Reinforcement Learning in Tic-Tac-Toe Game and Its Similar Variations

Group Members: Peng Ding and Tao Mao
Thayer School of Engineering at Dartmouth College

1. Introduction

Board games, which could be as simple as Tic-Tac-Toe, are where human wisdom has shined since a long time ago and are what human intelligence is trying to implement for machine intelligence for decades. Many of machine learning techniques have been explored, exploited, developed and argued extensively as researchers work on these canonical problems.

Perhaps, “minimax search” is seemingly the most successful one among many for it and its effective variations balance the searching efficiency and computational complexity, not to mention the glory of “Deep Blue” playing against Chess Master Kasparov. However, in common sense, minimax search is nothing but an enforced version of brute force search while it has a weak ability of automatically evaluating the board situation (or we call “state”). This makes it trivial by hand coding initially and also, sometimes, impossible to be applied.

Researchers had been intensively interested in finding a method of evaluation the board state in the last century and maybe still today. In early 1990’s, Sutton and Barto [1] systematically developed an unsupervised learning method-reinforcement learning; Watkins [2] proposed an important online implementation called Q-learning and proved its convergence, making the online technique work powerfully. If applied to games, reinforcement learning only needs the values of final states, which are easy to determine. For example, Tesauro utilized this method and succeeded in the solution of Gammon game [3].

Tic-tac-toe is traditionally a popular board game among kids: in its 3 by 3 board two persons alternately place one piece at a time; one wins when he or she has three pieces of his or her own in a row, whether horizontally, vertically, or diagonally. This work will employ reinforcement learning methods in its version of “afterstate” evaluation to implement simple board games such as Tic-Tac-Toe.

2. Methods: Reinforcement Learning

(1) Representation of the board state

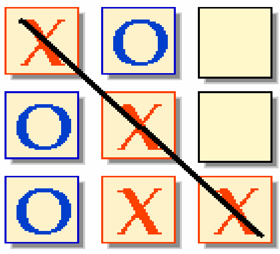


Figure 1 Winning situation for player "X" in Tic-tac-toe [4]

Multi-dimensional vectors are used to describe the state space of each situation. For example, as shown in Figure 1, we give it the following representing vector

$$s = [1,\ 2,\ 0,\ 2,\ 1,\ 0,\ 2,\ 1,\ 0]^{\mathrm{T}} \tag{1}$$

where 1 indicates player "X" places a piece in this location, 2 indicates player "O" places a piece in this location, and 0 indicates this is an empty location.

(2) Reinforcement learning for evaluation of afterstates
Reinforcement learning (RL) is an unsupervised machine learning technique, which "learns" from the interactive environment's rewards to approximate values of state-action pairs and maximize the long-term sum of rewards. It has four essential components: state set *S*, action set *A*, rewards from the environment *R*, and values for state-action pairs *V*.

A little different from the standard RL and specifically for board game applications, we combine state set *S* and action set *A* into a new "afterstate" set ***S***. The reasons are: (1) in board games, a state after a move is deterministic; (2) Different "prestate" and action may come to the same "afterstate", thus possibly holding redundancy in many state-action pairs.

We will use temporal-difference method, one of reinforcement learning techniques, to approximate state values by updating values of visited states after each training game.

$$V(s) \leftarrow V(s) + \alpha\left[V(s') - V(s)\right] \tag{2}$$

where *s* is the current state, *s*' is the next state, V(*s*) is a state value for state *s*, and α is the learning rate, varying within (0, 1].

The state value would converge if following conditions are met [2].

$$\lim_{T\to\infty}\sum_{t=0}^{T}\alpha_t = \infty \quad \text{and} \quad \lim_{T\to\infty}\sum_{t=0}^{T}\alpha_t^{\,2} < \infty \tag{3}$$

During learning phase, the policy selection is usually made based on the Boltzman’s distribution

$$\pi(s) = \Pr(s) = \frac{e^{V(s)/\tau}}{\sum_{i=1}^{n_s} e^{V(s_i)/\tau}} \tag{4}$$

where n_s is the number of afterstates, and τ is the “temperature”. High τ makes the probability distribution virtually uniform, while low τ makes the policy nearly a greedy selection. This property will result in robots making role decisions elegantly from “soft” to “hard” (from random to *Q*-value dependently selective).

And the final game policy π^* could be obtained by

$$\pi^* = \arg\max_{s_i} V(s_i) \tag{5}$$

(3) Training dataset
For simple games such as Tic-tac-toe, two compute agents will play against each other and learn game strategies from simulated games. This training method is called self-play, which has several advantages such as that an agent has general strategies rather than those associated with a fixed opponent [5]. Most of this project would stick to the generation of self-play training and the result shows its good performance though it might have a slow convergence problem in the early training stage.

3. Implementations

(1) Programming
The project is implemented in C++ by modifying a program posted online [6]. The original just implemented a basic function of two human playing alternatively but **no** AI algorithm at all in the program. We modified the code to make the game played by the computer for one step and by human for another and also embedded the reinforcement learning algorithm for AI core before the game starts.

(2) Two phases in the game framework
The algorithm's framework consists of learning phase and game-play phase. Below is the brief description of the algorithm's structure:

a. In each episode of the learning phase
1) Observe a current board state s;
2) Make a next move based on the distribution of all available V(s') of next moves;
3) Record s' in a sequence;
4) If the game finishes, it updates the values of the visited states in the sequence and starts over again; otherwise, go to 1).

b. The game-play phase makes a "greedy" decision based on the learned state values. Every time the computer is making next move,
1) Observe a current board state s;
2) Make a next move based on the distribution of all available V(s') of next moves;
3) Until the game is over and it starts over again; otherwise, go to 1).

It should be noted that these online afterward learning may not be included in Step 3) of the game-play phase as long as the game strategies are considered to be solid.

(3) Implementations of Tic-tac-toe and Four-in-a-row
We have implemented the games of Tic-Tac-Toe and Four-in-a-row using the proposed methods.

Basically, we need a matrix (3x3 or 4x4) to store information of the board state and use function `display()`to show it on the screen. We use function `move()` to make a next move. It is noted that the number of possible states for Tic-Tac-Toe is 19683 while the number for Four-in-a-row is 43046721. The latter is more complicated than the former considering its possible situations. Therefore, it needs more self-play games in the learning phase, as many as 100 millions, which takes several hours to train. However, the computer's performance in Four-in-a-row still can not reach that in Tic-Tac-Toe partially because the game is apt to get a draw unless either side makes a big mistake and the other side seizes it. So, we stored the trained state values in file named "qvalue.bin".

***Please note** the program files for Tic-Tac-Toe are included in the folder "**Tic-Tac-Toe**", and those for Four-in-a-row are included in the folder "**4inarow**". They should be run OK in Visual Studio **2008** environment. Since we used the absolute file address in the code for "4inarow", the program file of Four-in-a-row should be put directly un**der C drive be**fore it can run. So, please run C++ projects **directly under C drive**.

4. Results and Discussions

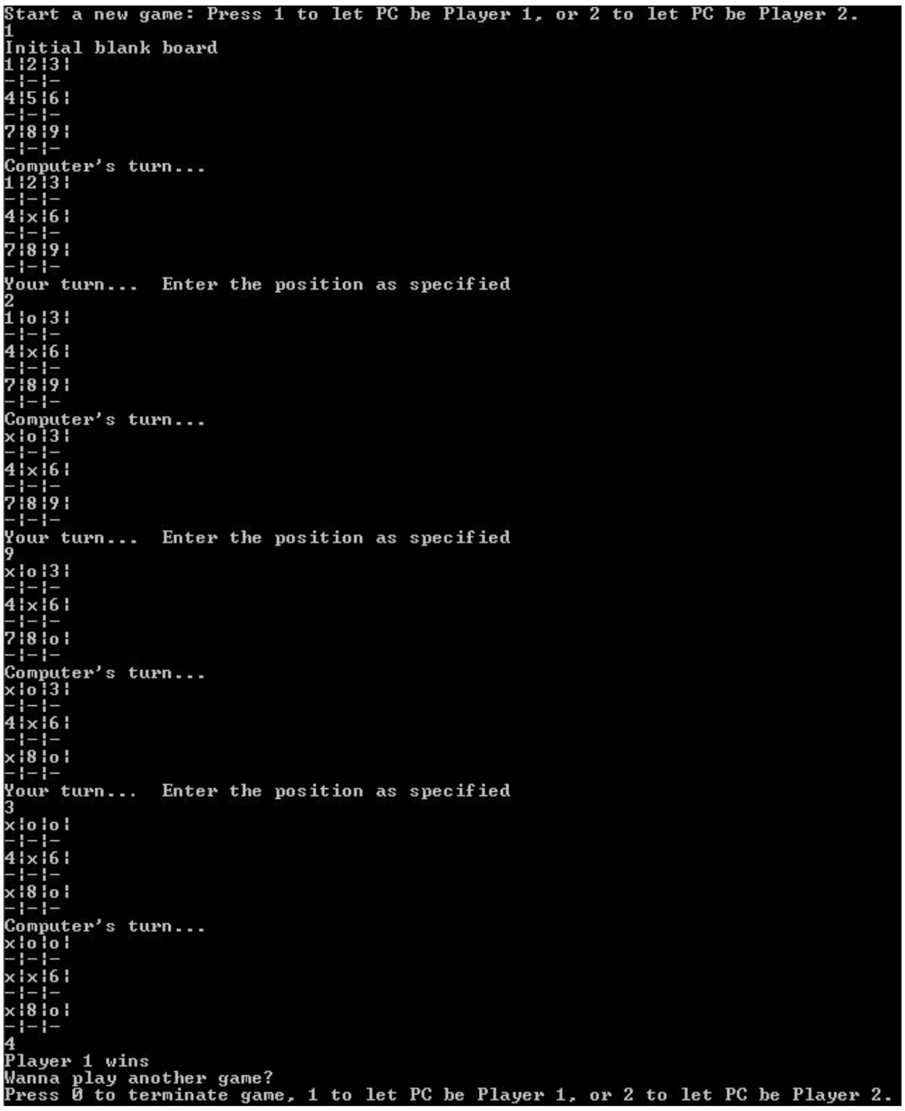
(1) Game play of Tic-Tac-Toe and Four-in-a-row
Typical game plays of Tic-Tac-Toe and Four-in-a-row are shown in Figure 2 and Figure 3. We can find that human can not beat the computer if the computer goes first.

(2) Numerical results and some discussions
We use the numerical results of Tic-Tac-Toe to provide an extensive analysis of the convergence for the adopted methods. We used 100,000 self-play games to train the computer to play Tic-Tac-Toe (approximately one minute) and use one billion self-play games to train the computer to play 4-in-a-row (several hours).

Player X represents the computer and Player O represents human. Player X goes first. We will show the convergence of "afterstate" value using the following example.

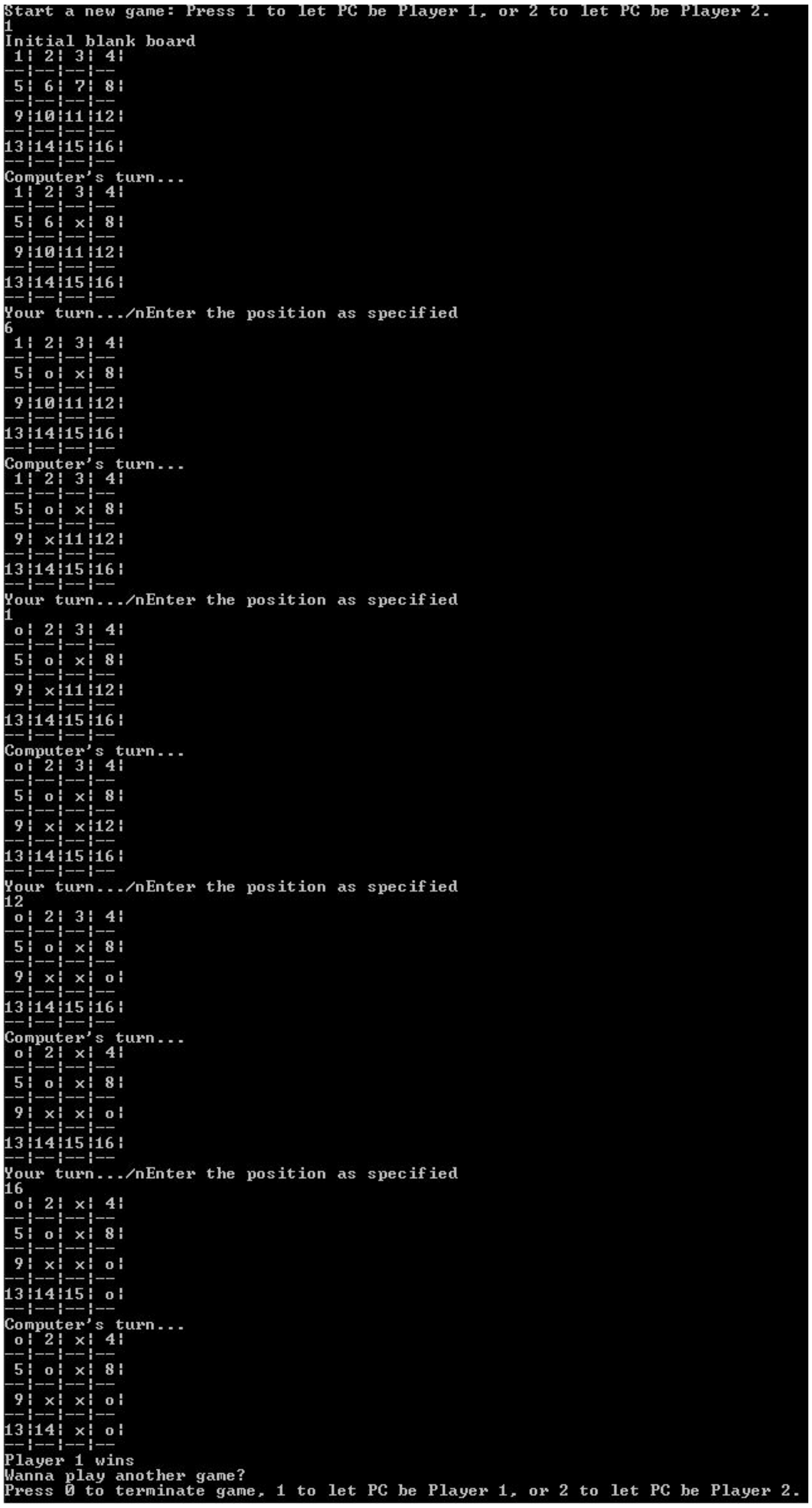
Which position is chosen to be the opening position is very crucial for Tic-Tac-Toe game. It is easily verified that the choice of central position results in "no-loss" guarantee. Therefore, the computer should be able to find the great value of opening in the center.

Figure 4 shows the convergence of the values of nine opening positions including the center.



```
Start a new game: Press 1 to let PC be Player 1, or 2 to let PC be Player 2.
1
Initial blank board
1¦2¦3¦
-¦-¦-
4¦5¦6¦
-¦-¦-
7¦8¦9¦
-¦-¦-
Computer's turn...
1¦2¦3¦
-¦-¦-
4¦x¦6¦
-¦-¦-
7¦8¦9¦
-¦-¦-
Your turn...  Enter the position as specified
2
1¦o¦3¦
-¦-¦-
4¦x¦6¦
-¦-¦-
7¦8¦9¦
-¦-¦-
Computer's turn...
x¦o¦3¦
-¦-¦-
4¦x¦6¦
-¦-¦-
7¦8¦9¦
-¦-¦-
Your turn...  Enter the position as specified
9
x¦o¦3¦
-¦-¦-
4¦x¦6¦
-¦-¦-
7¦8¦o¦
-¦-¦-
Computer's turn...
x¦o¦3¦
-¦-¦-
4¦x¦6¦
-¦-¦-
x¦8¦o¦
-¦-¦-
Your turn...  Enter the position as specified
3
x¦o¦o¦
-¦-¦-
4¦x¦6¦
-¦-¦-
x¦8¦o¦
-¦-¦-
Computer's turn...
x¦o¦o¦
-¦-¦-
x¦x¦6¦
-¦-¦-
x¦8¦o¦
-¦-¦-
4
Player 1 wins
Wanna play another game?
Press 0 to terminate game, 1 to let PC be Player 1, or 2 to let PC be Player 2.
```

Figure 2 Game play of Tic-Tac-Toe



```
Start a new game: Press 1 to let PC be Player 1, or 2 to let PC be Player 2.
1
Initial blank board
 1¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ 6¦ 7¦ 8¦
--¦--¦--¦--
 9¦10¦11¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Computer's turn...
 1¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ 6¦ x¦ 8¦
--¦--¦--¦--
 9¦10¦11¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Your turn.../nEnter the position as specified
6
 1¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦10¦11¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Computer's turn...
 1¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦11¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Your turn.../nEnter the position as specified
1
 o¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦11¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Computer's turn...
 o¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦ x¦12¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Your turn.../nEnter the position as specified
12
 o¦ 2¦ 3¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦ x¦ o¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Computer's turn...
 o¦ 2¦ x¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦ x¦ o¦
--¦--¦--¦--
13¦14¦15¦16¦
--¦--¦--¦--
Your turn.../nEnter the position as specified
16
 o¦ 2¦ x¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦ x¦ o¦
--¦--¦--¦--
13¦14¦15¦ o¦
--¦--¦--¦--
Computer's turn...
 o¦ 2¦ x¦ 4¦
--¦--¦--¦--
 5¦ o¦ x¦ 8¦
--¦--¦--¦--
 9¦ x¦ x¦ o¦
--¦--¦--¦--
13¦14¦ x¦ o¦
--¦--¦--¦--
Player 1 wins
Wanna play another game?
Press 0 to terminate game, 1 to let PC be Player 1, or 2 to let PC be Player 2.
```

Figure 3 Game play of Four-in-a-row

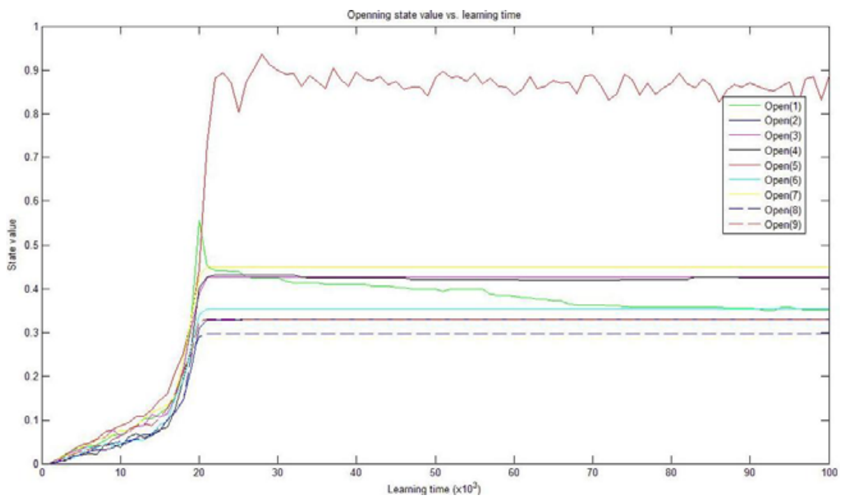


Figure 4 Convergence of nine opening positions: the central position is the best choice

Assume that the computer chooses the center for opening and the game has reached state S1, as shown in Figure 5. Now it is again player X's turn. Obviously, chance of winning the game is greater if player "X" takes the position of 1, 3, 7 or 9 (Figure 5).

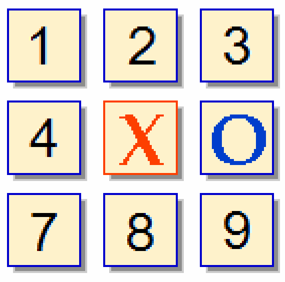


Figure 5 Board state S1

We assume that the computer has been trained enough to choose an action/position leading to a higher state value as, now, Player X chooses position 1 shown in Figure 6.

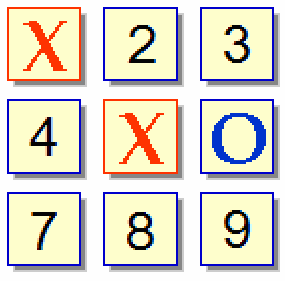


Figure 6 Board state S2 after Player "X" takes position 1

The plot below shows the afterstate value of state S1 over learning time (Figure 7). The value converges to approximately one. It explains that, after enough time of learning, Player "X" knows winning from state S2 is almost guaranteed.

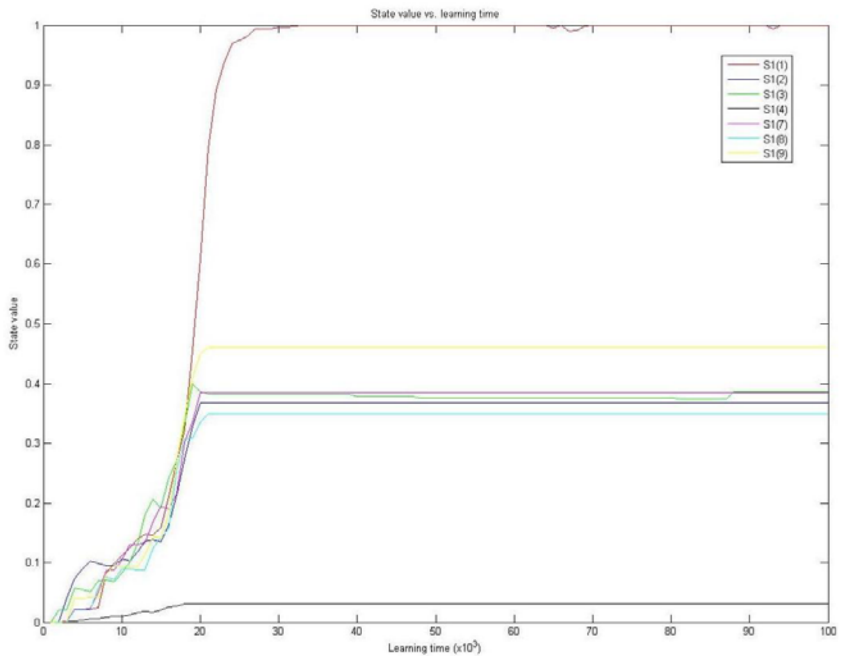


Figure 7 Afterstate values of state S1 vs. learning time

Then Player O chooses position 9 to avoid losing the game (Figure 8). In the next step, a well-learned player X (computer) would choose position 3 (Figure 9).

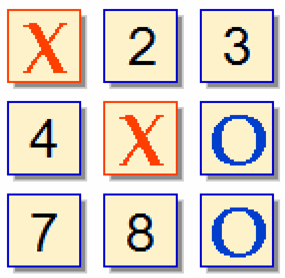


Figure 8 Board state S3 after Player "O" takes position 9

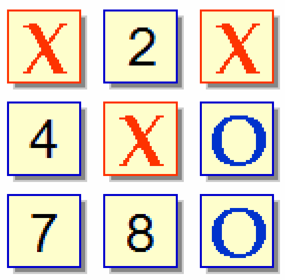


Figure 9 Board state S4

Up to this point, if it learns in sufficient time, Player X, the computer, will win the game no matter what position Player O takes. It is explained in more details that: if Player O chooses position 2, Player X will takes position 7 to win the game; if Player O chooses position 7, Player X will takes position 2 to win the game; if Player O chooses position 4 or position 8, Player "X" will win the game by either taking position 2 or position 7. Anyway, the computer finally wins.

Figure 10 shows how afterstate values of S3 converge over time. We notice that the true value of the crucial state does not quickly converge to the real value (i.e. one) due to the insufficient learning time. However, it will not prevent the algorithm make a right decision in game-play because the value of the crucial state stands out compared to those of others'.

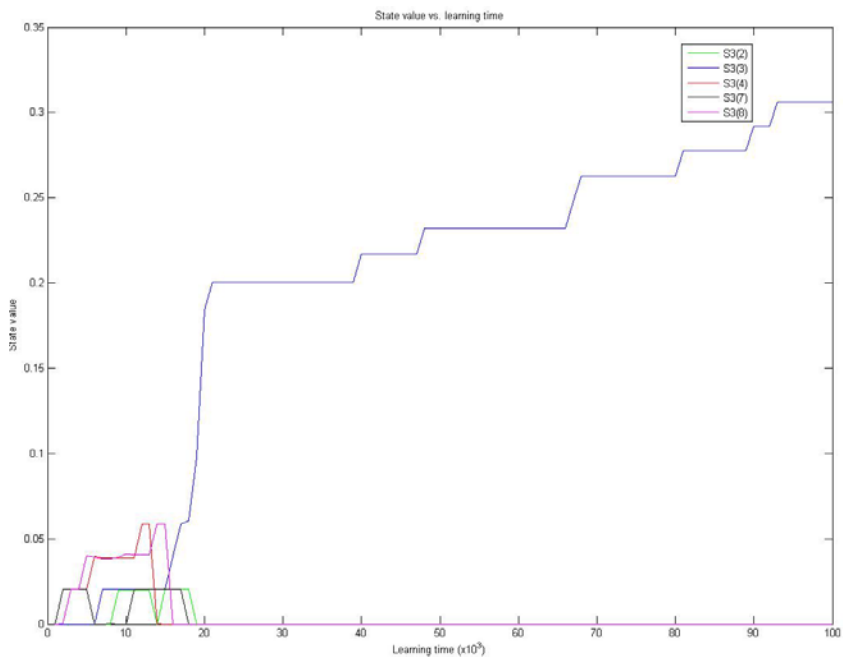


Figure 10 Afterstate values of state S3 vs. learning time

In conclusion, the computer has learned the correct game policies through the training methods in order to win, or, at least, not to lose.

5. Conclusions

The experiments of two games implemented in C++ demonstrate success of the applications of reinforcement learning to in the board games.

As we know, reinforcement leaning requires little prior knowledge except the evaluation of final states to generate rewards, which this method can forward as the evaluation information to earlier game stages. Since, with sufficient learning time, the algorithm would guarantee the convergence of state values the computer can use these values to determine its moves in the game.

Self-play training strategy makes the machine learning method look magic. It means that the computer can learn “knowledge” from the games between the computer and a rival as identical as itself. The more it trains itself, the more knowledge it gets; the more it trains itself, the more sophisticated its game policies are.

However, the significant issue of RL is the “curse of dimensionality”: as the board enlarges and the number of the states also increases, it needs a lot of self-play training. One possible solution would be to use real data of game between human, with which the game policies are trained more on purpose rather than based on many random moves.

Anyway, a well-trained computer via the proposed method of reinforcement learning in this case can play the games competently.

REFERENCES

[1] R. Sutton and A. Barto. Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA: pp. 10-15, 156. 1998.
[2] C. Watkins and P. Dayan. Q-learning. Machine Learning, 8(3): 279-292, 1992.
[3] G. Tesauro. Temporal difference learning and TD-Gammon. Communications of ACM, 38(3): pp. 58-68, 1995.
[4] Figure source: Boulter.com/ttt/
[5] I. Ghory. Reinforcement learning in board games, Technique report, Department of Computer Science, University of Bristol. May 2004.
[6] Original C++ code of Tic-Tac-Toe: http://www.dreamincode.net/code/snippet784.htm