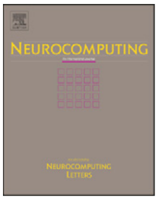
Contents lists available at SciVerse ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom



Self-teaching adaptive dynamic programming for Gomoku

Dongbin Zhao, Zhen Zhang*, Yujie Dai

State Key Laboratory of Intelligent Control and Management of Complex Systems, Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China

ARTICLE INFO

Keywords:
Gomoku
Reinforcement learning
Adaptive dynamic programming
Temporal difference learning
Neural network

ABSTRACT

In this paper adaptive dynamic programming (ADP) is applied to learn to play Gomoku. The critic network is used to evaluate board situations. The basic idea is to penalize the last move taken by the loser and reward the last move selected by the winner at the end of a game. The results show that the presented program is able to improve its performance by playing against itself and has approached the candidate level of a commercial Gomoku program called 5-star Gomoku. We also examined the influence of two methods for generating games: self-teaching and learning through watching two experts playing against each other and presented the comparison results and reasons.

1. Introduction

Gomoku is a board game that originated from one of the various kinds of black and white chess games in ancient China. Nowadays, it has become a popular game played in many places of the world. Gomoku is played between two players on a 15×15 square mesh. A sufficient amount of black or white pieces are offered to each player. The players in turns place one piece on the board. The winner is the first who forms a line of at least five adjacent pieces of his color, in horizontal, vertical or diagonal directions on the board. Such winning line is called five-in-a-row which is also referred to the name of the game Gomoku. If the board is filled completely with pieces and no one gets a five-in-a-row, this game is a draw. Gomoku has simple rules but high complexity as well, which makes it a suitable test-bed for a wide class of artificial intelligent algorithms.

One popular algorithm of playing Gomoku is game tree searching, which is often combined with a board evaluation function of leaf board situations. This method searches the partial game tree with a root node of the current board situation. If there is a leaf board situation with five-in-a-row of its colors, the move sequence leading to this situation is returned. Otherwise a board evaluation function is used to select an optimal position among all leaf board situations. According to William [1], a complete search to a depth of n moves need evaluation of $p!/(p-n)!$ board situations, where p is the number of legal positions a piece can occupy. Thus a complete game analysis is impossible. The history heuristic and alpha–beta search can be used to speed up game tree searching [2]. Another algorithm was proposed by Allis and Herik [3]. The core idea of this algorithm is to search winning threat sequences. Besides, Freisleben presented a neural network that was able to learn to play Gomoku [4]. This neural network is penalized or rewarded according to a special reinforcement learning algorithm, called comparison training [5].

Reinforcement learning (RL) is to learn how to map an action to a state to get the maximal reward from interacting with the environment. The merit of this algorithm lays on the fact that no explicit teacher is needed during the learning process so we can expect to get a more adaptive system using reinforcement learning algorithm. The difficulty of it is that the learning system must solve a temporal credit assignment problem, i.e., assign credit to each of the states and actions that result in the final outcome of an episode. However, reinforcement learning has attracted considerable interest for many years. Apart from its application to many practical problems, it has also been used as a model for explaining the action-selection mechanism of the brain [6]. Temporal difference (TD) learning algorithm is one way to assign temporal credit. This learning method utilizes the difference between two temporally successive predictions. One of the most successful applications of TD learning algorithm is TD-Gammon, a program that plays the game called backgammon. The latest version of TD-Gammon approaches the level of the world's best human players [7–9]. TD learning has been applied to Gomoku by Mo [10], but the results are not as exciting as in TD-Gammon. TDLeaf(λ), a variation on the TD(λ) algorithm, has been applied to a chess program called KnightCap [11]. It is thought that KnightCap's success is partially contributed by the appropriate choice of initial weights [12]. To select initial weights to speed up training, a Mini-max initialization method is proposed [13]. Another TD learning algorithm, kNN-TD(λ) method has been proposed and proved a robust high performance implementation of RL algorithm [14]. In TD learning, the action decision or the value function can be also described in continuous form, approximated by nonlinear functions such as neural networks. This is the key idea of Adaptive Dynamic Programming (ADP), first proposed in [15–17]. ADP has

* Corresponding author.
E-mail addresses: dongbin.zhao@ia.ac.cn (D. Zhao), zhangzdlut@gmail.com, zhen.zhang@ia.ac.cn (Z. Zhang), daiyujie07@mails.gucas.ac.cn (Y. Dai).

0925-2312/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.neucom.2011.05.032

been applied to many fields such as missile guidance [18] and pendulum robots [19].

We present a program ST-Gomoku employing ADP to learn by playing against itself to be a good player for Gomoku. ST-Gomoku utilizes a neural network to evaluate board situations and decide which move is supposed to be taken next. By playing against itself, the neural network is trained to learn the winning probability of any possible board situation. During the training process, one player should select the move that leads to the board situation with his maximal winning probability. On the contrary, the other player should choose the move that leads to the board situation with the minimal winning probability of his opponent. It will be shown that ST-Gomoku approaches the candidate level of a commercial Gomoku program called 5-star Gomoku. Using ST-Gomoku as an expert, we also test and compare two different methods for generating games used for training: (1) self-teaching and (2) learning through watching two experts playing against each other. We did the comparison under the inspiration of Wiering's research on backgammon [20], with the objective to find out which method combined with temporal difference learning can gain the best performance and the reasons behind it.

This paper is organized as follows. Section 2 gives a brief review of RL algorithm. In Section 3, we provide the topology of the evaluation neural network and how to train it. In Section 4, the experimental results indicating the performance of the obtained expert program and the comparison results between two different methods of generating games are presented and discussed. Section 5 concludes this paper and points out the direction for future research.

2. RL and ADP

RL uses cumulative reward $R(t)$ to evaluate the policy followed by an agent. It is defined as follows:

$$R(t)=r(t+1)+\gamma r(t+2)+\gamma^2 r(t+3)+\cdots=\sum_{(k=0)}^{T}\gamma^k r(t+k+1) \qquad (1)$$

where γ is the discount factor within [0, 1], T is the ending time of an episode and $r(t+1)$ is the reward received at time $t+1$. The cumulative reward is often called cost-to-go function, which is represented by $V(t)$ in ADP. The most basic structure of ADP is heuristic dynamic programming(HDP), whose structure is shown in Fig. 1.

The current system state $\mathbf{x}(t)$ is fed forward to the action network, which generates the control action $\mathbf{u}(t)$. The system model is used to predict the next step transition state $\mathbf{x}(t+1)$, which is fed forward to the utility function r to produce a reward $r(\mathbf{x}(t+1))$. The critic network is used to estimate the cost-to-go function V based on the system states only. Then the reward $r(\mathbf{x}(t+1))$, the estimation $V(t)$ and $V(t+1)$ are used to update the weights of the critic network to make the cost-to-go function V satisfy the Bellman equation. The difference between $V(t)$ and the desired objective $U_c(t)$ is used to train the action network to produce the optimal policy. Note that first, there can be no system model network in this structure and second, the critic network and the action network are not restricted to a particular structure.

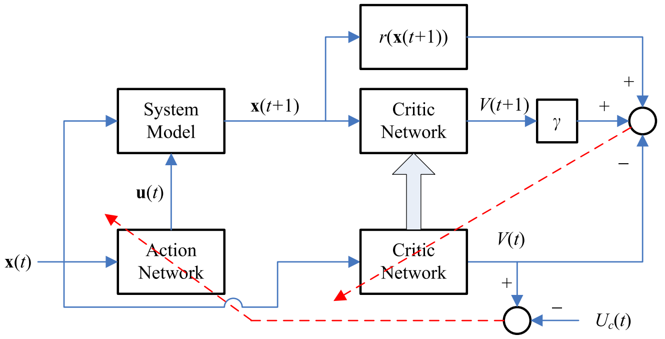


Fig. 1. The HDP structure.

3. Self-teaching ADP for Gomoku

In our self-teaching ADP for Gomoku, the critic network is a neural network which is used to evaluate board situations. The action network is not a neural network. It works together with the critic network to determine an action, which will be elucidated in Section 3.3. Hereby, the neural network means the critic network if there is no explicit explanation.

3.1. The state

To evaluate a board situation, the first important issue is how to describe it. Based on the research by Mo [10], we extend previous patterns to 20 patterns for each of the two players, that is, a total of 40 patterns to describe a board situation. Eight typical patterns are shown in Fig. 2. Each pattern is unique and consists of five or six adjacent positions in a line which are only occupied by the pieces of the same color. The number of each pattern in horizontal, vertical and diagonal lines can be identified to partially describe a board situation. Besides, whose turn to move is another important piece of information for describing a board situation. Finally, the factor of whether a player is in the offensive or in the defensive is added, considering the fact that in Gomoku the strategies are quite different between the offensive and defensive players. Therefore, patterns, turns and the offensive/defensive issue together constitute a state which is a generalized description for a board situation.

The patterns used in this paper are carefully selected. Though several patterns may not be accepted by a Gomoku master player, we still use them because on the one hand some standard patterns are complicated and recognition of them is time-consuming, on the other hand the patterns we use are simple and can be recognized with ease. We use patterns instead of raw information for three reasons. First, patterns reflect the essence and features of the game in a more readily comprehensible way than raw information does. Second, it is expected that raw information has a much broader state space than patterns have. Finally, we have attempted to use raw information to describe board situations. However, the results are not satisfying.

The turn, that is, whose move is it, is another important state variable for describing board situations. This issue is crucial for Gomoku in particular. For example, if both sides have already got a pattern e in Fig. 2, then clearly the winner is the one who is to move. We expect that the evaluation function for Gomoku is not as smooth as the one for backgammon and that two input nodes are not enough to reflect the importance of turn. Therefore, for each pattern we assign two input nodes to represent the turn. The offensive/defensive issue decides the opening of a game to a large extent. In some chess programs, an opening book is utilized to

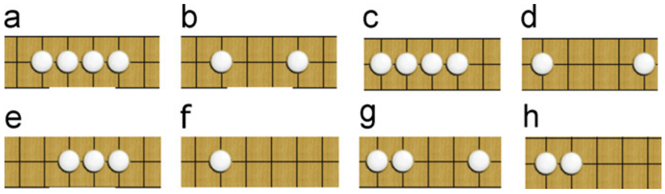


Fig. 2. Some patterns used in the presented program.

Please cite this article as: D. Zhao, et al., Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing (2011), doi:10.1016/j.neucom.2011.05.032

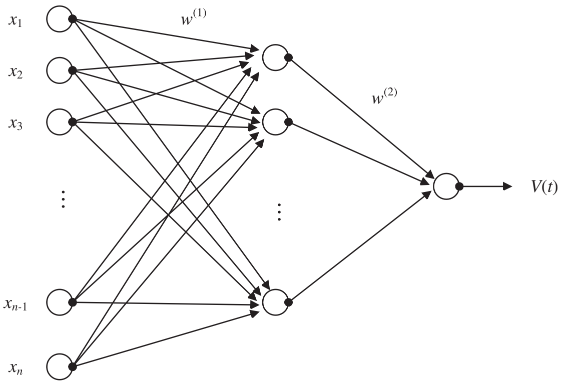


Fig. 3. Critic network topology for Gomoku.

Table 1
Coding method used in the presented program.

Value of n	Input 1	Input 2	Input 3	Input 4	Input 5
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	0
4	1	1	1	1	0
>4	1	1	1	1	$(n-4)/2$

generate the opening moves, while in our Gomoku program we simply use two input nodes to indicate which player is the first to move. All moves, including the opening moves, are generated according to the evaluation function.

3.2. The value function

A feed-forward three-layered fully connected neural network shown in Fig. 3 is adopted to evaluate board situations.

There are 274 nodes in the input layer, 100 nodes in the hidden layer and 1 node in the output layer. Five input nodes indicate the number of every pattern except for five-in-a-row. The coding method for these inputs is similar to the one used by Tesauro in TD-Gammon [21] and is shown in Table 1, where n denotes the number of a pattern.

The number of the special pattern, five-in-a-row, is represented by 1 input node. If this pattern shows up, then its input is 1, otherwise 0. Thus there are a total of $5\times 19\times 2+1\times 2=192$ input nodes representing the number of each pattern. Besides, two input nodes for every pattern indicate the turn to move. We assume that there are two players, player 1 and player 2. If a pattern shows up and it is player 1's turn to move, then the first input takes on the value 1 and the second one takes on the value 0. If a pattern shows up and it is player 2's turn to move, then the inputs for these two nodes are reverted. If a pattern does not appear, then both the corresponding two inputs are 0. Finally, two input nodes deal with the offensive/defensive issue. If player 1 moves first, then the first input is 1 and the second input is 0. If player 2 moves first, then the inputs for these two nodes are reverted. Thus the number of input nodes adds up to $192+80+2=274$. The output of the neural network means the winning probability of player 1 starting from a board situation, i.e., the value of the evaluation function at a state and it is of the form:

$$h_i(t)=\sum_{j=1}^{n} x_j(t)w_{ji}^{(1)}(t) \tag{2}$$

$$g_i(t)=\frac{1}{1+\exp^{-h_i(t)}} \tag{3}$$

$$p(t)=\sum_{i=1}^{m} w_i^{(2)}(t)g_i(t) \tag{4}$$

$$V(t)=\frac{1}{1+\exp^{-p(t)}} \tag{5}$$

where $w_{ji}^{(1)}$ the weight from the jth input node to the ith hidden node; x_j the jth input of the evaluation neural network; n total number of input nodes; h_i the ith hidden node input; g_i the ith hidden node output; $w_{\mathrm{i}}^{(2)}$ the weight from the ith hidden node to the output node; m total number of hidden nodes; p the input of the output node; $V(t)$ the output of the evaluation neural network.

The presented program plays against itself on a platform called Piskvorky [22]. Both the two sides use the same neural network to evaluate board situations. Thus player 1 chooses the move that leads to the state with the maximal output value obtained from the neural network. Player 2 selects the move that leads to the state with the minimal output value obtained by the same neural network. Since it is hard to give the reward r after a single move until the end of a game, the reward is set to 0 during the game. After a game, if player 1 wins, the final reward is 1, if he loses, the reward is 0, and if he draws against his opponent, the reward is 0.5.

To train the neural network, we define the prediction error as

$$e(t)=\alpha[r(t+1)+\gamma V(t+1)-V(t)] \tag{6}$$

and the objective error to be minimized is

$$E(t)=\tfrac{1}{2}e^2(t) \tag{7}$$

where α is the learning rate and γ is the discount factor.

The weight update rule for the neural network is based on the gradient decent method, described by

$$\Delta w(t)=-l(t)\frac{\partial E(t)}{\partial W(t)} \tag{8}$$

where l is the learning rate of the evaluation neural network. By applying the error back propagation mechanism, the weights are updated as follows:

$$\Delta w_i^{(2)}(t)=l_2(t)\left[-\frac{\partial E(t)}{\partial w_i^{(2)}(t)}\right] \tag{9}$$

$$\Delta w_i^{(2)}(t)=l_2(t)\left[-\frac{\partial E(t)}{\partial e(t)}\frac{\partial e(t)}{\partial V(t)}\frac{\partial V(t)}{\partial p(t)}\frac{\partial p(t)}{\partial w_i^{(2)}(t)}\right] \tag{10}$$

$$\Delta w_i^{(2)}(t)=l_2(t)\alpha e(t)\exp^{-p(t)}V^2(t)g_i(t). \tag{11}$$

$$\Delta w_{ji}^{(1)}(t)=l_1(t)\left[-\frac{\partial E(t)}{\partial w_{ji}^{(1)}(t)}\right] \tag{12}$$

$$\Delta w_{ji}^{(1)}(t)=l_1(t)\left[-\frac{\partial E(t)}{\partial e(t)}\frac{\partial e(t)}{\partial V(t)}\frac{\partial V(t)}{\partial p(t)}\frac{\partial p(t)}{\partial g_i(t)}\frac{\partial g_i(t)}{\partial h_i(t)}\frac{\partial h(t)}{\partial w_{ji}^{(1)}(t)}\right] \tag{13}$$

$$\Delta w_{ji}^{(1)}(t)=l_1(t)\alpha e(t)\exp^{-p(t)}V^2(t)w_i^{(2)}(t)g_i^2(t)\exp^{-h_i(t)}x_j(t). \tag{14}$$

3.3. The action

For Gomoku, a good move is often near a position which has been occupied. Based on this fact, we can reduce the action space by only considering the empty positions near the ones occupied. During the training process, when there are several alternative

Please cite this article as: D. Zhao, et al., Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing (2011), doi:10.1016/j.neucom.2011.05.032

actions which have equally high evaluation, we simply choose the one that is last found.

To cope with the exploration–exploitation dilemma, we explore the state space of Gomoku in two ways combined. The first way is to let player 2 randomly select his first move whether he is in the defensive or in the offensive. In the meanwhile, we let player 1 place his piece on the center of the board if he is in the offensive and select his first move randomly if he is in the defensive. The second way is to let both players select moves following ε-greedy policy, which is defined by

$$a(t)=\begin{cases}\arg\max\limits_{a} V(t+1) & \text{with probability } 1-\varepsilon\\ \text{random action} & \text{with probability } \varepsilon\end{cases} \tag{15}$$

As the number of games increases, the value of ε should decrease gradually. After a number of games, the first way of exploring should be eliminated.

Also, we have attempted to improve learning efficiency. First, the number of training times for one state transition is set to 50 rather than 1. Second, the program is forced to form a five-in-a-row if there is a line of five adjacent positions of which four positions are occupied, with one position empty. It should be noted that it is the only rule that explicitly teach the neural network how to play Gomoku. After a number of games, this rule can be eliminated.

4. Experimental results and discussion

The presented neural network, training method and action selection method have been implemented. Our goals are first, comparing five different architectures of neural networks and pick out the one which performs best when playing against a commercial program called 5-star Gomoku, and second, utilizing the best program as an expert to help examine and compare two different methods of generating games: self-teaching and watching.

4.1. Comparison between five different architectures of neural networks

*Case*1: *ST-Gomoku*. First we try to obtain an expert program by self-teaching. As the decisive reward is given at the end of a game, we set the discount factor γ to 1, indicating that future reward should be taken into account as strongly as possible. The learning rate α is set to 0.2, and both l_1 and l_2 are set to 0.05 as constant. The weights are initialized randomly as small values between -0.5 and 0.5. Under the condition of selecting moves by the first method of exploration mentioned in Section 3.3 and ε-greedy policy with $\varepsilon=0.1$, 10,000 games are played. Then the trained neural network is gained to play with another program called 5-star Gomoku. During the contest, unlike what we did in the training process, we let our program randomly select a move among positions which have equally maximal winning probability, with the purpose of decreasing the similarity among testing games. As a result, it beats the beginner level of 5-star Gomoku, with a score of winning 20 games out of 30. The presented program learns a few rules but still has severe problems in defense.

Based on weights obtained above, a second 10,000 games are played with the same settings as above. Then, four 10,000 games are played with $\varepsilon=0.1$, 0.08, 0.07, 0.05 respectively and without applying the first way of exploration. We notice that during the training process the evaluation for a win converges to value 1 and that for a fail converges to value 0. Then this trained neural network competes again with 5-star Gomoku. This time the neural network wins all the 30 games when it plays against the beginner level and wins 22 games out of 30 when it plays against the dilettante level. It wins 13 out of 30 games when it plays against the candidate level of 5-star Gomoku. The moves selected by the neural network are much cleverer than before, even though occasionally distinct bad moves might be chosen.

At last, the neural network is continuously trained for another 20,000 games with $\varepsilon=0.1$ and the results achieved are no better than those before. So far we have obtained a Gomoku program which we refer to as ST-Gomoku (self-teaching Gomoku).

Case 2: *SST1-Gomoku*. We have attempted to simplify the neural network by using less input nodes and less hidden nodes. First, the number of patterns is directly used as inputs, as opposed to the coding method employed in ST-Gomoku. Therefore, the number of inputs is reduced to 122. Second, the hidden nodes have been decreased to 60. We trained this neural network for 80,000 games in the same way as what we did for ST-Gomoku. We call this program SST1-Gomoku (the simplified version 1 of ST-Gomoku).

Case 3: *SST2-Gomoku*. We have also attempted to use only two input nodes to represent the turn. This program is referred to as SST2-Gomoku.

*Case*4: *SST3-Gomoku*. We eliminate all the input nodes which represent the turn and decrease the number of hidden nodes to 60. This program is referred to as SST3-Gomoku. After 30,000 training games with $\varepsilon=0.1$, the neural network was totally beaten by the beginner level of 5-star Gomoku with a score of 0:30. We also find that in the last dozens of training games, the evaluation for the endgame still fluctuates sharply, as opposed to Case 1 where the evaluation for a win is closely to value 1 and the evaluation for a fail is closely to value 0.

*Case*5: *RIST-Gomoku*. Finally, we attempt to design a neural network taking raw information as its inputs. Two input nodes indicate the turn. Two input nodes are used to represent the state of a position. If a position is empty, then the two corresponding inputs are set to 0. If it is occupied by player 1, then the first input is set to 1. If it is occupied by player 2, then the second input is set to 1. Thus for a 15×15 board, there are a total of $2\times225+2=452$ input nodes to represent board situations. The number of hidden nodes is still 100. We refer to this program as RIST-Gomoku (self-teaching Gomoku using raw information). We trained this network for 30,000 games with $\varepsilon=0.1$ and tested it. As a result, it lost all the 30 games playing against the beginner level of 5-star Gomoku.

SST1-Gomoku and SST2-Gomoku have also been played against 5-star Gomoku separately. The comparison results of the five different cases are shown in Table 2.

According to the experimental results, we choose ST-Gomoku as the expert program to help examine the influence of different methods of generating games.

4.2. Comparison between two methods for generating games

To compare the two different methods for generating training games—watching and self-teaching, we present the performance index and test conditions as follows: (1) we define a correct move as a move selected by ST-Gomoku. (2) The correctness is defined

Table 2
Comparison results of five different Gomoku programs.

Case	Input (Turn)	Hidden	Training	Beginner	Diletante	Candidate
Case 1	274 (80)	100	60,000	30:0	22:8	13:17
Case 2	122 (80)	60	80,000	30:0	15:15	14:16
Case 3	196 (2)	60	80,000	30:0	23:7	11:19
Case 4	194 (0)	60	30,000	0:30	–	–
Case 5	452 (2)	100	30,000	0:30	–	–

as the ratio of the cumulative number of correct moves to the cumulative number of total moves in 1000 games. (3) We only consider moves selected by evaluation function after two rounds of a game, that is, the first two moves of each side or random selected moves are not counted as the number of total moves in our experiment. (4) The topology of neural networks and the parameter settings except for initial neural network weights are the same as those for ST-Gomoku. (5) The initial neural network weights are the same for watching and self-teaching, yet they are different from the ones used in ST-Gomoku. These are for three reasons: (1) The opening moves may be good even if they are not selected by ST-Gomoku. (2) We expect that the higher the correctness is, the better performance a program gains. (3) We want to get general results independent of the initial weights. The results of comparison between watching and self-teaching are shown in Figs. 4–6.

From Fig. 4, we noticed that self-teaching performs better than watching. The similar result appears in Wiering's research on backgammon [18]. We agree Wiering's explanation that the learner has never participated in move selection. We also note that the performance progresses fastest at the beginning phase of the learning process and that the learning efficiency decreases as the training proceeds. This could be explained by the facts that: (1) the initial level of the learner is relative low and (2) since the reward is given at the end of a game, the most crucial patterns which directly decide the result of a game such as five-in-a-row and pattern a in Fig. 2 are the first to be evaluated accurately. Only if these decisive patterns are correctly evaluated, could the other patterns such as pattern b and pattern e in Fig. 2 that lead to these patterns be correctly evaluated.

In Fig. 5 we can see that although the methods for generating training games are different, in each case, the performances of player 1 and player 2 are almost the same during the 80,000 training games. The reason is that both the programs for player 1 and player 2 improve the same neural network.

If we take a close look at Fig. 6, we can see that the result for self-teaching is quite interesting. At first, a game does not last long. As the training proceeds, the number of total moves decreases rapidly until 3000 games have been played. Then it increases in a zigzag until 20,000 training games. The reason might be that the pattern five-in-a-row directly leads to the end of a game. Therefore, during the first 3000 games, the program of self-teaching learned how to win a game—how to correctly evaluate the pattern five-in-a-row but had not yet learned how to defense—how to correctly evaluate other important patterns such as pattern a and pattern e in Fig. 2, leading to the decrease of the number of total moves per game. After the program learned the correct evaluation of some important patterns, it can evanish the advantageous patterns of its opponent, leading to the increase of rounds per game. During the 21th 1000 games, there is a sharp increase. The program can play against itself for more rounds than it can do at initial. The reason can be that we eliminate the first way of exploring after 20,000 games. Thus the opening of a game is not as stochastic as before and become more equal for both sides than before, leading to more rounds in a game. After 60,000 games have been played, the number of total moves per game fluctuates around 23. There are two points needed to be clarified. First, the number of total moves is the same as defined in Section 4.2, so the number of actual rounds per game is a little higher. However, we can just ignore this factor by adding an offset. Second, in the scenario of watching, the moves are selected according to the static evaluation function of ST-Gomoku, so its number of total moves varied not very much.

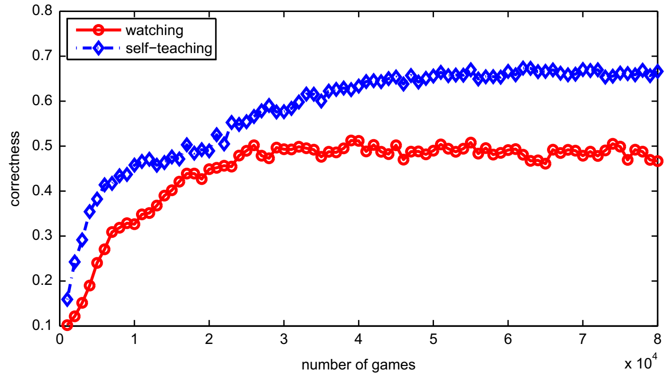


Fig. 4. Correctness for methods of watching and self-teaching for generating games. The correctness is the ratio of the number of correct moves to the number of total moves for both sides of the game.

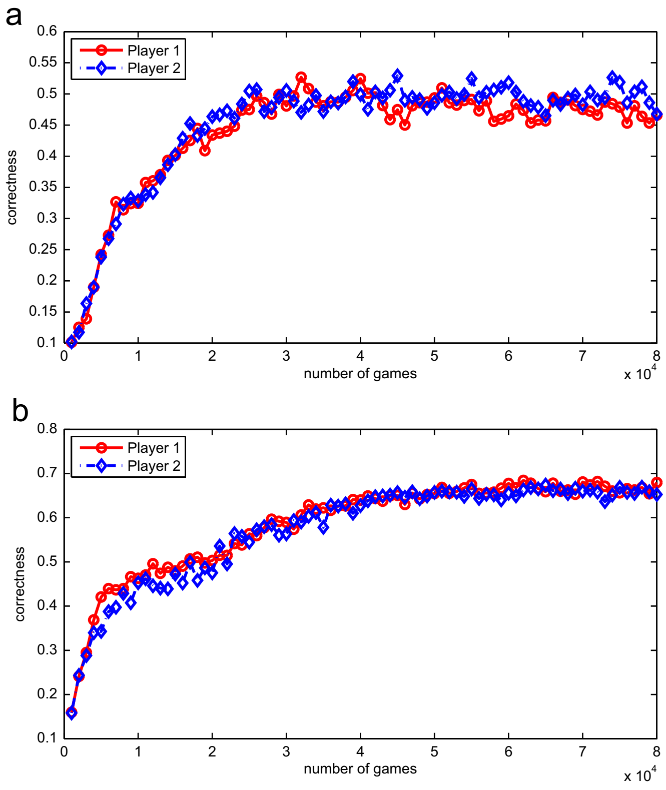


Fig. 5. Correctness for each side of the game. The correctness is the ratio of the number of correct moves to the number of the total moves for each player. (a) Watching and (b) self-teaching.

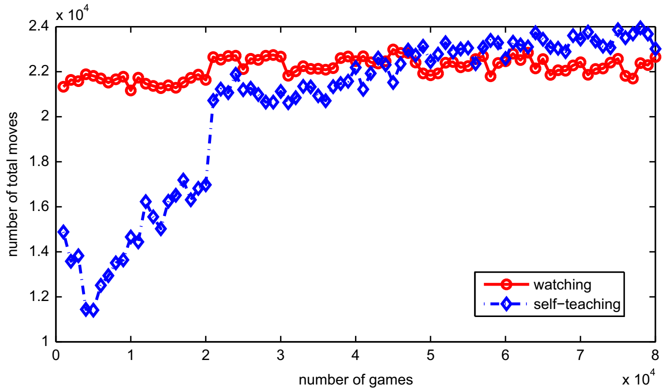


Fig. 6. The number of total moves using methods of watching and self-teaching.

Please cite this article as: D. Zhao, et al., Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing (2011), doi:10.1016/j.neucom.2011.05.032

We let the watching program and ST-Gomoku play against each other. As a result, ST-Gomoku won all 30 games.

4.3. Discussion

This paper utilizes ADP to learn to play Gomoku. The results for ST-Gomoku indicate that the patterns used here together with playing order can be used to describe a board situation well. The results also show that the way of exploring the state space of Gomoku is an important factor of improving the performance of the neural network. An important feature of backgammon is that there are dice to place to decide a deterministic action space, simplifying the decision process. Unlike backgammon, Gomoku uses ε-greedy policy to explore the action space. If ε is sufficiently small, the action space could not be fully explored. On the other hand, we may expect that too much stochastic moves would decrease the learning quality. Thus a good exploration scheme for such a deterministic game should explore the state space efficiently and limit the use of random move selection as well. The exploration method proposed in Section 2 has both the merits. The reason is that for Gomoku, different opening moves often lead to different remaining move sequences taken by each player. The occasional bad moves indicate that the neural network have not been fully trained to map the correct winning probability to every board situation. There is no further improvement on performance after 60,000 training games. This might be explained by the fact that the patterns used here can not completely describe a board situation, which limits the performance of ST-Gomoku. If more patterns are added to the neural network, better results might be achieved.

The turn contributes to the success of ST-Gomuku as a key factor, which can be seen from the comparison results between ST-Gomoku, SST2-Gomoku and SST3-Gomoku. The architecture of neural network used in ST-Gomuku can be simplified further while keeping a satisfying level. This could be explained by the fact that many crucial patterns such as pattern a, pattern c and pattern e appear at most once or twice in a game of Gomoku. Therefore, for those decisive patterns, there is no need to assign five input nodes to represent the number of them, two or three nodes for each pattern might be enough.

It is hard to train a neural network like RIST-Gomoku well within dozens of thousands of games and weather it can converge is still unknown. The raw information describes board situations completely, while it accompanies the tremendous state space that is much broader than that of a neural network which uses patterns to describe board situations. Another reason might be the associative connections between the input layer and the hidden layer. There may be another way to connect the two layers which makes more sense than full connections.

If people want to play a board game well, a good start might be observing how other players play this game. However, this is not enough. They also need practice. For it is only through practice can they learn their deficiencies which are not easily found by just observing. This might explain the relative low performance of the watching program. Besides, learning should be a process that from low level to high level. People prefer to start learning by playing against a player who has a little higher level than themselves. As they improve their skill, they would choose more challenging players as their opponents. Thus, it is expected that in the watching program if the learner start to learn by watching two players who gradually improve their skill, then the performance may be better. Although self-teaching is superior, it also has its own disadvantages. Its learning efficiency is not high, which can be seen from our experimental results. A supervisor together with self-teaching might be able to solve this problem.

5. Conclusions and future direction

In this paper a neural network that is able to learn to play Gomoku is provided. Using ADP, the presented program ST-Gomoku can learn and improve evaluation function by playing against itself. ST-Gomoku approaches the candidate level of 5-star Gomoku. The patterns used here are particular for Gomoku. However, the coding method and the way of exploration proposed here could be applied to a class of board games. We have also examined two different methods for generating games and presented the results and reasons. We have been carrying on our research on the effects of adding more patterns and changing the architecture of the neural network. Also, how to give reward before the end of a game is another area for future research.

Acknowledgments

This work was supported partly by National Natural Science Foundation of China (Nos. 60874043, 60921061 and 61034002). We especially acknowledge Yongsheng Su and Zhaohui Hu for valuable discussions. We also thank Haoxing Wang and Yuan Xu for assistance with the experiments.

References

[1] T.K. William, S. Pham, Experience-based learning experiments using Gomoku, in: Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, Charlottesville, Virginia, USA, vol. 2, October 13–16, 1991, pp. 1405–1410.

[2] J. Schaeffer, The history heuristic and alpha–beta search enhancements in practice, IEEE Trans. Pattern Anal. Mach. Intell. 11 (11) (1989) 1203–1212.

[3] L.V. Allis, H.J. Herik, M.P.H. Huntjens, Gomoku and Threat-Space Search, ⟨http://citeseer.ist.psu.edu/viewdoc/summary?⟩, 2010, doi:10.1.1.96.5836.

[4] B. Freisleben, A neural network that learns to play five-in-a-row, in: Second New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems, 1995, pp. 87–90.

[5] G. Teasauro, Connectionist learning of expert preferences by comparison training, in: D.S. Touretzky (Ed.), Advances in Neural Information Processing Systems, vol. 1, Morgan Kaufman, San Francisco, 1989, pp. 99–106.

[6] F. Ishida, T. Sasaki, Y. Sakaguchi, H. Shimai, Reinforcement-learning agents with different temperature parameters explain the variety of human action-selection behavior in a Markov decision process task, Neurocomputing 72 (2009) 1979–1984.

[7] G. Tesauro, Neurogammon: a neural-network backgammon program, in: Proceedings of International Joint Conference Neural Networks, San Diego, California, USA, June 17–21, 1990, pp. 33–40.

[8] G. Tesauro, Practical issues in temporal difference learning, Mach. Learn. 8 (1992) 257–277.

[9] G. Tesauro, TD-Gammon, A self-teaching backgammon program achieves master-level play, Neural Comput. 6 (1994) 215–219.

[10] J.M. Mo, Study and Practice on Machine Self-Learning of Game-Playing. Master Thesis, Guangxi Normal University, 2003.

[11] J. Baxter, A. Tridgell, L. Weaver, Learning to play chess using temporal differences, Mach. Learn. 40 (2000) 243–263.

[12] M. Jacek, Knowledge-free and learning-based methods in intelligent game playing, Stud. Comput. Intell. 276 (2010) 71–89.

[13] X.M. Zhang, Y.Q. Chen, N. Ansari, Y.Q. Shi, Mini-max initialization for function approximation, Neurocomputing 57 (2004) 389–409.

[14] J.A. Martin H, J. Lope, D. Maravall, Robust high performance reinforcement learning through weighted k-nearest neighbors, Neurocomputing 74 (2011) 1251–1259.

[15] A.G. Barto, R.S. Sutton, C.W. Anderson, Neuron like adaptive elements that can solve difficult learning control problems, IEEE Trans. Syst. Man Cybern. 13 (1983) 834–847.

[16] P.J. Werbos, A Menu of Designs for Reinforcement Learning Over Time, Neural Networks for Control, MIT Press, Cambridge, 1990.

[17] D.P. Bertsekas, J.N. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, Belmont, 1996.

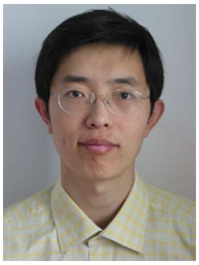
[18] J. Dalton, S.N. Balakrishnan, A neighboring optimal adaptive critic for missile guidance, Math. Comput. Model. 23 (1) (1996) 175–188.

[19] D.B. Zhao, J.Q. Yi, D.R. Liu, Particle swarm optimized adaptive dynamic programming, in: Proceedings of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, Honolulu, Hawaiian Islands, USA, April 1–5, 2007, pp. 32–37.

[20] M.A. Wiering, Self-play and using an expert to learn to play backgammon with temporal difference learning, J. Intell. Learn. Syst. Appl. 2 (2010) 57–68.

Please cite this article as: D. Zhao, et al., Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing (2011), doi:10.1016/j.neucom.2011.05.032

[21] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, The MIT Press, Cambridge, 1998.
[22] 〈http://gomocup.wz.cz/gomoku/download.php〉, 2010.



Dongbin Zhao (M'06, SM'10) received the B.S., M.S., Ph.D. degrees in August 1994, August 1996, and April 2000 respectively, in materials processing engineering from Harbin Institute of Technology, China. Dr. Zhao was a postdoctoral fellow in humanoid robot at the Department of Mechanical Engineering, Tsinghua University, China, from May 2000 to January 2002. He is currently an associate professor at the State Key Laboratory of Intelligent Control and Management of Complex Systems, Institute of Automation, Chinese Academy of Sciences, China. He has published one book and over 30 international journal papers. His current research interests lies in the area of computational intelligence, adaptive dynamic programming, robotics, intelligent transportation systems, and process simulation.

Zhen Zhang received the B.S. degree from China University of Petroleum, Dongying, China in 2006, and M.S. degree in Control Theory and Control Engineering in Dalian University of Technology in 2009, China. Since 2010 he has been a Ph.D. candidate at the State Key Laboratory of Intelligent Control and Management of Complex Systems, in Institute of Automation, Chinese Academy of Sciences. His main research interests include the application of reinforcement learning and neural networks to traffic signal control in urban areas.

Yujie Dai received the B.S. degree in Tianjin University, China in 2006. He has been a Ph.D. candidate at the State Key Laboratory of Intelligent Control and Management of Complex Systems, in Institute of Automation, Chinese Academy of Sciences since 2007. His current research interests include the application of computational intelligence and adaptive dynamic programming in intelligent transportation systems.

Please cite this article as: D. Zhao, et al., Self-teaching adaptive dynamic programming for Gomoku, Neurocomputing (2011), doi:10.1016/j.neucom.2011.05.032