6,678	93.4	74.0	24.0
256	83,884 ± 3,592	92.8	79.0	35.6
512	85,943 ± 5,194	92.4	77.4	39.0
1,024	80,851 ± 2,023	92.6	77.4	32.4
2,048	75,319 ± 4,306	92.8	74.2	28.0
4,096	63,442 ± 4,886	91.8	69.4	12.2

in a random place.

For the rest of this study, we use batch1K.

3.2 Exploiting Symmetry of Boards

Symmetry often plays an important role in game playing and in machine learning. In the game 2048, we have eight symmetric boards for each position by rotation and/or reflection.

In this study, we considered utilizing the symmetry of boards in two ways. First, we utilized symmetry in the selection of moves in generators. In addition to simple, which directly selected a hand without looking at symmetric boards, we implemented the following three methods of selecting moves.

- ave calculated all the evaluation values of the symmetric boards and selected a move with the maximum *average* value.
- max calculated all the evaluation values of the symmetric boards and selected a move with the maximum *maximum* value. (an optimistic move selection)
- min calculated all the evaluation values of the symmetric boards and selected a move with the maximum *minimum* value. (a pessimistic move selection)

Secondly, we utilized symmetry in the training by data augmentation. By data augmentation, we fed all the symmetric boards to the training function, and therefore the number of training data increased by a factor of eight. We conducted an experiment to evaluate how symmetry helped the training for all the combination of move-selection algorithms and with/without data augmentation (Experiment 2).

For each setting, we executed the training program for 24 hours. We took a snapshot after each increment of 10^6 positions, and monitored the progress of training with the latest 100 games. **Figure 6** shows the progress of training without data augmentation, and **Fig. 7** with data augmentation. **Table 5** summarizes the best results from the last five snapshots for each setting, in terms of mean and standard deviation of the average score, the achievement ratios of 2,048-, 4,096-, and 8,192-tiles, and the training speed.

Results and Discussion

First, it is worth noting that the number of trained positions almost halved with the generator's algorithms with symmetric boards. Even though we called the DNN evaluation function once per batch of 8 symmetric boards, the overhead of generating and evaluating eight symmetric boards was not negligible. Compared with this, the data augmentation with symmetric boards did not have overhead (in fact, the training with data augmentation was faster for some undetermined reason).

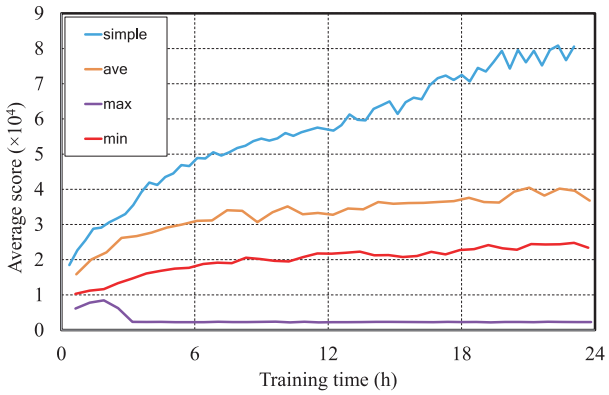


Fig. 6 Experiment 2. Average scores without data augmentation.

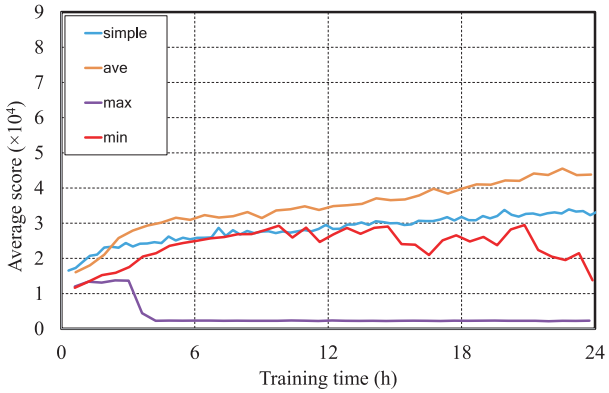


Fig. 7 Experiment 2. Average scores with data augmentation.

Table 5 Results of Experiment 2. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	speed (pos./h)
Without data augmentation					
simple	80,851 ± 2,023	92.6%	77.4%	32.4%	2.78×10^6
ave	40,460 ± 601	87.8%	29.0%	0.6%	1.47×10^6
max	2,356 ± 152	0.0%	0.0%	0.0%	1.55×10^6
min	24,785 ± 2,631	58.2%	3.8%	0.0%	1.56×10^6
With data augmentation					
simple	33,928 ± 1,276	80.5%	18.0%	0.0%	3.11×10^6
ave	45,523 ± 2,715	91.3%	42.8%	0.8%	1.55×10^6
max	2,305 ± 65	0.0%	0.0%	0.0%	1.64×10^6
min	22,400 ± 5,574	47.8%	0.8%	0.0%	1.63×10^6

Secondly, the optimistic algorithm max definitely failed the training. It quickly dropped down to a level similar to random-players. The results of the pessimistic algorithm min were also poor, and the average score gradually decreased with data augmentation.

As the author expected, the algorithm ave worked better than simple with data augmentation. However, without data augmentation, the algorithm simple significantly outperformed ave. I suggest the following reason. With the algorithm ave, the selected move should have large evaluation values for many (or all) symmetric boards. This means that the training of the network proceeded in a symmetric way in some sense even without data augmentation. The network trained with ave captured symmetric features and thus it captured less essential features. The network trained with simple captured more features but the

Fig. 8 Experiment 3. Average scores for different networks.

Table 6 Result of Experiment 3. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	speed (pos./h)
CNN22	80,851 ± 2,023	92.6%	77.4%	32.4%	2.78×10^6
CNN12	59,083 ± 3,998	93.4%	69.4%	3.4%	1.51×10^6
CNN22ACG	76,466 ± 5,375	93.2%	75.0%	28.6%	2.67×10^6
MLP	59,349 ± 3,388	90.6%	68.2%	10.2%	2.80×10^6

trained network became asymmetric. In fact, the author confirmed the network trained by simple returned completely different values for symmetric boards. This fact explains ineffective data augmentation for simple: the data augmentation fed symmetric inputs to the training, which resulted in spoiling simple's advantage of asymmetry.

From here on out in this study, we use simple for the generator and do not use data augmentation by symmetric boards.

3.3 Comparison with Other Networks

In previous work [16], the author designed three policy networks and applied supervised learning. We conducted an experiment to compare those three networks in the setting of value networks trained by reinforcement learning (Experiment 3). These networks used were the same as those in our previous work, except for the last layer outputting only one value (Table 2).

CNN12 This network consisted of three convolution layers with 1×2 filters and two full-connect layers. For each input of convolution layers, those filters were applied in the horizontal or vertical directions, yielding two outputs. Note that the weights in the filters were shared according to the symmetry to halve the number of parameters.

CNN22ACG This network consisted of two convolution layers with 2×2 filters and three full-connect layers. Note that the number of filters were different from CNN22.

MLP (multi-layer perceptrons) This network consisted of five full-connect layers.

For each network, we executed the training program for 24 hours. We took a snapshot after each increment of 10^6 positions, and monitored the progress of training with the latest 100 games. **Figure 8** shows the progress of training. **Table 6** summarizes the best results from the last five snapshots for each setting, in terms of mean and standard deviation of the average score, the achievement ratios of 2,048-, 4,096-, and 8,192-tiles, and the training speed.

Table 7 Result of Experiment 4. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	speed (pos./h)
CNN22	80,851 ± 2,023	92.6%	77.4%	32.4%	2.78×10^6
CNN22S	78,252 ± 3,227	94.0%	77.2%	29.8%	2.82×10^6
CNN22SS	59,412 ± 3,167	92.4%	67.0%	7.4%	2.71×10^6
CNN22SSS	43,263 ± 1,626	84.8%	43.0%	0.4%	2.79×10^6

Results and Discussion

Under the same training time, the CNNs with 2×2 filters outperformed the CNN12 and MLP networks. It is worth noting that the computation of the CNN12 network was more expensive than the others and under the same number of trained positions the results of CNN22 and CNN12 were almost the same.

The result of CNN22 was a bit better than CNN22ACG. The difference between them was the number of filters in layers. CNN22 had more filters in the first and the second convolution layers (but less in the third full-connect layer). We will discuss the relationship between the number of filters and the average score after the next experiment.

It is worth noting that CNNs worked better than MLP even though the board size is quite small. The author considers that this is because CNNs can capture local features well and it matches with the game 2048 for the merges of adjacent tiles. The CNN22 network worked better than CNN12 (or comparable under the same number of trained positions), which was different from our previous study with policy networks trained by supervised learning [16]. A further investigation will be needed to clarify the reason of this, but a possible reason is that we applied a set of filters both horizontally and vertically in CNN12 (see (b) in Fig. 3) and it reduced the advantage of asymmetry.

3.4 Changing Network Size

The training of the CNN22 network required 8,445 MB of GPU memory, and therefore it would be almost the largest network executable on commodity GPUs. We conducted an experiment to evaluate the relation between the number of parameters and the players' performance (Experiment 4). Based on the CNN22 network, we designed three smaller networks, CNN22S, CNN22SS and CNN22SSS, by reducing the number of filters in each layer by a factor of 0.75, 0.5, and 0.25, respectively. The number of parameters decreased by the factor squared as shown in Table 2.

For each network, we executed the training program for 24 hours. We took a snapshot after each increment of 10^6 positions, and monitored the progress of training with the latest 100 games. **Figure 9** shows the progress of training. **Table 7** summarizes the best results in the last five snapshots for each setting, in terms of mean and standard deviation of the average score, the achievement ratios of 2,048-, 4,096-, and 8,192-tiles, and the training speed.

Results and Discussion

From the results, we can roughly see that if we double the number of filters in each layer, the average score increases by a factor of $\sqrt{2}$. This means that if we increase the number of parameters by a factor of a, the average score increases by a factor

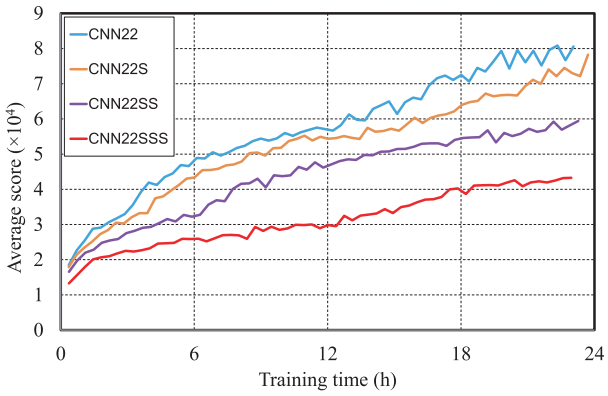


Fig. 9 Experiment 4. Average scores for networks of different sizes.

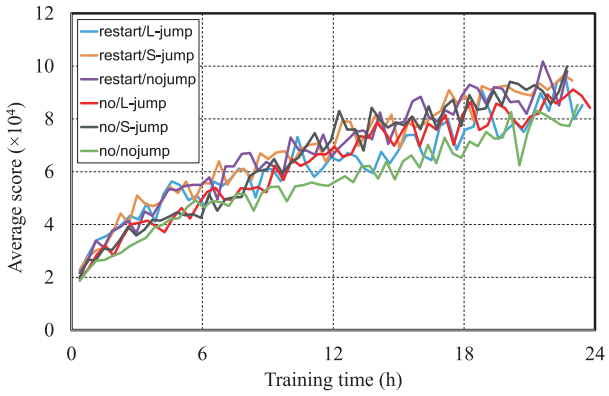


Fig. 10 Experiment 5. Average scores of greedy (1-ply search) play.

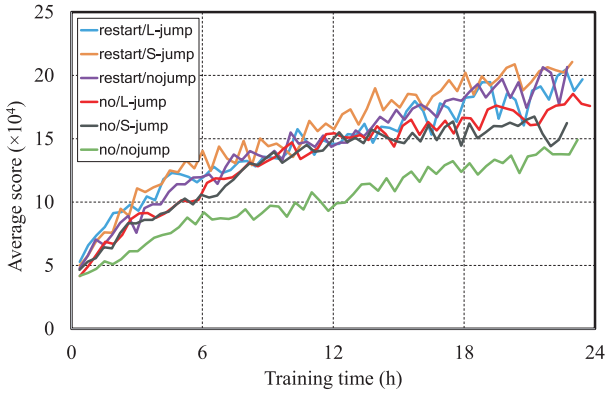


Fig. 11 Experiment 5. Average scores of expectimax (2-ply search) play.

of $\sqrt[4]{a}$.

3.5 Game-specific Tricks in Training Data Generation

In the experiments so far, we did not utilize game-specific features in the training, except for the idea of the TD-afterstate algorithm. Now we introduce some game-specific tricks in training-data generation to enhance the performance of the trained networks.

In this study, we took the following two characteristics of the game into account.

- When we play a game longer, the game generally becomes harder. Very large tiles like 8,192 or 16,384 become obstacles when we deal with the smaller tiles.
- In particular, the game is very hard just before achieving a very large tile, because we need to have a sequence of large tiles on the board (e.g., before achieving a 16,384-tile, we should have 8,192-, 4,096-, 2,048-, 1,024-, ... tiles on the board).

The author proposed the *restart strategy* [15] to adapt to the first characteristic. With this idea we restart a game-play from the middle of the play record: more precisely, let a game-play start at turn s_k and end at turn e_k, then the next play starts at the middle $s_{k+1} = (s_k + e_k)/2$ (If the play is too short, $(e_k - s_k) \leq 10$, we start the next play from an initial state). With this restart strategy, networks can learn more from the later stages of the game.

For the second characteristic, we propose the *jump-start strategy*. Instead of starting from an initial state, we started a game-play from a state that already included large tile(s). Since we executed five generator threads in our training, we allocated a different starting point for each generator thread as follows. In the following definition, an initial tile is either 2-tile $p_2 = 0.9$ or 4-tile $p_4 = 0.1$.

L-jump (1) two initial tiles, (2) a 4,096-tile and an initial tile, (3) a 8,192-tile and an initial tile, (4) a 8,192-tile and a 4,096-tile, (5) a 16,394-tile and an initial tile [*4].

S-jump (1) two initial tiles, (2) a 2,048-tile and an initial tile, (3) a 4,096-tile and an initial tile, (4) a 4,096-tile and a 2,048-tile, (5) a 8,192-tile and an initial tile.

nojump All the threads started with two initial tiles.

Note that this idea of the jump-start strategy was similar to Wu et al.'s multi-staged training [28] and Jaśkowski's carousel shaping [10]. These techniques stored specific positions (when we achieved a large tile or simply after specific number of turns) and started a game-play from one of those positions. In our jump-start strategy, we simply started with positions with large tile(s) and do not need to collect those positions.

[*4] These positions are unreachable in a real play since a new tile appears after each move.

Table 8 Result of Experiment 5 for greedy (1-ply search) play. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	16,384
With Restarting					
L-jump	95,216 ± 10,845	90.0%	75.3%	41.4%	5.8%
S-jump	96,595 ± 12,188	91.6%	78.3%	43.1%	5.1%
nojump	101,782 ± 7,811	94.1%	78.7%	46.3%	4.8%
Without Restarting					
L-jump	91,149 ± 14,921	93.0%	76.2%	40.7%	1.6%
S-jump	99,835 ± 8,281	95.2%	82.6%	50.1%	0.5%
nojump	85,344 ± 8,003	95.4%	81.3%	35.6%	0.0%

We conducted an experiment for each combination of with/without the restart strategy and the jump-start strategies (Experiment 5). We took a snapshot for each after 10^6 positions. In this experiment, we evaluate the snapshot with the greedy (1-ply search) and the expectimax (2-ply search) plays. The numbers of test games were 300 for the greedy play and 100 for the expectimax 2-ply play. **Figures 10** and **11** show the average scores of greedy play and expectimax 2-ply play, respectively. **Tables 8** and **9** summarize the best results from the last five snapshots for the greedy play and the expectimax 2-ply play, respectively, in terms of mean and standard deviation of the average score, the achievement ratios of 2,048-, 4,096-, 8,192-, and 16,384-tiles.

Table 9 Result of Experiment 5 for expectimax (2-ply search) play. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for five runs.

	average score	achievement ratio 2,048	4,096	8,192	16,384
With Restarting					
L-jump	204,584 ± 11,109	99.0%	96.6%	82.6%	40.8%
S-jump	210,642 ± 25,023	100.0%	97.2%	87.0%	43.2%
nojump	206,645 ± 8,864	99.2%	96.2%	85.6%	39.8%
Without Restarting					
L-jump	185,436 ± 6,991	99.4%	96.2%	85.6%	27.8%
S-jump	167,544 ± 5,311	99.2%	98.0%	88.0%	17.2%
nojump	149,193 ± 9,118	100.0%	98.6%	86.0%	3.6%

Since the training speed was almost the same, we omit it from the table.

Results and Discussion

For both the greedy and the expectimax 2-ply plays, the game-specific training methods worked well. In particular, we found bigger improvements with the expectimax 2-ply play. It is worth noting that the average scores were increasing even at 24 hours for all the training methods.

We found an important difference between the restart and the jump-start strategies on the achievement ratio of a 16,384-tile. For the greedy play, the restart strategy increased the ratio from 0.0% to 4.8% while the jump-start strategy from 0.0% to 1.6%. For the expectimax play, the player with the restart strategy achieved a 16,384-tile for 39.8%, which was significantly more often than the player with the jump-start strategy. Note that the improvements by combining these two strategies was rather small, and furthermore the average score was lower than that of player with the restart strategy only in some cases. This suggests that the advantages by the jump-start strategy were almost covered by the restart strategy.

We will utilize the restart strategy and the jump-start strategy S-jump, which achieved a 16,384-tile the most often.

4. Longer Training with Best Configuration

4.1 Training up to 240 Hours

With the experiments in Section 3, we selected the following options for the training of DNNs:

- fixed-size batch training batch1K,
- asymmetric generator algorithm simple without data augmentation,
- network structure CNN22, and
- with the restart strategy and the jump-start strategy S-jump.

Under this configuration, we executed the training program for 240 hours (Experiment 6). In this experiment, we executed *one* program on a computer. Though the program used only one GPU, the single execution was a bit faster and did training of 700×10^6 positions in total.

We took a snapshot after each increment of 5×10^6 positions, and monitored the progress of training with test plays with the greedy (1-ply search) and the expectimax (2-ply and 3-ply search) plays: we played 300 games with the greedy play for each snapshot; 100 games with the expectimax 2-ply play for each snapshot; and 20 games with the expectimax 3-ply play for every other snapshot (for every 10×10^6 positions). **Figures 12**, **14**, and **16** show the average scores over the training for the greedy, the expectimax 2-ply, and the expectimax 3-ply plays, respectively. **Figures 13**, **15**, and **17** show the achievement ratios of 2,048-, 4,096-, 8,192-, 16,384-, and 32,768-tiles for the greedy, the expectimax 2-ply, and the expectimax 3-ply plays, respectively.

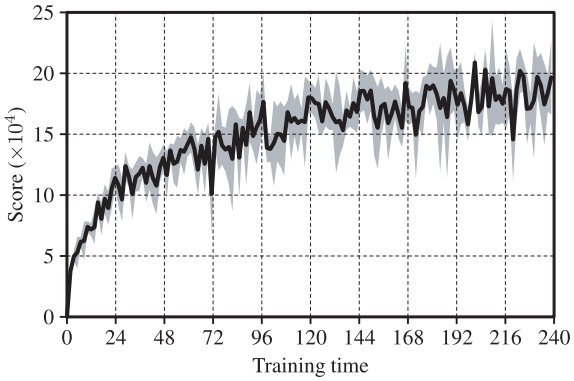


Fig. 12 Experiment 6. Average scores of greedy (1-ply search) play. The dark area shows the size of standard deviation among five runs.

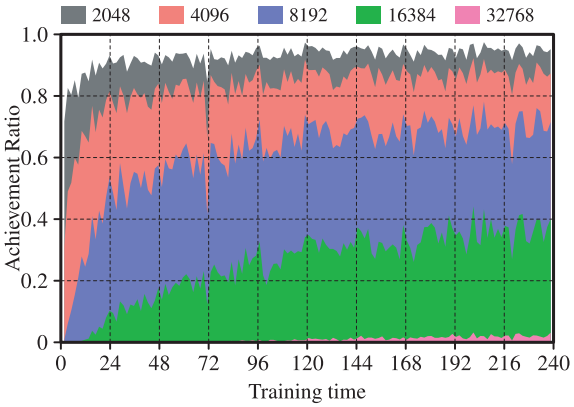


Fig. 13 Experiment 6. Achievement ratios of greedy (1-ply search) play.

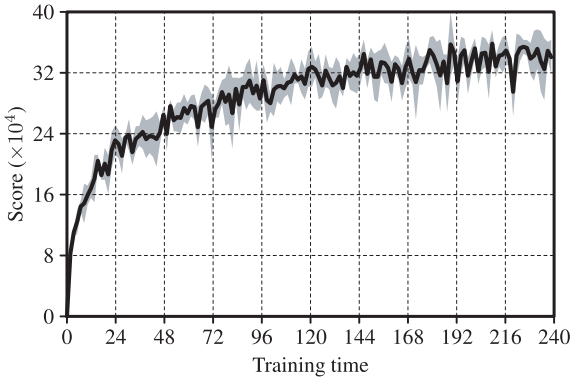


Fig. 14 Experiment 6. Average scores of expectimax (2-ply search) play. The dark area shows the size of standard deviation among five runs.

Results and Discussion

By increasing the training time from 24 hours to 240 hours, we obtained significant improvement for both the average score and the achievement ratios. The average score increased from 114,108 to 196,660 with the greedy play, from 231,325 to 340,514 with the expectimax 2-ply play, and from 252,070 to 381,880 with the expectimax 3-ply play. The achievement ratios of a 16,384-tile increased from 10.3% to 40.1% with the greedy play, from 49.4% to 77.4% with the expectimax 2-ply play, and from 59.0% to 85.0% with the expectimax 3-ply play.

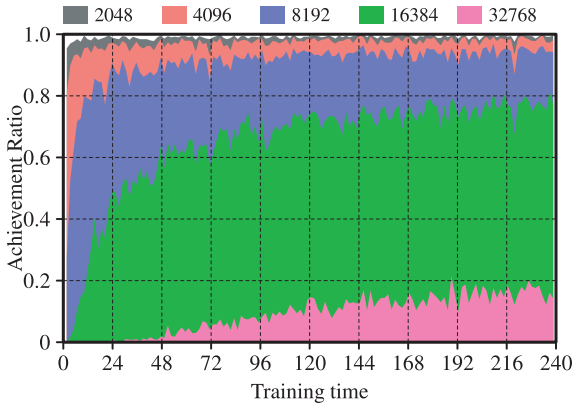


Fig. 15 Experiment 6. Achievement ratios of expectimax (2-ply search) play.

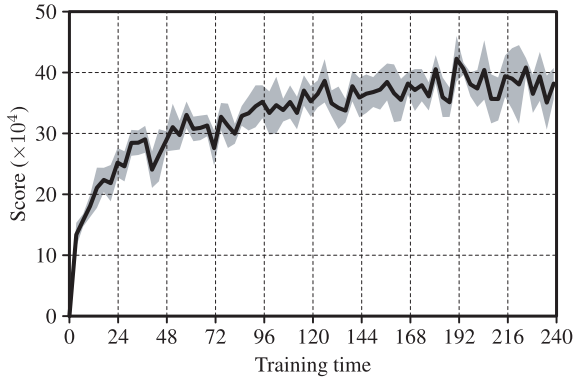


Fig. 16 Experiment 6. Average scores of expectimax (3-ply search) play. The dark area shows the size of standard deviation among five runs.

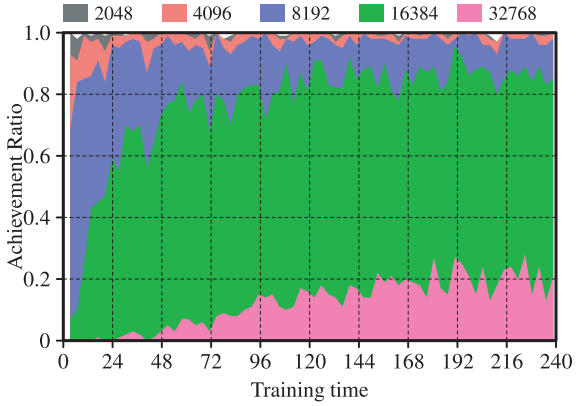


Fig. 17 Experiment 6. Achievement ratios of expectimax (3-ply search) play.

The average score and the achievement ratios of the greedy play and expectimax 2-ply search were still increasing even after the 240-hour training. Those of the expectimax 3-ply search, however, seems to reach the convergence.

4.2 Detailed Evaluation with Best-trained Networks

After Experiment 6, we selected the best network in terms of the average score of the expectimax 2-ply play, for up to each increment of 24 hours up to 240 hours. We executed 500 games of the greedy play, 200 games of the expectimax 2-ply play, and 50 games of the expectimax 3-ply play for ten different random seeds (Experiment 7). **Table 10** summarizes the mean and standard deviation of the average score, the achievement ratios of 2,048-, 4,096-, 8,192-, 16,384-, and 32,768-tiles.

Table 10 Results of Experiment 7. The column of the average score shows the mean (before ±) and the standard deviation (after ±) of average scores for ten runs.

	average score	achievement ratio 2,048	4,096	8,192	16,384	32,768
Greedy play						
24 h	123,160 ± 2,180	95.9%	86.8%	57.1%	10.8%	0.0%
48 h	130,239 ± 4,175	95.6%	85.0%	61.8%	15.9%	0.0%
72 h	165,093 ± 3,785	96.3%	90.3%	71.4%	28.2%	0.2%
96 h	202,146 ± 5,605	97.2%	91.2%	77.1%	42.7%	1.5%
120 h	176,654 ± 3,621	96.1%	89.0%	70.8%	33.3%	1.2%
144 h	210,145 ± 3,095	96.7%	91.8%	78.6%	44.0%	2.8%
168 h	186,425 ± 5,830	95.4%	90.4%	73.7%	38.2%	0.9%
192 h	197,110 ± 6,295	95.6%	89.2%	74.9%	39.5%	2.9%
216 h	228,100 ± 6,511	98.1%	94.1%	82.9%	49.9%	3.3%
240 h	206,802 ± 5,479	96.9%	91.9%	77.9%	43.8%	3.2%
Expectimax 2-ply play						
24 h	239,866 ± 6,414	99.7%	98.5%	91.3%	54.2%	0.1%
48 h	263,276 ± 5,300	99.8%	98.7%	93.7%	62.5%	0.9%
72 h	287,763 ± 8,140	99.4%	98.5%	93.2%	68.3%	4.9%
96 h	335,398 ± 10,653	99.8%	99.2%	96.0%	77.7%	12.8%
120 h	323,766 ± 13,322	99.5%	98.5%	94.3%	73.0%	13.1%
144 h	347,705 ± 10,194	99.8%	99.3%	96.3%	78.7%	15.8%
168 h	347,561 ± 8,626	99.5%	99.0%	96.3%	80.4%	15.1%
192 h	356,190 ± 15,850	99.5%	98.5%	96.2%	78.8%	19.2%
216 h	367,024 ± 11,165	99.8%	99.3%	97.2%	82.8%	18.9%
240 h	355,285 ± 8,636	99.2%	98.7%	95.5%	81.3%	17.7%
Expectimax 3-ply play						
24 h	268,758 ± 11,888	100.0%	99.6%	96.6%	66.6%	0.2%
48 h	305,141 ± 8,454	99.8%	99.4%	97.6%	77.0%	2.2%
72 h	313,906 ± 18,324	99.8%	99.2%	97.6%	79.6%	4.2%
96 h	366,132 ± 23,132	99.8%	99.8%	97.6%	87.6%	14.4%
120 h	353,911 ± 18,916	100.0%	99.0%	97.6%	83.0%	14.6%
144 h	377,691 ± 17,622	100.0%	99.2%	97.8%	87.4%	18.2%
168 h	387,064 ± 17,625	100.0%	99.6%	98.8%	89.0%	20.4%
192 h	406,927 ± 27,410	100.0%	99.6%	98.8%	90.2%	26.0%
216 h	404,717 ± 13,802	100.0%	100.0%	99.4%	93.2%	23.8%
240 h	394,632 ± 18,777	100.0%	100.0%	99.0%	87.6%	24.2%

Results and Discussion

As we can see in Table 10, the best network (after training of 700×10^6 positions) combined with the expectimax search achieved the average score 406,927, achievement ratio 99.6% for a 4,096-tile, 98.8% for an 8,192-tile, 90.2% for a 16,384-tile, and 26.0% for a 32,768-tile. With the combination of value networks and the expectimax search, we obtained significantly better results compared with the state-of-the-art DNN-based player [16]. It is worth noting that the best network after reinforcement learning achieved an average score of 228,100 without tree search, which is comparable to an average score of 215,802 of the policy network after supervised learning [16].

4.3 Comparison with State-of-the-art Player

The state-of-the-art player of the game 2048 was based on N-tuple networks (NTNs) trained by a reinforcement learning technique [10]. This study tried to develop a very strong player with the combination of DNNs and reinforcement learning, but the results were inferior to those: an average score of 324,710 with the greedy play and 511,759 with the expectimax 3-ply play. The big gap was in the achievement ratio of a 32,768-tile: DNN 3.3% vs NTN 19% with the greedy play and DNN 26.0% vs NTN 50% with the expectimax 3-ply search.

There was also a big gap in the computational cost. In this study, we ran the training for 240 hours with 7×10^8 positions. Existing studies on NTNs did the training for 131 hours

Table 11 Summary of existing DNN-based players for 2048.

	authors	Network	Training	number of weights[(a)]	ave. score
Policy	Dedieu and Amar [6]	MLP (3 FC)	reinforcement learning	0.07×10^6	—
	Kondo and Matsuzaki [12]	CNN (5 conv (2×2) + 1 FC)	supervised learning	0.81×10^6	93,830
	Matsuzaki [16]	CNN (3 conv (dual 1×2) + 2 FC), policyAS	supervised learning	3.69×10^6	215,802
Dual-head	Allik et al. [2]	ResNet (1 conv (3×3), 20 res, 2 FC)	supervised learning	2.72×10^6	≈31,000
Value	Guei et al. [8]	CNN (2 conv (2×2) + 2 FC)	TD learning	—	≈11,400
	tjwei [25]	CNN (2 conv (dual 1×2) + 1 FC)	supervised learning	16.94×10^6	85,351
	tjwei [25]	CNN (2 conv (dual 1×2) + 1 FC)	TD learning	16.94×10^6	≈33,000
	Virdee [26]	CNN (2 conv (dual 1×2) + 2 FC)	reinforcement learning	1.97×10^6	≈16,000
	Adamkiewicz [1]	—	Q-learning	—	< 10,000
	Qiu et al. [20]	CNN (2 conv (dual $1\times2 + 2\times2$) + 1 FC)	TD learning	25.40×10^6	≈39,000[(b)]
	Ji et al. [11]	CNN (2 conv (dual 1×2) + 1 FC)	Q-learning	4.49×10^6	< 20,000
	Wiese [27]	MLP (3 FC)	Q-learning	0.07×10^6	—
	Samir [21]	LSTM (1 conv + LSTM + 2 FC)	Q-learning	0.69×10^6	—
	Goga [7]	MLP (6 FC)	Dueling and double DQN	0.30×10^6	≈ 12000[(b)]
	this work	CNN (2 conv (2×2) + 3 FC)	TD learning	2.90×10^6	228,100[(c)]
	this work	CNN (2 conv (2×2) + 3 FC)	TD learning	2.90×10^6	406,717[(d)]

Abbreviations: conv = convolution layer(s), FC = full-connect layer(s), policyAS = policy with afterstates

[(a)] Number of weights are calculated by the author [(b)] Calculated from the achievement ratios

[(c)] With the greedy (1-ply search) play [(d)] With the expectimax (3-ply search) play

with 400×10^8 positions [10] or for 32 hours with 400×10^8 positions [16]. Therefore, the training of DNNs was 2–8 times as slow as that of NTNs (if we compare the number of positions, the factor became 100–400). The computational cost of DNNs was also large in playing. For a move in the greedy play, DNNs took 1.5 ms while NTNs took 3.9–9.5 μs (a factor of 160–380). For a move in the expectimax 3-ply search, DNNs took 85 ms while NTNs took 0.7–4.4 ms (a factor of 19–130). [*5]

5. DNN-based Players for Game 2048

Since 2048 is a simple and interesting game, it is often used in teaching programming and machine-learning technologies [9]. We can find several programs and reports online, some of which were developed in course projects [1], [2], [11], [20]. In this section, we review existing DNN-based players for 2048.

Many DNN-based players were developed in the value-network approach [1], [7], [8], [11], [20], [21], [25], [26], [27]. The work most related to this study was by Guei et al. [8]. They designed CNNs with 2×2 and 3×3 filters and applied Q-learning and TD learning on the afterstates (states after slide&merge). Though we cannot find the details of training methods in the paper, the result was not so good and the average score was about 11,400 in the best case. The idea of input encoding in this paper has been widely used in many other implementations.

The best score achieved with value networks were by tjwei [25]. The network is a CNN with 1×2 filters applied in two ways (we extended the idea in the design of CNN12 network). The average score was 85,351 with a supervised learning method while it was about 33,000 with a reinforcement learning method. The idea of this CNN structure was used in several other implementations [11], [16], [20], [26]. The player developed by Qiu et al. [20] achieved a 4,096-tile for 34% of games, which was the best result of the DNN-based players trained by a reinforcement learning method.

[*5] The factor was larger in the case of the greedy play due to the slow processing in Python code. The 3-ply expectimax search player with DNNs generated all the possible states (< 65,536) and evaluated them in a batch.

In addition to CNNs and MLPs [7], [27], some other enhanced networks were tested: for example long short-term memory (LSTM) networks [21] and residual networks (ResNet) [2]. These enhanced networks did not perform well in particular with reinforcement learning methods.

A few implementations have been reported for the policy networks [6], [12], [16]. The author took the approach of policy network trained by a supervised learning method in the previous work and achieved state-of-the-art results for DNN-based players for 2048. We first examined CNNs with 2–9 convolution layers [12] and obtained an average score of 93,830 with five convolution layers. We analyzed the inner-workings of these CNNs and found an interesting difference between 2-layer and 3-layer networks [17]. We then examined two more network designs (CNN12 and MLP for policy networks) as well as increasing the number of weights [16]. We found CNN12 worked well in the supervised learning setting, and achieved an average score of 215,802 with the input extended with afterstates (called policyAS).

6. Conclusion

In this study, we investigate the reinforcement learning methods for DNN-based value networks for the game 2048. We designed several options based on the TD-afterstate algorithm [24] and selected the following ones after experiments: (1) fixed-size batch (N = 1,024), (2) no use of symmetry in generator's plays nor in data augmentation, (3) a CNN with 2×2 filters, and (4) with the restart strategy and the jump-start strategy. Under this configuration, we executed the training program for 120 hours. With the best value network after the 120-hour training, we achieved an average score of 184,546 with the greedy (1-ply search) play and 386,973 with the expectimax (3-ply search) play. We achieved much better results than the state-of-the-art DNN-based players [16], [20], [25] with combination of DNN-based evaluation functions and a game-tree search method.

Since we have succeeded in developing a DNN-based player comparable to NTN-based ones, it would be an interesting future work to compare DNNs and NTNs in details. In particular, by us-

ing the analysis methods [17] of the inner-workings of DNNs, we would like to evaluate the generalization ability of DNNs. Another future direction is to extend our DNN-based players in several aspects: for example with dual-head networks and Monte-Carlo tree search techniques in AlphaZero [22].

Acknowledgments Part of this work was supported by JSPS KAKENHI Grant Number JP20K12124. The experiments in this paper were conducted with the support of the IACP cluster in Kochi University of Technology.

References

[1] Adamkiewicz, M.: Q-learning for 2048: Exploring combinations of reinforcement learning and game tree search (2018), available from ⟨http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/mikadam/poster.pdf⟩.
[2] Allik, K., Rebane, R.-M., Sepp, R. and Valgma, L.: 2048 Report, available from ⟨https://neuro.cs.ut.ee/wp-content/uploads/2018/02/alphago.pdf⟩ (2018).
[3] Ballard, B.W.: The *-minimax search procedure for trees containing chance nodes, *Artificial Intelligence*, Vol.21, No.3, pp.327–350 (1983).
[4] Cirulli, G.: 2048, available from ⟨http://gabrielecirulli.github.io/2048/⟩ (2014).
[5] David, O.E., Netanyahu, N.S. and Wolf, L.: DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess, *International Conference on Artificial Neural Networks and Machine Learning* (*ICANN 2016*), pp.88–96 (2016).
[6] Dedieu, A. and Amar, J.: Deep Reinforcement Learning for 2048 (2017), available from ⟨http://www.mit.edu/˜amarj/files/2048.pdf⟩.
[7] Goga, A.: Reinforcement learning in 2048 game, Bachelor thesis of Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava (2018).
[8] Guei, H., Wei, T., Huang, J.-B. and Wu, I.-C.: An Early Attempt at Applying Deep Reinforcement Learning to the Game 2048, *Workshop on Neural Networks in Games* (2016).
[9] Guei, H., Wei, T.-H. and Wu, I.-C.: 2048-like games for teaching reinforcement learning, *ICGA Journal*, Vol.42, No.1, pp.14–37 (2020).
[10] Jaśkowski, W.: Mastering 2048 with Delayed Temporal Coherence Learning, Multi-Stage Weight Promotion, Redundant Encoding and Carousel Shaping, *IEEE Trans. Computational Intelligence and AI in Games*, Vol.10, No.1, pp.3–14 (2018).
[11] Ji, Y., Wang, C. and Zhang, S.: Designing an AI Agent for Game 2048 (2018), available from ⟨http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/cwang17/poster.pdf⟩.
[12] Kondo, N. and Matsuzaki, K.: Playing Game 2048 with Deep Convolutional Neural Networks Trained by Supervised Learning, *Journal of Information Processing*, Vol.27, pp.340–347 (2019).
[13] Lai, M.: Giraffe: Using Deep Reinforcement Learning to Play Chess, Master's thesis, Imperial College London, arXiv, Vol.1509.01549v1 [cs.AI] (2015).
[14] Li, J., Koyamada, S., Ye, Q., Liu, G., Wang, C., Yang, R., Zhao, L., Qin, T., Liu, T.-Y. and Hon, H.-W.: Suphx: Mastering Mahjong with Deep Reinforcement Learning, arXiv, Vol.2003.13590 [cs.AI] (2020).
[15] Matsuzaki, K.: Developing 2048 Player with Backward Temporal Coherence Learning and Restart, *Proc. 15th International Conference on Advances in Computer Games* (*ACG2017*), pp.176–187 (2017).
[16] Matsuzaki, K.: A Further Investigation of Neural Network Players for Game 2048, *Proc. 16th International Conference on Advances in Computer Games* (*ACG2019*) (2019). Submitted for final publication.
[17] Matsuzaki, K. and Teramura, M.: Interpreting Neural-Network Players for Game 2048, *Proc. 2018 Conference on Technologies and Applications of Artificial Intelligence* (*TAAI 2018*), pp.136–141 (2018).
[18] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M.: Playing Atari With Deep Reinforcement Learning, *NIPS Deep Learning Workshop* (2013).
[19] Moravcík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. and Bowling, M.H.: DeepStack: Expert-level artificial intelligence in heads-up no-limit poker, *Science*, Vol.356, No.6337, pp.508–513 (2017).
[20] Qiu, R., Tian, Y. and Zhou, Y.: The 2048 Challenge (2019), available from ⟨https://stanford-cs221.github.io/autumn2019-extra/posters/4.pdf⟩.
[21] Samir, M.: 2048 Deep Recurrent Reinforcement Learning, available from ⟨https://github.com/Mostafa-Samir/2048-RL-DRQN⟩.
[22] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, arXiv, Vol.1712.01815 [cs.AI] (2017).
[23] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. and Hassabis, D.: Mastering the game of Go without human knowledge, *Nature*, Vol.550, pp.354–359 (2017).
[24] Szubert, M. and Jaśkowski, W.: Temporal Difference Learning of N-Tuple Networks for the Game 2048, *2014 IEEE Conference on Computational Intelligence and Games*, pp.1–8 (2014).
[25] tjwei: A Deep Learning AI for 2048, available from ⟨https://github.com/tjwei/2048-NN⟩.
[26] Virdee, N.: Trained A Convolutional Neural Network To Play 2048 using Deep-Reinforcement Learning (2018), available from ⟨https://github.com/navjindervirdee/2048-deep-reinforcement-learning⟩.
[27] Wiese, G.: 2048 Reinforcement Learning, available from ⟨https://github.com/georgwiese/2048-rl⟩.
[28] Wu, I.-C., Yeh, K.-H., Liang, C.-C., Chang, C.-C. and Chiang, H.: Multi-Stage Temporal Difference Learning for 2048, *Technologies and Applications of Artificial Intelligence*, Lecture Notes in Computer Science, Vol.8916, pp.366–378 (2014).
[29] Yakovenko, N., Cao, L., Raffel, C. and Fan, J.: Poker-CNN: A Pattern Learning Strategy for Making Draws and Bets in Poker Games, arXiv, Vol.1509.06731 [cs.AI] (2015).

Kiminori Matsuzaki is a Professor of Kochi University of Technology. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming, algorithm derivation, and game programming. He is also a member of ACM, JSSST, and IEEE.