# Reinforcement Learning to Create Value and Policy Functions using Minimax Tree Search in Hex

Kei Takada, Hiroyuki Iizuka and Masahito Yamamoto

***Abstract*—Recently, the use of reinforcement-learning algorithms has been proposed to create value and policy functions, and their effectiveness has been demonstrated using Go, Chess, and Shogi. In previous studies, the policy function is trained to predict the search probabilities of each move output by Monte Carlo Tree Search; thus, a number of simulations are required to obtain the search probabilities. We propose a reinforcement-learning algorithm with game of self-play to create value and policy functions such that the policy function is trained directly from the game results without the search probabilities. In this study, we use Hex, a board game developed by Piet Hein, to evaluate the proposed method. We demonstrate the effectiveness of the proposed learning algorithm in terms of the policy function accuracy, and play a tournament with the proposed computer Hex algorithm DeepEZO and 2017 world-champion programs. The tournament results demonstrate that DeepEZO outperforms all programs. DeepEZO achieved a winning percentage of 79.3% against the world-champion program MoHex2.0 under the same search conditions on $13 \times 13$ board. We also show that the highly accurate policy functions can be created by training the policy functions to increase the number of moves to be searched in the loser position.**

***Index Terms*—Reinforcement Learning, Hex, Value Function, Policy Function.**

## I. Introduction

HEX is a two-player board game that was invented independently by Piet Hein and John Nash [1]. It is one of the board games that has been included in the Computer Olympiad, which is a game-based event involving competition between computer programs. Typically, computer Hex algorithms employ alpha-beta minimax, e.g., EZO-CNN, or Monte Carlo tree search (MCTS), e.g., MoHex, methods as game tree search algorithms [2]. Similar to Go, Hex has large branching factors. For computer Hex algorithms to efficiently perform a deep search, it is important to develop methods that can appropriately select moves to be searched.

The creation of evaluation functions, i.e., value and policy functions, is essential to develop effective artificial intelligence (AI) algorithms for game tree searches. The value function scores a position, and its value is used to determine the next move; thus, an accurate value function results in efficient programs. The policy function evaluates the next candidate moves, after which it determines the moves to be searched and the game tree search order. Moves are pruned based on the policy function, which is called forward-pruning. If the accuracy of the policy function is low, better moves may be pruned, which will negatively affect search results. The development highly accurate evaluation functions is therefore required to obtain superior computer AI algorithms.

Traditionally, evaluation functions have been created by combining manually quantified position and move features. Many methods have been proposed for creating evaluation functions using extracted features, and their effectiveness has been demonstrated [3], [4]. However, it is difficult to create highly accurate functions using such extracted features. Recently, the use of convolutional neural networks (CNNs), which can learn such features, has been proposed for many games [5]. It has been shown that the evaluation accuracy of functions that use CNNs is greater than that of traditional evaluation functions [6], [7].

Reinforcement-learning algorithms using games of self-play have attracted attention because they may be able to outperform manual evaluations. It has been demonstrated that employing reinforcement learning to train functions with a CNN is effective for several board games, e.g., Hex and Go [8], [9]. Silver et al. proposed a reinforcement-learning algorithm that creates a value function (*Value*) and a policy function (*Policy*). This algorithm has been extremely successful with Go (AlphaGo Zero [10]), Chess, and Shogi (AlphaZero algorithm [11]). These methods require the computation of search probabilities of each move output by the MCTS to train the policy function, which is trained to predict the search probabilities. Therefore, a number of simulations are required to determine the search probabilities and to train the policy function. The learning algorithm that does not require the search probabilities can reduce the number of simulations and computational cost for the learning.

In this paper, we propose a reinforcement-learning algorithm using games of self-play to create value and policy functions using a CNN in Hex. The self-play player uses the minimax tree search of depth one with forward-pruning based on the moves determined by the policy function. The primary difference between our proposed method and existing methods (i.e., AlphaGo Zero and AlphaZero algorithm) is the method to create the policy function, and our proposed algorithm does not use the search probabilities for learning. When the policy function does not use the search probabilities, it has to be trained using the state evaluation values of the evaluation functions, such as DDPG [12]. The policy always follows the value function. In the case of the game tree search, the policy

The authors are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Hokkaido, Japan (e-mail: takada@complex.ist.hokudai.ac.jp; iizuka@complex.ist.hokudai.ac.jp; masahito@complex.ist.hokudai.ac.jp)

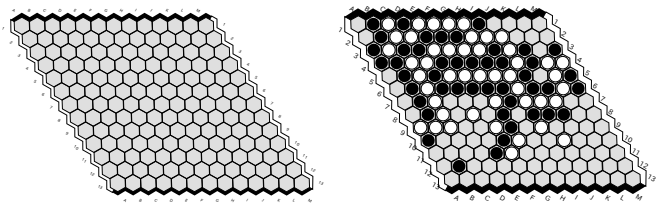
function can be trained to predict the best moves of the search results by the value function. However, there are bad moves that have resulted in losses in those search results, and learning bad moves may decrease the ability to select the good move for the policy function. Therefore, in our proposed algorithm, the policy function is trained to predict the search result of the minimax tree search in the winner position in the case where the next player won, and to increase the number of moves to be searched in the loser position in the case where the next player lost. In order to demonstrate the effectiveness of the proposed algorithm, we compare the proposed algorithm with the learning algorithm having the policy function that estimates the value functions regardless of the winner or loser positions. Another difference with previous studies (i.e., AlphaGo Zero) is that the value function is trained based on the depth one special case of the TreeStrap algorithm [13], which can be regarded as a type of Value Iteration [14]. We implemented our proposed methods on the Hex player, and developed the computer Hex algorithm called DeepEZO, which we compared with world-champion programs in 2017 to demonstrate the performance of DeepEZO.

The remainder of this paper is organized as follows. In Section II, we review the rules and features of Hex and conventional evaluation functions. In Section III, we propose the value and policy functions and a reinforcement-learning algorithm. In addition, we describe how games of self-play are played and how the functions are trained. In Section IV, our proposed algorithm is compared with the conventional policy training method. Those algorithms were implemented on computer Hex algorithms called DeepEZO (using our learning algorithm) and DeepEZO-Cross (the same, except for the learning algorithm for the policy), and compared them to show that the evaluation accuracy of the policy function created by our proposed algorithm is higher. In Section V, we develop DeepEZOs, which perform deep searches, and the tournaments are played by DeepEZOs and world-champion programs. We show that DeepEZOs have high winning percentages compared with the other programs under the same search conditions, and that DeepEZO using the evaluation functions created by the proposed algorithm has a high elo score. A general discussion is given in Section VI, and conclusions are presented in Section VII, including suggestions for future work.

## II. HEX

### A. Rules and Features

Hex is played on a rhombic board consisting of hexagonal cells. The game was developed for an $n \times m$ board (where $n$ and $m$ are natural numbers); however, an $n \times n$ board is generally used (e.g., Fig. 1a shows a $13 \times 13$ board). Recently, both $11 \times 11$ and $13 \times 13$ boards have been used in the Computer Olympiad [15]. Two players have uniformly colored pieces (e.g., black and white), and the game proceeds with players placing their stones in turn on empty cells. The two opposing black sides of the board are assigned to the black player, and the other opposing sides are assigned to the white player. The goal of the game is to connect the two opposing sides using the player's colored pieces. Fig. 1b shows a winning configuration for a white player. In this study, black was assigned as the first player and white was the second player.



(a) Hex game board (b) Game board showing a winning configuration for white

Fig. 1. $13 \times 13$ Hex board. The top and bottom sides are assigned to the black player, and the left and right sides are assigned to the white player. The player who connects the assigned sides using their stones wins.

It has been shown that there exists a winning strategy for the Hex game [16], i.e., the game result is determined based on the first move if two players play perfectly. Currently, a specific winning strategy for all first moves has been demonstrated for $9 \times 9$ or smaller boards [17]. It has also been shown that the game cannot end in a draw, and that the game is a PSPACE-complete problem [18], [19]. The first computer Hex algorithm was developed by Claude Shannon and E.F. Moore in 1953; [20] since then, many computer Hex algorithms have been developed.

### B. Conventional Methods to Build Value Functions

In Hex, the value and policy functions can be designed based on the network characteristics of the position. Positions on the Hex board can be expressed as a network by considering the cells as nodes and connecting adjacent nodes with links (or edges) [21]. Position features can be quantified by calculating the network characteristics based on the board graphs. A previous study proposed and demonstrated the effectiveness of a value function that employs an electric resistance model that considers the board graph as an electric circuit [22]. Wolve is a computer Hex algorithm that uses the electric resistance model, and it placed second at the 2012 Computer Olympiad [23]. EZO-CNN is a computer Hex algorithm that uses a value function based on 12 network characteristics. In addition, our submission placed second at the 2017 Computer Olympiad [2]. These computer Hex algorithms use iterative deepening depth-first search, and play the next move based on their evaluation functions.

The results of the recent Computer Olympiad indicate that the accuracy of evaluation functions that use the board graph is insufficient [2]. This is because EZO-CNN and Wolve cannot defeat MoHex2.0, which is based on MCTS. MoHex2.0 plays the next move based on the playout simulations, which improves the quality using expert game records. As a result, a CNN has been proposed to create more accurate evaluation functions [24], [25]. The CNN is expected to learn position features that cannot be represented by network characteristics.

Several reinforcement-learning algorithms that train value and policy functions independently have been proposed [8],

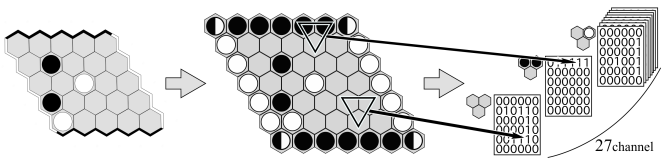



Fig. 2. Creation of position input. The left diagram shows positions on a $5 \times 5$ board. The middle diagram shows the sides in the left diagram with stones. Here, cells on corners are considered cells in which both black and white stones are placed. The right diagram is the input of the left diagram. Places corresponding to three mutually adjacent cells patterns become one.

[24], and it has been demonstrated that CNNs provide greater evaluation accuracy than that of classical evaluation functions. In addition, the reinforcement-learning algorithm called Expert Iteration(ExIt), which trains two functions, has been proposed; the effectiveness of ExIt is shown on a $9 \times 9$ board [26]. ExIt is a method that is similar to the AlphaZero algorithm, and uses the search probability output from MCTS to train the policy function.

To train the value function, the TD-Leaf and TreeStrap algorithms have been proposed in chess [13], [27]. For TD-Leaf, when $s_t$ is defined as the position at the $t$-th turn, the weights of the value function are updated to move the evaluation value of minimax search at $s_t$ towards the evaluation value of minimax search at $s_{t+1}$. For the TreeStrap algorithm, the weights are updated to move the evaluation value of position $s$ towards the evaluation value of minimax search at $s$, where the position $s$ is the interior position within the tree search. In both methods, the learning is performed to move the evaluation value of the current position towards the evaluation value of the position after playing the moves, but the sensitivity to the strength of the player is different. In TD-Leaf, the quality of the player is important because the positions that occur at subsequent time steps are used for learning, and it is known that the convergence of learning takes time when the player is weak. However, TreeStrap is less sensitive to the strength of the player because the positions that are used are based on hypothetical minimax play.

## III. Proposed Reinforcement-Learning Algorithm with Games of Self-Play

We propose value and policy functions that use CNNs. The functions are trained using games of self-play with minimax tree search. The value function is trained to predict the game result, and the policy function is trained to determine appropriate moves to be searched for each position.

In this section, we first describe the proposed functions and how the CNN input is created. Then, we describe the self-play player, propose the learning algorithm, and describe the loss function of the proposed functions.

### A. Representation of Board Position for CNN

We used the input proposed in our previous study, and expect this input to make it easier to learn the cell adjacency [25]. Next, we describe how to create the previously proposed CNN input.

TABLE I
Network Structure of Proposed Value Function.

| Layer type | Image size | Channel | Kernel size |
|---|---|---|---|
| Input | $14 \times 14$ | 27 | – |
| Conv 1 | $12 \times 12$ | 128 | $3 \times 3$ |
| Conv 2 | $10 \times 10$ | 128 | $3 \times 3$ |
| Conv 3 | $8 \times 8$ | 128 | $3 \times 3$ |
| Conv 4 | $6 \times 6$ | 128 | $3 \times 3$ |
| Conv 5 | $4 \times 4$ | 128 | $3 \times 3$ |
| Conv 6 | $2 \times 2$ | 128 | $3 \times 3$ |
| Conv 7 | $1 \times 1$ | 128 | $2 \times 2$ |
| Full Connection | – | 168 | – |
| Output | – | 1 | – |

In Hex strategy, it is important to consider cell adjacency because the goal of Hex is to connect two opposite sides. To make it easier to learn cell adjacency, we focus on three mutually adjacent cells to create the input. A cell can take three states corresponding to the placement of a player's stone, placement of the opponent's stone, and where no stone is placed. Therefore, the combined states of three adjacent cells yield 27 patterns. Here, the position is represented by 27 channels, and each pattern forms a channel. Fig. 2 shows an example of creating inputs from a position at which the black player makes a move. In each channel, the number corresponding to a specific channel pattern becomes one, and the others remain zero.

When the white player has the next move, we reflect the board about one of the diagonals, and swap the black and white cells because the Hex board is symmetric. When the board is reflected, the left and right sides become the top and bottom sides, respectively. This configuration is obtained to eliminate the learning cost required to consider that the side to be connected by the player is different.

### B. Proposed Value Function Using CNN

The network structure of the proposed value function is similar to that of our previous study, with the exception of the parameters [8]. The proposed value function takes the board position as the input, and outputs a scalar evaluation value of the given position. Table I shows the network structure for the proposed value function. The value function consists of convolution layers and a fully connected layer, and all activation functions are rectified linear units (ReLU) [28]. The stride size is one for all convolutional layers. In the output layer, the output range is 0 to 1 because a sigmoid activation function is used. The value function aims to output 1 if the next player has a winning position, and 0 if the next player has a losing position.

### C. Proposed Policy Function Using CNN

The network structure of the proposed policy function is similar to that of our previous study [25]. The proposed policy function takes the board position as its input and outputs the probability distribution of all next candidate moves. Table II shows the network structure for the proposed policy function. The policy function consists of only convolutional layers, and

TABLE II
NETWORK STRUCTURE OF PROPOSED POLICY FUNCTION.

| Layer type | Image size | Channel | Kernel size |
|---|---|---|---|
| Input | 14 × 14 | 27 | – |
| Conv 1 - 7 | 14 × 14 | 128 | 3 × 3 |
| Conv 8 | 13 × 13 | 128 | 2 × 2 |
| Output | 13 × 13 | 1 | – |

all activation functions in each convolutional layer are ReLUs. Convolutional layers 1 to 7 use $3 \times 3$ filters with a stride and zero-pad of one, and convolutional layer 8 uses $2 \times 2$ filters with a stride of one. We introduce a 10% SpatialDropout to the input layer to increase generalization ability [29]. In the output layer, the probability distribution is output using the softmax function.

### D. Self-Play Player with Proposed Selective Search

The self-play player uses a minimax tree search of depth one with forward-pruning. Here, the moves to be searched are determined by the proposed policy function. The output of the policy function is a probability distribution of next moves, and only moves with a high-probability value are searched. Specifically, moves for which the probability value exceeds $1/C$ are searched, where $C$ is the number of cells on the board (e.g., $C = 169$ on a $13 \times 13$ board). Exceptionally, if the number of moves that exceed the threshold is less than three, the three highest moves are searched, i.e., the minimum search width is three. When there are no more than three empty cells at the given position, the player searches all empty cells. There are no specific methods available for determining the minimum search width. Here, it was determined by considering the balance between the risk of pruning good moves and the efficiency of the search carried out by reducing the number of moves to be searched. Where the selected moves are played from the root position, each position is evaluated by the proposed value function. The move with the highest evaluation value is selected as the next move.

### E. Proposed Learning Algorithm with Games of Self-Play

Algorithm 1 shows the proposed learning algorithm. Here, positions that have appeared in games of self-play are used to train the proposed functions. We define a position at the $t$-th turn as $s_t$, a move at $s_t$ as $a_t$, and the evaluation value at $s_t$ as $v(s_t)$. In addition to the position-move pairs in games of self-play, the rotated position-move pairs $(s_t^{rotated}, a_t^{rotated})$ are used as training data because they are essentially the same as the original pairs. The rotation operation is similar to the method described in Section IV A. The training data set $\mathbf{D}$ is initialized ($\mathbf{D}$ becomes an empty set) for each epoch because high-quality data are required to create highly accurate functions.

Because learning various positions leads to improved generalization ability for the proposed functions, the following techniques are introduced to games of self-play in order to increase the number of training positions.

**Algorithm 1** Proposed learning algorithm.

**function** LEARNING
  **for** $epoch = 0$ to $E$ **do**
    Initialize training data set $\mathbf{D}$
    **for** $game = 0$ to $G$ **do**
      $p1$ uses the latest functions
      $p2$ uses the latest or past functions
      Randomly select which player is first
      Get random number $T$
      $t \leftarrow T$
      $s_t \leftarrow$ GETSTARTINGPOSITION($p1$,$p2$,$T$)
      **while** *not terminal* $s_t$ **do**
        $p \leftarrow$ Next player selected from $p1$ and $p2$
        $p$ searches best move $a_t$ at $s_t$
        $(s_t, a_t)$ is added to $\mathbf{D}$
        $(s_t^{rotated}, a_t^{rotated})$ is added to $\mathbf{D}$
        $s_{t+1} \leftarrow$ Position playing $a_t$ at $s_t$
        $t \leftarrow t + 1$
      **end while**
      ($s_t$,$\emptyset$) and ($s_t^{rotated}$,$\emptyset$) are added to $\mathbf{D}$
    **end for**
    Update the proposed functions using $\mathbf{D}$
  **end for**
**end function**
**function** GETSTARTINGPOSITION($p1$,$p2$,$T$)
  $s_0 \leftarrow$ Initial position
  $s_1 \leftarrow$ Position playing the random move at $s_0$. The move is selected from the cells within two rows from the side.
  **if** $T$ is 1 **then**
    **return** $s_T$
  **end if**
  **for** $i = 1$ to $T - 1$ **do**
    $p \leftarrow$ Next player selected from $p1$ or $p2$
    $p$ searches best move $a_i$ at $s_i$
    $s_{i+1} \leftarrow$ Position playing $a_i$ at $s_i$
  **end for**
  $s_T \leftarrow$ Position playing random move at $s_{T-1}$
  **return** $s_T$
**end function**

*1) Player Randomness:* We introduce two randomness techniques to a player's search, one of which is related to the moves to be searched. The player searches moves with a low-probability value. Moves with a probability value that is less than $1/C$ are searched with a probability of 20%. This is done to avoid searching only the currently favored moves. This operation is not performed to play the next move randomly, as with $\epsilon$-greedy, but to determine the moves to be searched that are evaluated by the value function. The other technique is related to the evaluation value output by the value function. A small random number is added to the evaluation value. The value function has an output ranging from 0 to 1 because it uses the sigmoid function. The range of the random number to be added is -0.05 to 0.05. This range was determined empirically; however, by adding this random number, the player can approximate random play if they use

an untrained instance of the proposed value function.

*2) Random First Move:* The first moves of games of self-play are selected randomly from cells within two rows from the side of the board. In Hex, first moves near the center of the board are advantageous for the first player. When starting a game near the center, it is expected that many favorable positions will appear for the first player. The value function must accurately evaluate positions where it is difficult to determine which player has an advantage. Learning from many difficult positions can be expected to lead to accurate evaluations by the proposed value function; thus, games of self-play begin at cells within two rows from the side of the board. The number of cells at the start of the game is 88 in a $13 \times 13$ board ($a1$ to $a13$, $b1$ to $b13$, $l1$ to $l13$, $m1$ to $m13$, $c1$ to $k1$, $c2$ to $k2$, $c12$ to $k12$ and $c13$ to $k13$).

*3) Random Move During Games of Self-play:* A random move is played during a game of self-play. The self-play players play a game over $T$ moves until position $s_T$. Here, $T$ is a random integer ($1 \leq T \leq 60$). The maximum value of $T$ was determined empirically, and it is possible that the game will end within the maximum $T$ value even if the player plays randomly. At position $s_T$, the next move $a_T$ is played randomly from all empty cells. The self-play players play the game from position $s_{T+1}$ after playing $a_T$ at $s_T$ until the game terminates.

*4) Older Trained Functions:* The most recent functions and older functions are used in games of self-play. The proposed functions are stored every 25 epochs. Here, an epoch is the period required to learn positions that have appeared in a given number of games of self-play. One player always uses the most recent value and policy functions. However, another player may use older functions. The functions to be used are selected randomly from the latest, second, or third latest functions, and the selection of these functions is performed randomly for each game.

*5) Determined Move at Position Where Winning Connection Exists:* The *winning connection* is a special *virtual connection*. The virtual connection is a link between two cell groups that can be connected by playing the best move even if the player has the second move at the current positions [21]. Virtual connections can be found using an h-search algorithm, and the virtual connection between two opposite sides is referred to as a winning connection [30]. A winning connection indicates that either player can connect two opposite sides by playing perfectly, where the best move is found by the h-search algorithm. Therefore, if one player has found a winning connection by performing an h-search, the game tree search is not necessary. At the position where a winning connection exists, the player with the winning connection plays the winning move to connect two opposite sides, and the other player plays the most obstructive move (referred to as the *mustplay* [31]).

### F. Loss Functions

In games of self-play, the weights of the proposed functions are updated by the stochastic gradient descent method using the position-move pairs ($s$,$a$). Here, we describe the loss function of each function.

*1) Value Function:* The proposed value function is trained by the depth one special case of the TreeStrap algorithm [13], and predicts the game's result. The reason for which the TreeStrap algorithm is used is that the method that is less sensitive to the strength of the player is necessary because the value function is initialized to random weights in the proposed algorithm. We give the final reward to only terminal positions. The loss function $Loss_{eval}$ is minimized and defined as follows.

$$Loss_{eval} = \sum_{(s,a)\in\mathbf{D}} loss_e(s,a), \tag{1}$$

$$loss_e(s,a) = \begin{cases} -\log(1-v(s)) & (s \text{ is terminal}) \\ (v(s)-v(s'))^2 & (\text{otherwise}), \end{cases} \tag{2}$$

where $\mathbf{D}$ is a set of position-move pairs ($s$,$a$) in games of self-play, $v(s)$ is the evaluation value of the position output by the proposed value function at position $s$, and $s' = m(s,a)$ is the position at which move $a$ was played at position $s$. In terminal positions, the player who lost the game has the next move. Thus, the proposed value function should output zero at the terminal positions in $\mathbf{D}$.

A log loss is used at terminal positions to evaluate the terminal position accurately. The error given to the value function at the terminal position is larger than that when using the squared temporal-difference error. The value function is trained more strictly to the terminal positions.

*2) Policy Function:* To determine the moves to be searched according to the positions, the loss function of the proposed policy function differs depending on whether the position is a " winner position " where the next player won or a 'loser position' where the next player lost. If the training position is the winner position, the policy function learns to reduce the search width because the search width was sufficient. However, if the training position is a loser position, the policy function learns to increase the search width. The reason for increasing the search width in the loser position is to search for better moves that are pruned from the tree search. The policy function is used as the deciding search width for the player; therefore, one of the reasons for losing the game is that the policy function has a low evaluation accuracy, and prunes the better moves from the search. To play a good move in the loser position, it is necessary to search widely. By changing the learning policy according to the position, it becomes possible to determine the number of moves to be searched.

The proposed policy function is trained to minimize the following loss function $Loss_{policy}$:

$$Loss_{policy} = \sum_{(s,a)\in\mathbf{D}} loss_p(s,a), \tag{3}$$

$$loss_p(s,a) = \begin{cases} -\log \pi(a,s) & (s \text{ is winner position}) \\ -\lambda \sum_{m\in A} \pi(m,s) \log \pi(m,s) & (\text{otherwise}), \end{cases} \tag{4}$$

where $A$ is a set of all moves on the board, $\pi(a,s)$ is the output of the proposed policy function for move $a$ at position $s$, and $\pi(a,s)$ is the probability value. Here, $\lambda$ is a constant parameter

that controls the search width, and we used $\lambda = 10^{-1}$ in this study. As a result, we minimize the cross-entropy loss in the winner position and maximize the entropy loss in the loser position.

## IV. Training the Proposed Functions

We trained the value and policy functions using the proposed learning algorithm (Section III), and developed the computer Hex algorithm DeepEZO using the proposed functions. To demonstrate that the learning is properly performed by the proposed learning algorithm, we indicate the winning percentages of DeepEZO against the previous computer Hex algorithm MoHex2.0. To demonstrate the effectiveness of the proposed learning algorithm, the proposed algorithm is also compared with the learning algorithm, where the policy function predicts the moves selected by the search based on the value function.

The only difference between the proposed algorithm and the learning algorithm to be compared is the update method of the policy function. In the learning algorithm to be compared, the policy function is trained to predict the search results in both winner and loser positions, which means that the loss function uses only cross entropy loss. In our proposed algorithm, the policy function is trained to predict the good moves in the winner position, and to increase the number of moves to be searched in the loser positions. The policy function is used when deciding the moves to be searched; therefore, if we assume that the highly accurate value function is obtained, there is no reason to be lost, except for the cases where the policy function prunes the winning moves, or when it is impossible to win the game. In this sense, the moves in the loser positions should not be entrained, and the function should increase the number of moves to be searched.

In this section, we first describe the computer Hex algorithm used in this section, and then we train the proposed functions. Next, two policy functions are compared. We used a computer with an Intel(R) Core(TM) i7-7700KCPU (4.20GHz) and an Nvidia GTX 1080 GPU. The CNN models, value and policy functions, were developed using Torch [32].

### A. Computer Hex Algorithms

The computer Hex algorithms used in this study were implemented using the open-source Benzene framework [33], [34]. All players are allowed to prune provably inferior moves and play the mustplay [31]. These methods exclude moves that are meaningless to the search, and it allows the player to play definitive moves to win or lose.

*1) DeepEZO:* DeepEZO is the proposed computer Hex algorithm. DeepEZO uses the iterative deepening depth-first search as the game tree search algorithm [35], as well as the proposed value and proposed policy functions. The method employed to determine the moves to be searched is the same as that described in Section III-D. We do not add randomness to the search during the evaluation games. DeepEZO only searches moves with a high-probability value from the policy function (no other moves are searched). Here, the minimum search width is three.

We also prepared DeepEZO-Cross, where the policy function is obtained by the learning with only cross entropy. The evaluation functions used for DeepEZO-Cross are obtained by the learning, where only the cross entropy loss is used as the loss function of the policy function. The policy function is always trained to predict the search results regardless of the winner or loser positions. DeepEZO and DeepEZO-Cross use the same algorithms, except for the value and policy functions.

*2) MoHex2.0:* MoHex2.0 uses the Monte Carlo tree search as the game tree search algorithm [36]. MoHex2.0 has been the world-champion program at the Computer Olympiad (2013 to 2017), and is the strongest computer Hex algorithm.

MoHex-CNN was the world-champion program for the $13 \times 13$ board at the 2017 [2], [37]. To demonstrate the effectiveness of the proposed functions, it is important to compare the proposed functions with the currently best program. However, MoHex-CNN is not currently available. Thus, MoHex2.0 was used in this study because it was the world-champion for the $11 \times 11$ in 2017, and is a sufficiently strong program.

In this paper, MoHex2.0 uses one thread for the search, and does not use pondering. This means that MoHex2.0 parameters "num_threads," "lock_free," and "ponder" are one, zero, and zero, respectively.

### B. Experiments

Our proposed functions were trained in reinforcement learning using self-play. We show that the learning is properly performed based on the winning percentages of DeepEZO against MoHex2.0. The effectiveness of our proposed training method is shown compared with that of DeepEZO-Cross in terms of the winning percentages against MoHex2.0.

In order to show that the trained policy function can prune bad moves and keep good moves properly, we indicate the difference in the winning percentages against MoHex2.0 between the two types of DeepEZO with a different search width (DeepEZO and DeepEZO-all). DeepEZO-all searches all next candidate moves at all positions. The search results of DeepEZO and DeepEZO-all are the same unless the policy function prunes the best move because they use the same value function. We also compared DeepEZO-Cross and DeepEZO-Cross-all, and the only difference between DeepEZO-Cross and DeepEZO-Cross-all is also the search width.

*1) Game Conditions Between DeepEZO and MoHex2.0 :* The games between DeepEZO (DeepEZO-all, DeepEZO-Cross, and DeepEZO-Cross-all) and MoHex2.0 started at all opening cells in the board. There are 169 opening cells on a $13 \times 13$ board. One game was played for the first and second players for each opening. A total of 338 games were played. The search depth of DeepEZOs was two, and the search time of MoHex2.0 was up to 30s per move. These computer Hex algorithms used one thread for the search, and they did not use a parallel solver.

*2) Training Conditions:* The number of games of self-play in a single epoch $G$ was 1,000, and the total number of epochs was 2,600, which means that 2.6 million games were played. The weights of the value and policy functions were

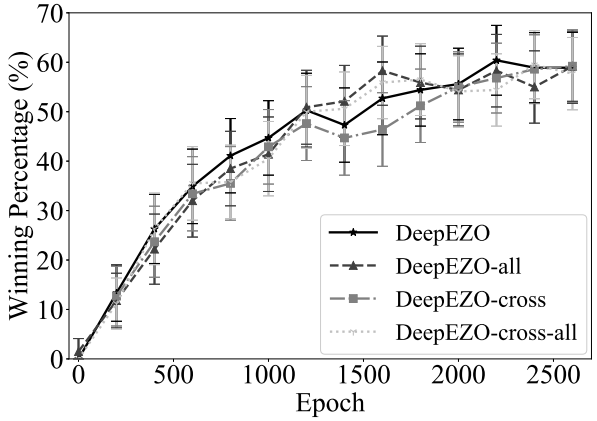



Fig. 3. Winning percentages of DeepEZO and DeepEZO-all compared with MoHex2.0 in the training process (error bar shows the standard error of games; 95% confidence).

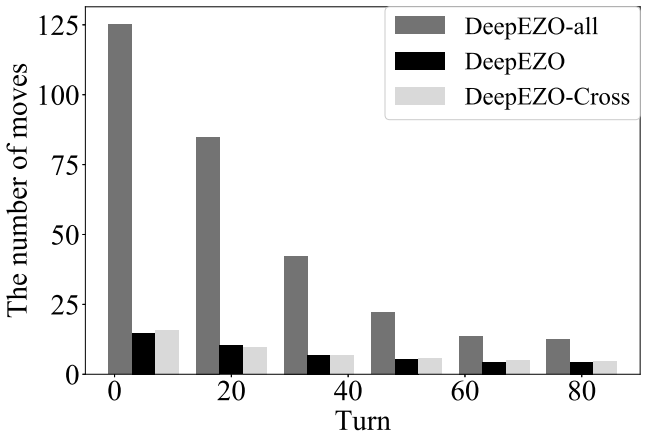



Fig. 4. Number of moves to be searched at each position classified based on the number of turns from the positions in games against MoHex2.0 in 2,600 epochs.

initialized to random weights. We used a mini-batch size of 32 positions and Adam [38] as the optimizer to train the proposed functions. Training was performed once.

*3) Training Result:* Fig. 3 shows the winning percentages of DeepEZOs against MoHex2.0 in the training process. The average total search time in a game for DeepEZO, DeepEZO-all, DeepEZO-Cross, DeepEZO-Cross-all, and MoHex2.0 was 30s, 295s, 26s, 283s, and 505s, respectively. The training periods were approximately 45 days and 54 days for DeepEZO and DeepEZO-Cross, respectively.

Both DeepEZO and DeepEZO-all had winning percentages greater than 50% against MoHex2.0, and the search time of DeepEZO was less than that of MoHex2.0. These results show that the learning of the proposed algorithm was properly performed, and the evaluation accuracy of the evaluation functions was high. In addition, the results confirm that the winning percentages of DeepEZO and DeepEZO-all are not significantly different, and that the search time of DeepEZO is less than that of DeepEZO-all. It is expected that our policy function can appropriately determine the moves to be searched.

There is no significant difference between the DeepEZO and DeepEZO-Cross algorithms. This result means that both methods can create a policy function that can appropriately determine moves to be searched and the value function with high evaluation accuracy. However, in this experiment, two policy functions may not be compared properly because DeepEZO and DeepEZO-Cross mainly select the next move based on the value function.

### C. Analysis of Policy Function

In this section, we analyze the trained policy function in terms of the number of moves to be searched and direct match performances in order to demonstrate the difference between the trained policy functions.

*1) Number of Moves to be Searched:* To show that the proposed policy function can reduce the number of moves to be searched, we compared the number of moves searched by DeepEZO, DeepEZO-all, and DeepEZO-Cross at each position in games against MoHex2.0 in 2,600 epochs. The number of moves to be searched for DeepEZO-all is equal to the number of possible moves. Fig. 4 shows the number of moves to be searched by DeepEZO, DeepEZO-Cross, and DeepEZO-all. The number of positions of DeepEZO, DeepEZO-Cross, and DeepEZO-all is 8,239, 8,516, and 16,381, respectively. All of the positions were categorized according to the number of turns, and the number of moves to be searched in each category was averaged. The results confirm that many moves were pruned from the game tree search by the proposed policy function, and there is no significant difference between DeepEZO and DeepEZO-Cross algorithms.

*2) Game of Two Policy Functions:* To demonstrate which policy function can select good moves, two policy functions were compared directly. We prepared two players who play the move with the highest evaluation value of the policy function, and two players play the games directly. $P_{prop}$ is the player who uses the policy function created by the proposed learning algorithm, and $P_{cross}$ is the player who uses the policy function created by the other learning algorithm. The game conditions are the same as those described in Section IV-B1, with the exception of the players used.

Fig. 5 shows the winning percentage of $P_{prop}$ against $P_{cross}$. The results confirm that $P_{prop}$ can realize a higher winning percentage against $P_{cross}$, which means that the evaluation accuracy of the policy function created by the proposed learning algorithm is higher than the policy function created by the learning in which the policy function is trained to always predict the search result.

## V. Computer Hex Algorithm Tournament

Here, we discuss the performance of DeepEZO and DeepEZO-Cross when performing the deep search in order to demonstrate that DeepEZO, which uses the evaluation functions created by the proposed algorithm, is superior than DeepEZO-Cross. In Section IV, we trained the proposed functions, and it is shown that the policy function created by the proposed algorithm has a higher evaluation accuracy. In addition, there is no significant difference between DeepEZO

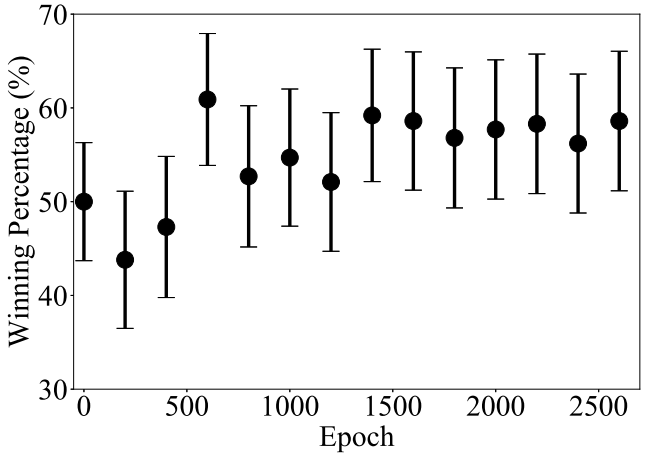



Fig. 5. Winning percentage of $P_{prop}$ against $P_{cross}$ in the training process (error bar shows standard error of games; 95% confidence).

and DeepEZO-Cross with a 2-ply search. There is no problem if the policy function can make the good moves the search target, even if the evaluation accuracy of the policy function is low because it is possible to finally select the good moves by performing the search with the value function. However, in a deep search, the policy function with a low evaluation accuracy may have the negative effect on the search result because the number of positions to be evaluated increases.

### A. Computer Hex Algorithms

To demonstrate the performance of DeepEZO under the same search conditions, we also prepared Wolve and EZO-CNN, which are classical computer Hex algorithms.

*1) EZO-CNN:* EZO-CNN uses iterative deepening depth-first search [25], and placed second at the 2017 Computer Olympiad. EZO-CNN was the strongest computer Hex algorithm of the programs based on the minimax tree search. The value function employed by EZO-CNN is an optimized function consisting of 12 network characteristics. EZO-CNN uses a policy function with a CNN created by supervised learning. The search width of EZO-CNN is constant at each position, which means that moves are searched in descending order of the probability value of the move obtained by the policy function.

*2) Wolve:* Wolve uses iterative deepening depth-first search, and was the second-place computer Hex algorithm at the 2012 Computer Olympiad [39]. The value and policy functions employed by Wolve are based on the electric resistance model, which is used by many other programs, such as Hexy and Six.

### B. Game Conditions

The tournaments were played by DeepEZO, DeepEZO-Cross, MoHex2.0, EZO-CNN, and Wolve. The search depth of EZO-CNN and Wolve was four, and the search width was eight. The search time of MoHex2.0 was up to 30s per move. To ensure equal conditions as the existing computer Hex algorithms, we prepared two types of DeepEZO, i.e., DeepEZO-4ply, which searched at a depth of four and has no time limit, and DeepEZO-30s, which searches for 30s per move and has no depth limit. We also prepared DeepEZO-Cross-4ply and DeepEZO-Cross-30s. In total, seven computer Hex algorithms were used in the tournament.

The games started at all opening moves in the board. Two games were played for the first and second player for each opening, and 676 games were played in total. With the exception of the number of games, the experimental conditions were similar to those described in Section IV-B1.

### C. Tournament Results

Table III shows the tournament results. The elo score is computed by BayesElo [40], and the standard error is about $\pm 11$ with 95% confidence. As can be seen, DeepEZO-30s has the highest elo score. To show the specific difference of the search between DeepEZO and DeepEZO-Cross, we indicate the number of positions to be evaluated and the search depth. The average number of positions to be evaluated in one position of DeepEZO-4ply and DeepEZO-Cross-4ply is 501.5 and 575.4, respectively. DeepEZO-Cross-4ply searches wider than DeepEZO-4ply, and this difference may be one of the reasons for which the average search time of DeepEZO-Cross-4ply is longer than that of DeepEZO-4ply. In addition, the average search depth in one position of DeepEZO-30s and DeepEZO-Cross-30s is 5.3 and 5.0, respectively. Further, the maximum search depth of DeepEZO-30s and DeepEZO-Cross-30s is 22 and 21, respectively. While the difference is small, it is shown that DeepEZO-30s searches deeply with the same search time.

DeepEZO-30s obtained a winning percentage of 79.3% against MoHex2.0 with the same search time. DeepEZO-4ply obtained a winning percentage of approximately 80.0% against programs based on the minimax tree search with the same search depth. DeepEZO obtained a very high winning percentage against world-champion programs, and it is obvious that the evaluation accuracy of the proposed evaluation functions is very high.

The search time of DeepEZO-30s was greater than that of MoHex2.0 even when the search time per move for each method was 30s because MoHex2.0 stops the search if the best move cannot change, even if it uses the remaining time. The average search time for each game differed; however, both programs required 30s per move.

## VI. Discussion

The experimental results indicate that the alpha-beta search using the proposed value and policy functions created by the proposed learning algorithm perform more pruning than the alpha-beta search using the functions created by the learning method that trains the policy function to predict the search result in both winner and loser positions. Fig. 4 shows that there is not much difference in the number of positions to be searched between two functions at each position. However, the number of positions to be evaluated for DeepEZO-4ply is smaller than that for DeepEZO-Cross-4ply. In addition, even within the same search time, DeepEZO-30s has a deeper search than DeepEZO-Cross-30s. These results indicate that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2019.2893343, IEEE Transactions on Games



TABLE III
WINNING PERCENTAGE % OF EACH TOURNAMENT (FIRST/SECOND PLAYER) FOR ROW PLAYER AGAINST COLUMN PLAYER AND SEARCH TIME OF EACH PROGRAM ($\pm$ IS STANDARD ERROR; 95% CONFIDENCE). STANDARD ERROR OF ELO IS ABOUT $\pm 11$ WITH 95% CONFIDENCE.

| | | A | B | C | D | E | F | G | Search time in one game | Elo score |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | **DeepEZO-30s** | - | $54.1 \pm 3.8$ (65.1/43.2) | $54.4 \pm 3.8$ (61.8/47.0) | $52.8 \pm 3.8$ (62.4/43.5) | $79.3 \pm 3.1$ (86.1/72.5) | $84.6 \pm 2.7$ (93.5/75.7) | $90.4 \pm 2.2$ (92.0/88.8) | 645.3 [sec] | 491 |
| **B** | **DeepEZO-4ply** | - | - | $52.1 \pm 3.8$ (63.0/41.1) | $48.2 \pm 3.8$ (57.4/39.1) | $75.3 \pm 3.3$ (80.5/70.1) | $80.6 \pm 3.0$ (89.3/71.9) | $86.2 \pm 2.6$ (88.2/84.3) | 488.7 [sec] | 450 |
| **C** | **DeepEZO-Cross-30s** | - | - | - | $52.1 \pm 3.8$ (64.2/39.9) | $79.6 \pm 3.0$ (83.1/76.0) | $84.5 \pm 2.7$ (89.3/79.6) | $89.2 \pm 2.3$ (93.2/85.4) | 655.4 [sec] | 465 |
| **D** | **DeepEZO-Cross-4ply** | - | - | - | - | $77.9 \pm 3.1$ (82.2/73.6) | $85.5 \pm 2.7$ (90.5/80.5) | $88.2 \pm 2.4$ (90.5/85.8) | 533.4 [sec] | 464 |
| **E** | **MoHex2.0** | - | - | - | - | - | $69.1 \pm 3.5$ (77.8/60.4) | $81.5 \pm 2.9$ (85.2/77.8) | 499.3 [sec] | 237 |
| **F** | **EZO-CNN** | - | - | - | - | - | - | $68.6 \pm 3.5$ (79.3/58.0) | 619.7 [sec] | 125 |
| **G** | **Wolve** | - | - | - | - | - | - | - | 327.5 [sec] | 0 |

DeepEZO-4ply performs more pruning than DeepEZO-Cross-4ply. It is believed that these results can be obtained because the policy function created by the proposed learning algorithm has a higher evaluation accuracy, as can be seen from Fig. 5. These are among the reasons for which DeepEZO-30s has the highest elo score from among all computer Hex algorithms. The reason for which the evaluation accuracy of the policy function created by the proposed learning algorithm is high is that only the good moves are learned. The moves at the loser position may be poor moves because they eventually lead to losses. It is considered that learning only good moves consistently improved the evaluation accuracy of the policy function.

The primary differences between the proposed learning method and the existing methods (i.e., AlphaGo Zero and the AlphaZero algorithm [10], [11]) are the game tree search algorithm and the method employed to create the policy function. In both methods, the policy function is created to increase the evaluation value of good moves and to reduce the value of bad moves. However, with the proposed method, the number of moves to be searched must be less than that of the existing methods. For each method, the number of moves to be searched depends on the evaluation value of the policy function, and moves with high values are more likely to be searched. The proposed method trains the policy function to increase the evaluation value of only the selected move in the winner position; thus, the number of moves to be searched tends to decrease at the winner position. However, the existing method trains the policy function to learn the search probability of each candidate move, and does not learn to increase the evaluation value of only the selected move. Of course, the number of moves to be searched will become small in the position where the winning move is obvious, but in the position where several actions are equally suitable, the number of moves to be searched will not become small. As a result, the proposed method may increase the risk of pruning the good moves by reducing the number of moves; however, there is a possibility that deeper searches and better move selection can be performed if the evaluation accuracy of the policy function is sufficiently high.

In terms of the training cost, the proposed learning algorithm may incur a lower cost than the AlphaZero algorithm. The AlphaZero algorithm requires the search probabilities of each candidate move to train the policy function. To obtain the search probabilities, many simulations are required, and this increases the computational cost. However, the proposed algorithm uses the search results instead of the search probabilities to train the policy function. The search result can be obtained from a 1-ply search, which means that each candidate move is sufficient to be evaluated only once. Because our proposed algorithm can reduce the number of evaluations and the computational cost, it may be possible to create the highly accurate evaluation functions more rapidly with the proposed learning algorithm.

We confirmed that the winning percentage of DeepEZO-30s against MoHex2.0 was greater than that of DeepEZO-4ply against MoHex2.0 (Table III). This shows that the strength of DeepEZO-30s was greater than that of a limited-width 4-ply search, and that the generalization ability of the proposed value function is very high. The search result of the minimax tree search changes if the evaluation of a certain position changes; therefore, the search result will be a worse move if the value function evaluates a certain position incorrectly. Although the number of positions to be evaluated is increased by performing a deep search, DeepEZO-30s can play a better move by evaluating the position appropriately.

Learning for the proposed functions required 45 days; however, this can be reduced easily. Most of the learning time was spent on games of self-play; therefore, the learning time can be reduced by increasing the speed of games of self-play. Each game is completely independent, and parallelization can be implemented easily. In this study, we used only a single CPU and a single GPU, and we expect that the learning time can be reduced considerably by parallelization of additional computational resources.

## VII. Conclusion

In this paper, we proposed a reinforcement-learning algorithm to create value and policy functions through games of self-play in Hex. In this study, games of self-play were played by players based on a minimax tree search. The value function was trained to predict the game result, and the policy function was trained to change the number of moves to be searched according to the given position. To demonstrate the effectiveness of the proposed learning algorithm, we compared the proposed learning algorithm with other learning algorithms. In addition, we developed the computer Hex algorithm DeepEZO using the minimax tree search, and compared it to other computer Hex algorithms. Based on the experimental results, we demonstrated that DeepEZO outperforms all of the other programs, which means that the trained value function has a high evaluation accuracy and generalization ability, and that the trained policy function can appropriately determine moves to be searched at each position. We also demonstrated that the policy function trained by the proposed learning algorithm has a higher evaluation accuracy than the policy function that is trained to always predict the search result.

In future, we aim to implement parallelization of the learning algorithm in order to improve the learning efficiency. In addition, we plan to investigate how the search depth in games of self-play affects the accuracy of the evaluation functions.

## Acknowledgement

This work was supported by Grant-in-Aid for JSPS Research Fellow Grand Number JP 16J02092 and the Global Station for Big Data and CyberSecurity, a project of the Global Institution for Collaborative Research and Education at Hokkaido University.

## References

[1] C. Browne, *Hex Strategy: Making the Right Connections*. A. K. Peters, Natick, MA, 2000.

[2] R. B. Hayward and N. Weninger, "Hex 2017: Mohex wins hex 11x11 and 13x13 tournaments," in *ICGA Journal*, vol. 39, 2017, pp. 222–227.

[3] K. Hoki and T. Kaneko, "Large-scale optimization for evaluation functions with minimax search," *Journal of Artificial Intelligence Research*, vol. 49, p. 527568, 2014.

[4] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[5] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, "Move evaluation in go using deep convolutional neural networks," in *3rd International Conference on Learning Representations*, 2015.

[6] R. Coulom, "Computing elo ratings of move patterns in the game of go," *ICGA Journal*, vol. 30, pp. 198–208, 2007.

[7] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, "Move evaluation in go using deep convolutional neural networks," in *3rd International Conference on Learning Representations*, 2015.

[8] K. Takada, H. Iizuka, and M. Yamamoto, "Reinforcement learning for creating evaluation function using convolutional neural network in hex," *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pp. 196–201, 2017.

[9] 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, 2016.

[10] 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. 354–359, 2017.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv:1712.01815*, December 2017.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[13] J. Veness, D. Silver, A. Blair, and W. Uther, "Bootstrapping from game tree search," in *Advances in Neural Information Processing Systems 22*, 2009, pp. 1937–1945.

[14] B. Scherrer, M. Ghavamzadeh, V. Gabillon, B. Lesner, and M. Geist, "Approximate modified policy iteration and its application to the game of tetris," *Journal of Machine Learning Research*, vol. 16, pp. 1629–1676, 2015. [Online]. Available: http://jmlr.org/papers/v16/scherrer15a.html

[15] R. Hayward, J. Pawlewicz, K. Takada, and T. van der Valk, "Mohex wins 2015 hex 11x11 and hex 13x13 tournaments," in *ICGA Journal*, vol. 39, 2017, pp. 60–64.

[16] J. Nash, "Some games and machines for playing them," Rand Corp D-1164, Tech. Rep., February 1952.

[17] J. Pawlewicz and R. B. Hayward, "Scalable parallel dfpn search," *Computers and Games CG2013 LNCS*, vol. 8427, pp. 138–150, 2013.

[18] S. Even and R. E. Tarjan, "A combinatorial problem which is complete in polynomial space," *Joutnal of ACM*, vol. 23, no. 4, pp. 710–719, October 1976.

[19] D. Gale, "The game of hex and the brouwer fixed-point theorem," *The American Mathematical Monthly*, vol. 86, no. 10, pp. 818–827, 1979.

[20] C. E. Shannon, "Computers and automata," in *Proceedings of the Institute of Radio Engineers*, vol. 41, no. 10, 1953.

[21] V. V. Anshelevich, "A hierarchical approach to computer hex," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 101–120, 2002.

[22] P. T. Henderson, "Playing and solving the game of hex," Ph.D. dissertation, University of Alberta, 2010.

[23] R. B. Hayward, "Mohex wins hex tournament," in *ICGA Journal*, vol. 35, no. 2, June 2012, pp. 124–127.

[24] K. Young, G. Vasan, and R. Hayward, "Neurohex: A deep q-learning hex agent," in *Compute Games Workshop, IJCAI*, 2016.

[25] K. Takada, H. Iizuka, and M. Yamamoto, "Computer hex using move evaluation method based on convolutional neural network," in *Computer Games Workshop IJCAI*, 2017.

[26] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Neural Information Processing Systems 2017*, 2017.

[27] J. Baxter, A. Tridgell, and L. Weaver, "TDLeaf($\lambda$): Combining temporal difference learning with game-tree search," *CoRR*, vol. cs.LG/9901001, 1999. [Online]. Available: http://arxiv.org/abs/cs.LG/9901001

[28] Y. B. Xavier Glorot, Antoine Bordes, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, vol. 15. PMLR, 2011, pp. 315–323.

[29] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 648–656.

[30] J. Pawlewicz, R. Hayward, P. Henderson, and B. Arneson, "Stronger virtual connections in hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 2, pp. 156–166, 2015.

[31] R. B. Hayward, Y. Bjrnsson, M. Johanson, M. Kan, N. Po, and J. van Rijswijck, "Solving 7 $\times$ 7 hex with domination, fill-in, and virtual connections," *Theoretical Computer Science*, vol. 349, pp. 123–139, 2005.

[32] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

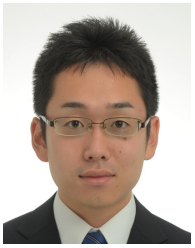
[33] B. Arneson, P. T. Henderson, and R. B. Hayward, "Benzene," http://benzene.sourceforge.net/, 2009-2012.

[34] K. Young, "benzene-vanilla," https://github.com/kenjyoung/benzene-vanilla, 2013.

[35] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug 1988.

[36] S.-C. Huang, B. Arneson, R. B. Hayward, M. Müller, and J. Pawlewicz, "Mohex2.0 : A pattern-based mcts hex player," *Computer and Games, Springer LNCS*, vol. 8427, pp. 60–71, 2014.

[37] C. Gao, R. B. Hayward, and M. Müller, "Move prediction using deep convolutional neural networks in hex," *IEEE Transactions on Games*, 2017.
[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2014.
[39] P. T. Henderson, "Playing and solving the game of hex," Ph.D. dissertation, the University of Alberta, 2010.
[40] R. Coulom, "Bayesian elo rating," https://www.remi-coulom.fr/Bayesian-Elo/, 2010.



**Kei Takada** Kei Takada received a MS degree in Graduate School of Information Science and Technology, Hokkaido University, Japan, in 2016. He is currently working toward the PhD degree at Graduate School of Information Science and Technology, Hokkaido University, Japan. He is a fellow of Japan Society for the Promotion of Science (JSPS) Research Fellowship for Young Scientist. His research interests include reinforcement learning, artificial player of the game, and human-machine interface.

**Hiroyuki Iizuka** Hiroyuki Iizuka received a PhD in multi-disciplinary sciences from the University of Tokyo, Japan, in 2004. Since 2005, he has been a research fellow of the Japan Society for the Promotion of Science. In 2005 and 2006, he was also a visiting research fellow at the Centre for Computational Neuroscience and Robotics at the University of Sussex. He was an assistant professor at the human information engineering laboratory, Osaka University (2008-2013). Currently, he is an associate professor at the autonomous systems engineering laboratory, Hokkaido University, Japan (2013-). His research interests include embodied cognition, complex adaptive systems, deep learning, swarm behavior, virtual reality, and the origin of life.

**Masahito Yamamoto** Masahito Yamamoto received a PhD in Graduate School of Engineering from Hokkaido University, Japan, in 1996. Since 1996, he has been a research fellow of the Japan Society for the Promotion of Science. He has been an assistant professor (1997-2000) and associate professor (2000-2012) in Hokkaido University. Currently, he is a professor at the autonomous systems engineering laboratory, Hokkaido University, Japan (2012-). His research interests include artificial life and intelligence, swarm intelligence, combinatorial optimization, and board game artificial intelligence (AI) programming.