

# Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning

Marco A. Wiering\*

Department of Artificial Intelligence, University of Groningen, Groningen, Netherlands.  
Email: m.a.wiering@rug.nl

Received October 22<sup>nd</sup>, 2009; revised January 10<sup>th</sup>, 2010; accepted January 30<sup>th</sup>, 2010.

## ABSTRACT

*A promising approach to learn to play board games is to use reinforcement learning algorithms that can learn a game position evaluation function. In this paper we examine and compare three different methods for generating training games: 1) Learning by self-play, 2) Learning by playing against an expert program, and 3) Learning from viewing experts play against each other. Although the third possibility generates high-quality games from the start compared to initial random games generated by self-play, the drawback is that the learning program is never allowed to test moves which it prefers. Since our expert program uses a similar evaluation function as the learning program, we also examine whether it is helpful to learn directly from the board evaluations given by the expert. We compared these methods using temporal difference methods with neural networks to learn the game of backgammon.*

**Keywords:** Board Games, Reinforcement Learning,  $TD(\lambda)$ , Self-Play, Learning From Demonstration

## 1. Introduction

The success of the backgammon learning program TD-Gammon of Tesauro (1992, 1995) was probably the greatest demonstration of the impressive ability of machine learning techniques to learn to play games. TD-Gammon used reinforcement learning [1,2] techniques, in particular temporal difference (TD) learning [2,3], for learning a backgammon evaluation function from training games generated by letting the program play against itself. This has led to a large increase of interest in such machine learning methods for evolving game playing computer programs from a randomly initialized program (*i.e.*, initially there is no a priori knowledge of the game evaluation function, except for a human extraction of relevant input features). Samuel (1959, 1967) pioneered research in the use of machine learning approaches in his work on learning a checkers program. In his work he already proposed an early version of temporal difference learning for learning an evaluation function.

For learning to play games, value function based reinforcement learning (or simply reinforcement learning) or evolutionary algorithms are often used. Evolutionary algorithms (EAs) have been used for learning to play backgammon [4], checkers [5], and Othello [6] and were quite successful. Reinforcement learning has been applied to learn a variety of games, including backgammon [7,8], chess [9,10], checkers [11,12,13], and Go [14].

Other machine learning approaches learn an opening book, rules for classifying or playing the endgame, or use comparison training to mimic the moves selected by human experts. We will not focus on these latter approaches and refer to [15] for an excellent survey of machine learning techniques applied to the field of game-playing.

EAs and reinforcement learning (RL) methods concentrate on evolving or learning an evaluation function for a game position and after learning choose positions that have the largest utility or value. By mapping inputs describing a position to an evaluation of that position or input, the game program can choose a move using some kind of look-ahead planning. For the evaluation function many function approximators can be used, but commonly weighted symbolic rules (a kind of linear network), or a multi-layer perceptron that can automatically learn non-linear functions of the input is used.

A difference between EAs and reinforcement learning algorithms is that the latter usually have the goal to learn the exact value function based on the long term reward (*e.g.*, a win gives 1 point, a loss  $-1$ , and a draw 0), whereas EAs directly search for a policy which plays well without learning or evolving a good approximation of the result of a game. Learning an evaluation function with reinforcement learning has some advantages such as better fine-tuning of the evaluation function once it is

quite good and the possibility to learn from single moves without playing an entire game. Finally, the evaluation function allows feedback to a player and can in combination with multiple outputs for different outcomes also be used for making the game-playing program play more or less aggressive.

In this paper we study the class of reinforcement learning methods named temporal difference (TD) methods. Temporal difference learning [3,7] uses the difference between two successive positions for back-propagating the evaluations of the successive positions to the current position. Since this is done for all positions occurring in a game, the outcome of a game is incorporated in the evaluation function of all positions, and hopefully the evaluation functions improves after each game. Unfortunately there is no convergence proof that current RL methods combined with non-linear function approximators such as feed-forward neural networks will find or converge to an optimal value function.

For learning a game evaluation function for mapping positions to moves (which is done by the agent), there are the following three possibilities for obtaining experiences or training examples; 1) Learning from games played by the agent against itself (learning by self-play), 2) Learning by playing against a (good) opponent, 3) Learning from observing other (strong) players play games against each other. The third possibility might be done by letting a strong program play against itself and let a learner program learn the game evaluation function from observing these games or from database games played by human experts.

**Research Questions.** In this paper we compare different methods for acquiring and learning from training examples. We pose ourselves the following research questions:

1) Which method combined with temporal difference learning results in the best performance after a fixed number of games? Is observing an expert player, playing against an expert, or self-play the best method?

2) When the learning program immediately receives accurate evaluations of encountered board positions, will it then learn faster than when it uses its initially randomized function approximator and TD-learning to get the board evaluations?

3) Is a function approximator with more trainable parameters more efficient for learning to play the game of backgammon than a smaller representation?

4) Which value for  $\lambda$  in TD ( $\lambda$ ) works best for obtaining the best performance after a fixed number of games?

**Outline.** This paper first describes game playing programs in section 2. Section 3 describes reinforcement learning algorithms. Then section 4 presents experimental results with learning the game of backgammon for which the above mentioned three possible methods for

generating training games are compared. Section 5 concludes this paper.

## 2. Game Playing Programs

Game playing is an interesting control problem often consisting of a huge number of states, and therefore has inspired research in artificial intelligence for a long time. In this paper we deal with two person, zero-sum, alternative move games such as backgammon, Othello, draughts, Go, and chess. Furthermore, we assume that there is no hidden state such as in most card games. Therefore our considered board games consist of:

- 1) A set of possible board positions.
- 2) A set of legal moves in a position.
- 3) Rules for carrying out moves.
- 4) Rules for deciding upon termination and the result of a game.

A game playing program consists of a move generator, a look-ahead algorithm, and an evaluation function. The move generator just generates all legal moves, possibly in some specific order (taking into account some priority). The look-ahead algorithm deals with inaccurate evaluation functions. If the evaluation function would be completely accurate, look-ahead would only need to examine board positions resulting from each legal move. For most games an accurate evaluation function is very hard to make, however. Therefore, by looking ahead many moves, positions much closer to the end of a game can be examined and the difference in evaluations of the resulting positions is larger and therefore the moves can be more easily compared. A well known method for looking ahead in games is the Minimax algorithm, however faster algorithms such as alpha-beta pruning, Negascout, or principal variation search [16,17] are usually used for good game playing programs.

If we examine the success of current game playing programs, such as Deep Blue which won against Kasparov in 1997 [18], then it relies heavily on the use of very fast computers and look-ahead algorithms. Deep Blue can compute the evaluation of about 1 million positions in a second, much more than a human being who examines less than 100 positions in a second. Also draughts playing programs currently place emphasis on look-ahead algorithms for comparing a large number of positions. Expert backgammon playing programs only use 3-ply look-ahead, however, and focus therefore much more on the evaluation function.

Board games can have a stochastic element such as backgammon. In backgammon dice are rolled to determine the possible moves. Although the dice are rolled before the move is made, and therefore for a one-step look-ahead the dice are no computational problem, this makes the branching factor for computing possible positions after two or more moves much larger (since then look-ahead needs to take into account the 21 outcomes of

the two dice). This is the reason that looking ahead many moves in stochastic games is infeasible for human experts or computers. For this Monte Carlo simulations [19] can still be helpful for evaluating a position, but due to the stochasticity of these games, many games have to be simulated.

On the other hand, we argue that looking ahead is not very necessary due to the stochastic element. Since the evaluation function is determined by dice, the evaluation function will become smoother since a position's value is the average evaluation of positions resulting from all dice rolls. In fact, in backgammon it often does not matter too much whether some single stone or field occupied by 2 or more stones are shifted one place or not. This can be again explained by the dice rolls, since different dice in similar positions can result in a large number of equal subsequent positions. Looking ahead multiple moves for backgammon may be helpful since it combines approximate evaluations of many positions, but the variance may be larger. A search of 3-ply is commonly used by the best backgammon playing programs [7,8].

This is different with e.g. chess or draughts, since for these games (long) tactical sequences of moves can be computed which let a player win immediately. Therefore, the evaluations of many positions later vary significantly and are more easily compared. Furthermore, for chess or draughts moving a piece one position can make the difference between a winning and losing position. Therefore the evaluation function is much less smooth (evaluations of close positions can be very different) and harder to learn. We think that the success of learning to play backgammon [8] relies on this smoothness of the evaluation function. It is well known that learning smooth functions requires less parameter for a machine learning algorithm and therefore faster search for a good solution and better generalization.

In the next section we will explain how we can use TD methods for learning to play games. After that the results of using TD learning for learning the game of Backgammon using different strategies for obtaining training examples will be presented.

### 3. Reinforcement Learning

Reinforcement learning algorithms are able to let an agent learn from its experiences generated by its interaction with an environment. We assume an underlying Markov decision process (MDP) which does not have to be known to the agent. A finite MDP is defined as; 1) The state-space  $S = \{s^1, s^2, \dots, s^n\}$ , where  $s_t \in S$  denotes the state of the system at time  $t$ ; 2) A set of actions available to the agent in each state  $A(s)$ , where  $a_t \in A(s_t)$  denotes the action executed by the agent at time  $t$ ; 3) A transition function  $P(s, a, s')$  mapping state action pairs  $s, a$  to a probability distribution of successor states  $s'$ ; 4) A

reward function  $R(s, a, s')$  which denotes the average reward obtained when the agent makes a transition from state  $s$  to state  $s'$  using action  $a$ , where  $r_t$  denotes the (possibly stochastic) reward obtained at time  $t$ ; 5) A discount factor  $0 \leq \gamma \leq 1$  which discounts later rewards compared to immediate rewards.

#### 3.1 Value Functions and Dynamic Programming

In optimal control or reinforcement learning, we are interested in computing or learning an optimal policy for mapping states to actions. We denote an optimal deterministic policy as  $\pi^*(s) \rightarrow a^*|s$ . It is well known that for each MDP, one or more optimal deterministic policies exist. An optimal policy is defined as a policy that receives the highest possible cumulative discounted rewards in its future from all states.

In order to learn an optimal policy, value-function based reinforcement learning [1,2,3] uses value functions to summarize the results of experiences generated by the agent in the past. We denote the value of a state  $V^\pi(s)$  as the expected cumulative discounted future reward when the agent starts in state  $s$  and follows a particular policy  $\pi$ :

$$V^\pi(s) = E \left( \sum_{i=0}^{\infty} \gamma^i r_i \mid s_0 = s, \pi \right)$$

The optimal policy is the one which has the largest state-value in all states. It is also well-known that there exists a recursive equation known as the Bellman optimality equation [20] which relates a state value of the optimal value function to other optimal state values which can be reached from that state using a single local transition:

$$V^*(s) = \sum_{s'} P(s, \pi^*(s), s') (R(s, \pi^*(s), s') + \gamma V^*(s'))$$

Value iteration can be used for computing the optimal V-function. For this we repeat the following update many times for all states:

$$V^{k+1}(s) = \max_a \sum_{s'} P(s, a, s') (R(s, a, s') + \gamma V^k(s'))$$

The agent can then select optimal actions using:

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s'} P(s, a, s') (R(s, a, s') + \gamma V^*(s'))$$

#### 3.2 Reinforcement Learning

Although dynamic programming algorithms can be efficiently used for computing optimal solutions for particular MDPs, they have some problems for more practical applicability; 1) The MDP should be known a-priori; 2) For large state-spaces the computational time would become very large; 3) They cannot be directly used in continuous state-action spaces.

Reinforcement learning algorithms can cope with these problems; first of all the MDP does not need to be known a-priori, all that is required is that the agent is allowed to interact with an environment which can be modeled as an MDP; secondly, for large or continuous

state-spaces, an RL algorithm can be combined with a function approximator for learning the value function. When combined with a function approximator, the agent does not have to compute state-action values for all possible states, but can concentrate itself on parts of the state-space where the best policies lead into.

There are a number of reinforcement learning algorithms, the first one known as temporal difference learning or TD(0) [3] computes an update of the state value function after making a transition from state  $s_t$  to state  $s_{t+1}$  and receiving a reward of  $r_t$  on this transition by using the temporal difference learning rule:

$$V(s_t) = V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t))$$

where  $0 < \alpha \leq 1$  is the learning rate (which is treated here as a constant, but should decay over time for convergence proofs). Although it does not compute action-value functions, it can be used to learn the value function of a fixed policy (policy-evaluation). Furthermore, if combined with a model of the environment, the agent can use a learned state value function to select actions:

$$\pi(s) = \operatorname{argmax}_a \sum_{s'} P(s, a, s') (R(s, a, s') + \gamma V(s'))$$

It is possible to learn the V-function of a changing policy that selects greedy actions according to the value function. This still requires the use of a transition function, but can be used effectively for e.g. learning to play games [7,8].

There exists a whole family of temporal difference learning algorithms known as TD( $\lambda$ )-algorithms [3] which are parameterized by the value  $\lambda$  which makes the agent look further in the future for updating its value function. It has been proved [21] that this complete family of algorithms converges under certain conditions to the same optimal state value function with probability 1 if tabular representations are used. The TD( $\lambda$ )-algorithm works as follows. First we define the TD(0)-error of  $V(s_t)$  as:

$$\delta_t = (r_t + \gamma V(s_{t+1}) - V(s_t))$$

TD( $\lambda$ ) uses a factor  $\lambda \in [0, 1]$  to discount TD-errors of future time steps:

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t^\lambda$$

where the TD( $\lambda$ )-error  $\delta_t^\lambda$  is defined as

$$\delta_t^\lambda = \sum_{i=0}^{\infty} (\gamma \lambda)^i \delta_{t+i}$$

**Eligibility traces.** The updates above cannot be made as long as TD errors of future time steps are not known. We can compute them incrementally, however, by using eligibility traces [3,22]. For this we use the update rule:

$$V(s) = V(s) + \alpha \delta_t e_t(s)$$

for all states, where  $e_t(s)$  is initially zero for all states and updated after every step by:

$$e_t(s) = \gamma \lambda e_{t-1}(s) + \eta_t(s)$$

where  $\eta_t(s)$  is the indicator function which returns 1 if state  $s$  occurred at time  $t$ , and 0 otherwise. A faster algorithm to compute exact updates is described in [23]. The value of  $\lambda$  determines how much the updates are influenced by events that occurred much later in time. The extremes are TD(0) and TD(1) where (online) TD(1) makes the same updates as Monte Carlo sampling. Although Monte Carlo sampling techniques that only learn from the final result of a game do not suffer from biased estimates, the variance in updates is large and that leads to slow convergence. A good value for  $\lambda$  depends on the length of an epoch and varies between applications, although often a value between 0.6 and 0.9 works best.

### 3.3 Reinforcement Learning with Neural Networks

To learn value functions for problems with many state variables, there is the curse of dimensionality; the number of states increases exponentially with the number of state variables, so that a tabular representation would quickly become infeasible in terms of storage space and computational time. Also when we have continuous states, a tabular representation requires a good discretization which has to be done a-priori using knowledge of the problem, and a fine-grained discretization will also quickly lead to a large number of states. Therefore, instead of using tabular representations it is more appropriate to use function approximators to deal with large or continuous state spaces.

There are many function approximators available such as neural networks, self-organizing maps, locally weighted learning, and support vector machines. When we want to combine a function approximator with reinforcement learning, we want it to learn fast and online after each experience, and be able to represent continuous functions. Appropriate function approximators combined with reinforcement learning are therefore feed-forward neural networks [24].

In this paper we only consider fully-connected feed-forward neural networks with a single hidden layer. The architecture consist of one input layer with input units (when we refer to a unit, we also mean its activation):  $I_1, \dots, I_{|I|}$ , where  $|I|$  is the number of input units, one hidden layer  $H$  with hidden units:  $H_1, \dots, H_{|H|}$ , and one output layer with output units:  $O_1, \dots, O_{|O|}$ . The network has weights:  $w_{ih}$  for all input units  $I_i$  to hidden units  $H_h$ , and weights:  $w_{ho}$  for all hidden  $H_h$  to output units  $O_o$ . Each hidden unit and output unit has a bias  $b_h$  or  $b_o$  with a constant activation of 1. The hidden units most often use sigmoid activation functions, whereas the output units use linear activation functions.

**Forward propagation.** Given the values of all input units, we can compute the values for all output units with

forward propagation. The forward propagation algorithm looks as follows:

1) Clamp the input vector  $I$  by perceiving the environment.

2) Compute the values for all hidden units  $H_h \in H$  as follows:

$H_h = \sigma(\sum_i w_{ih} I_i + b_h)$ , where  $\sigma(x)$  is the Sigmoid function:  $\sigma(x) = 1/(1+e^{-x})$ .

3) Compute the values for all output units  $O_o = \sum_h w_{ho} H_h + b_o$ .

**Backpropagation.** For training the system we can use the back-propagation algorithm [25]. The learning goal is to learn a mapping from the inputs to the desired outputs  $D_o$  for which we update the weights after each example. For this we use backpropagation to minimize the squared error measure:

$$E = \frac{1}{2} \sum_o (D_o - O_o)^2$$

To minimize this error function, we update the weights and biases in the network using gradient descent steps with learning rate  $\alpha$ . We first compute the delta values of the output units (for a linear activation function):

$$\delta_o(o) = (D_o - O_o)$$

Then we compute the delta values of all hidden units (for a sigmoid activation function):

$$\delta_h(h) = \sum_o \delta_o(o) w_{ho} H_h (1 - H_h)$$

Then we change all hidden-output weights and output bias values:

$$w_{ho} = w_{ho} + \alpha \delta_o(o) H_h; b_o = b_o + \alpha \delta_o(o)$$

And finally we change all input-hidden weights and hidden bias values:

$$w_{ih} = w_{ih} + \alpha \delta_h(h) I_i; b_h = b_h + \alpha \delta_h(h)$$

**Offline TD-methods.** All we need is a desired output and then backpropagation can be used to compute weight updates to minimize the error-function on every different example. To get the desired output, we can simply use offline temporal difference learning [26] which waits until an epoch has ended and then computes desired values for the different time-steps. For learning to play games this is useful, since learning from the first moves will not immediately help to play the rest of the game better. In this paper we used the offline TD( $\lambda$ ) method which provides the desired values for each board position, taking into account the result of a game and the prediction of the result by the next state. The final position at time-step  $T$  is scored with the result  $r_T$  of the game, *i.e.* a win for white ( $= 1$ ), a win for black ( $= -1$ ) or a draw ( $= 0$ ).

$$V'(s_T) = r_T \quad (1)$$

The desired values of the other positions are given by the following function:

$$V'(s_t) = \gamma V(s_{t+1}) + r_t + \lambda \gamma (V'(s_{t+1}) - V(s_{t+1}))$$

After this, we use  $V'(s_t)$  as the desired value of state  $s_t$  and use back-propagation to update all weights. In Backgammon, we used a minimax TD-rule for learning the game evaluation function. Instead of using an input that indicates which player is allowed to move, we always reverted the position so that white was to move. In this case, evaluations of successive positions are related by  $V(s_t) = -V(s_{t+1})$ . Without immediate reward and a discount factor of 1, the minimax TD-update rule becomes:

$$V'(s_t) = -V(s_{t+1}) + \lambda (V(s_{t+1}) - V'(s_{t+1}))$$

## 4. Experiments with Backgammon

Tesauro's TD-Gammon program learned after about 1,000,000 games to play at human world class level, but already after 300,000 games TD-Gammon turned out to be a good match against the human grand-master Robertie. After this TD-Gammon was enhanced by a 3-ply look-ahead strategy that made it even stronger. Currently, TD-Gammon is still probably the best backgammon playing program in the world, but other programs such as BGBlitz from Frank Berger or Fredrik Dahl's Jellyfish also rely on neural networks as evaluation functions and obtained a very good playing level. All of these programs are much better than Berliner's backgammon playing program BKG [27] which was implemented using human designed weighted symbolic rules to get an evaluation function.

### 4.1 Learning an Expert Backgammon Program

We use an expert backgammon program against which we can train other learning programs and which can be used for generating games that can be observed by a learning program. Furthermore, in later experiments we can evaluate the learning programs by playing test-games against this expert. To make the expert player we used TD-learning combined with learning from self-play using hierarchical neural network architecture. This program was trained by playing more than 1 million games against itself. Since the program was not always improving by letting it play more training games, we tested the program after each 10,000 games for 5,000 test games against the best previous saved version. Then we recorded the score for each test and the weights of the network architecture with the highest score were saved. Then after each 100,000 games we made a new opponent which was the previous network with the highest score over all tests and this program was also used as learning program and further trained by self-play while testing it against the previous best program. This was repeated until there was no more progress, *i.e.* the learning program was not able to significantly beat the previous best learned program anymore. This was after more than 1,000,000 training games.

**Architecture.** We used modular neural network architecture, since different strategic positions require different knowledge for evaluating the positions [28]. Therefore we used a neural network architecture consisting of the following 9 neural networks for different strategic position classes, and we also show how many learning examples these networks received during training this architecture by self-play:

1) One network for the endgame; all stones are in the inner-board for both players or taken out (10.7 million examples).

2) One network for the racing game or long endgame; the stones can not be beaten anymore by another stone (10.7 million examples).

3) One network for positions in which there are no stones on the bar or stones in the first 6 fields for both players (1.9 million examples).

4) One network if the player has a prime of 5 fields or more and the opponent has one piece trapped by it (5.5 million examples).

5) One network for back-game positions where one player has a significant pip-count disadvantage and at least three stones in the first 6 fields (6.7 million examples).

6) One network for a kind of holding game; the player has a field with two stones or more or one of the 18, 19, 20, or 21 points (5.9 million examples).

7) One network if the player has all its stones further than the 8 point (3.3 million examples).

8) One network if the opponent has all its stones further than the 8 point (3.2 million examples).

9) One default network for all other positions (34.2 million examples).

For each position which needs to be evaluated, our symbolic categorization module uses the above rules to choose one of the 9 networks to evaluate (and learn) a position. The rules are followed from the first category to the last one, and if no rule applies then the default category and network is used.

**Input features.** Using this modular design, we also used different features for different networks. E.g., the endgame network does not need to have inputs for all fields since all stones have been taken out or are in the inner-board of the players. For the above mentioned neural network modules, we used different inputs for the first (endgame), second (racing game), and other (general) categories. The number of inputs for them is:

1) For the endgame we used 68 inputs, consisting of 56 inputs describing raw input information and 12 higher level features.

2) For the racing game (long endgame) we used 277 inputs, consisting of the same 68 inputs as for the endgame, another 192 inputs describing the raw board information, and 17 additional higher level features.

3) For the rest of the networks (general positions) we used 393 inputs consisting of 248 inputs describing raw board information and 145 higher level features including for example the probabilities that stones can be hit by the opponent in the next move.

For the neural networks we used 7 output units in which one output learned on the average result and the other six outputs learned a specific outcome (such as winning with 3, 2, or 1 point or losing with 3, 2, or 1 point). The good thing of using multiple output units is that there is more learning information going in the networks. Therefore the hidden units of the neural networks need to be useful for storing predictive information for multiple related subtasks, possibly resulting in better representations [29]. For choosing moves, we combined the average output with the combined outputs of the other output neurons to get a single board position evaluation. For this we took the average of the single output (with a value between  $-3$  and  $3$ ) and the combined value of the other outputs times their predicted probability values. Each output unit only learned from the same output unit in the next positions using TD-learning (so the single output only learned from its own evaluations of the next positions). Finally, the number of hidden units (which use a sigmoid activation function) was 20 for the endgame and long endgame, and 40 for all other neural networks. We call the above described network architecture the large neural network architecture and trained it by self-play using  $TD(\lambda)$  learning with a learning rate of 0.01, a discount factor  $\gamma$  of 1.0, and a value for  $\lambda$  of 0.6. After learning we observed that the 2 different evaluation scores were always quite close and that the 6 output units usually had a combined activity close to 1.0 with only sometimes small negative values (such as  $-0.002$ ) for single output units if the probability of the result was 0, which only have a small influence on the evaluation of a position.

Now we obtained an expert program, we can use it for our experiments in analyzing the results of new learners that train by self-play, train by playing against this expert, or learn by viewing games played by the expert against itself.

## 4.2 Experiments with Learning Backgammon

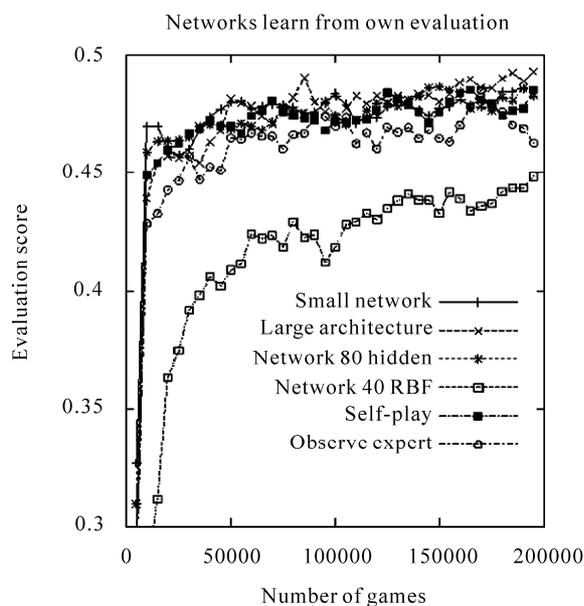
We first made a number of simulations in which 200,000 training games were used and after each 5,000 games we played 5,000 test games between the learner and the expert to evaluate the learning program. Because these simulations took a lot of time (several days for one simulation), they were only repeated two times for every setup.

The expert program was always the same as described before. For the learning program we also made use of a smaller architecture consisting of three networks; one for the endgame of 20 hidden units, one for the long end-

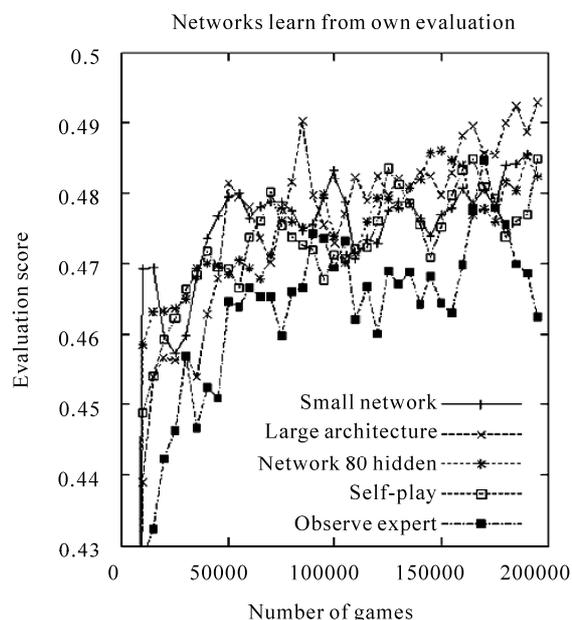
game (racing game) of 20 hidden units, and one for the other board positions with 40 hidden units. We also used a larger network architecture with the same three networks, but with 80 hidden units for the other board positions, and finally we used an architecture with 20, 20, 40 hidden units with a kind of radial basis activation function:  $H_j = e^{-(\sum W_{ij} I_i + b_j)^2}$ . These architectures were trained by playing training games against the expert. We also experimented with a small network architecture that learns by self-play or by observing games played by the expert against itself.

Because the evaluation scores fluctuate a lot during the simulation, we smoothed them a bit by replacing the evaluation of each point (test after  $n$  games) by the average of it and its two adjacent evaluations. Since we used 2 simulations, each point is therefore an average of 6 evaluations obtained by testing the program 5,000 games against the expert (without the possibility of doubling the cube). For all these experiments we used extended back-propagation [30] and TD( $\lambda$ )-learning with a learning rate of 0.01 and an eligibility trace factor  $\lambda$  of 0.6 that gave the best results in preliminary experiments. **Figures 1 and 2** show the obtained results.

First of all, it can be noted that the neural network architecture with RBF like activation functions for the hidden units works much worse. Furthermore, it can be seen that most other approaches work quite well and reach equity of almost 0.5. **Table 1** shows that all architectures, except for the architecture using RBF neurons, obtained an equity higher than 0.5 in at least one of



**Figure 1. Results for different architectures from learning against the expert, and the small architecture that learns by self-play or by observing games of the expert**



**Figure 2. Results for different architectures from learning against the expert, and the small architecture that learns by self-play or by observing games of the expert. More detailed plot without the architecture with RBF hidden units**

**Table 1. Results for the different methods as averages of 6 matches of 5,000 games played against the expert. Note that the result after 5,000 games is the average of the tests after 100, 5000, and 10000 games**

Architecture	5000	100,000	175,000	Max after	Max eval
Small Network	0.327	0.483	0.478	190,000	0.508
Large architecture	0.290	0.473	0.488	80,000	0.506
Network 80 hidden	0.309	0.473	0.485	155,000	0.505
Network 40 RBF	0.162	0.419	0.443	120,000	0.469
Small network Self-play	0.298	0.471	0.477	200,000	0.502
Small network Observing	0.283	0.469	0.469	110,000	0.510

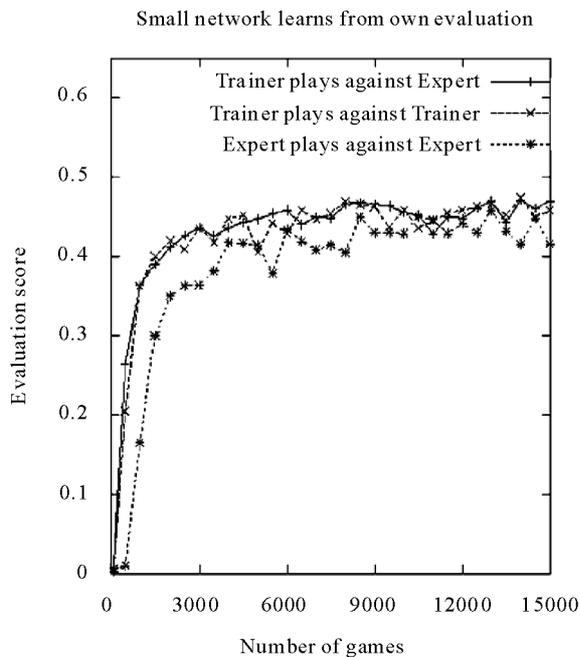
the 80 tests. Testing these found solutions 10 times for 5000 games against the expert indicated that their playing strengths were equal. If we take a closer look at **Figure 2**, we can see that the large architecture with many module finally performs a bit better than the other approaches and that learning by observing the expert reaches a slightly worse performance.

**Smaller simulations.** We also performed a number of smaller simulations of 15,000 training games where we tested after each 500 games for 500 testing games. We repeated these simulations 5 times for each neural network architecture and method for generating training

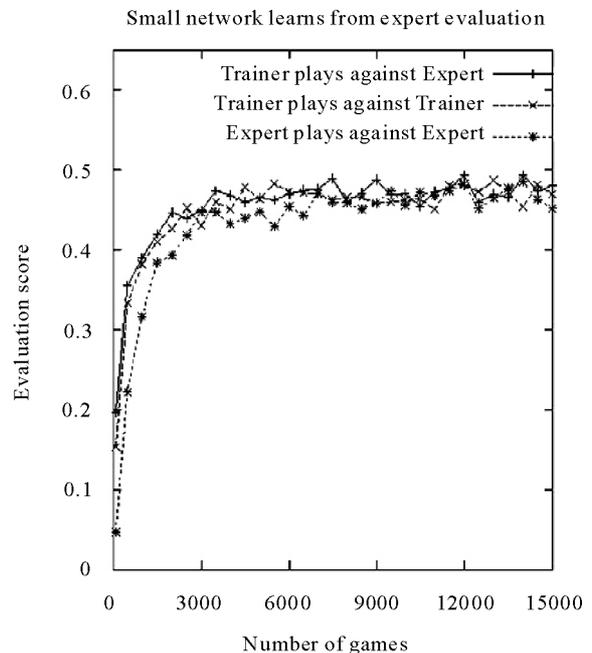
games. Because there is an expert available with the same kind of evaluation function, it is also possible to learn with TD-learning using the evaluations of the expert itself. This is very similar to supervised learning, although the agent generates its own moves (depending on the method for generating games). In this way, we can analyze what the impact of bootstrapping on an initially bad evaluation function is compared to learning immediately from outputs for positions generated by a better evaluation function. Again we used extended back-propagation [30] and TD( $\lambda$ ) with a learning rate of 0.01 and set  $\lambda = 0.6$ .

In **Figure 3**, we show the results of the smaller architecture consisting of three networks with 20, 20, and 40 hidden units. We also show the results in **Figure 4** where we let the learning programs learn from evaluations given by the expert program, but for which we still use TD-learning on the expert's evaluations with  $\lambda = 0.6$  to make training examples.

The results show that observing the expert play and learning from these generated games progress slower and reach slightly worse results within 15,000 games if the program learns from its own evaluation function. In **Figure 4** we can see faster learning and better final results if the programs learn from the expert's evaluations (which is like supervised learning), but the differences are not very large compared to learning from the own evaluation function. It is remarkable that good performance



**Figure 3.** Results for the small architecture when using a particular method for generating games. The evaluation on which the agent learns is its own



**Figure 4.** Results when the expert gives the evaluations of positions

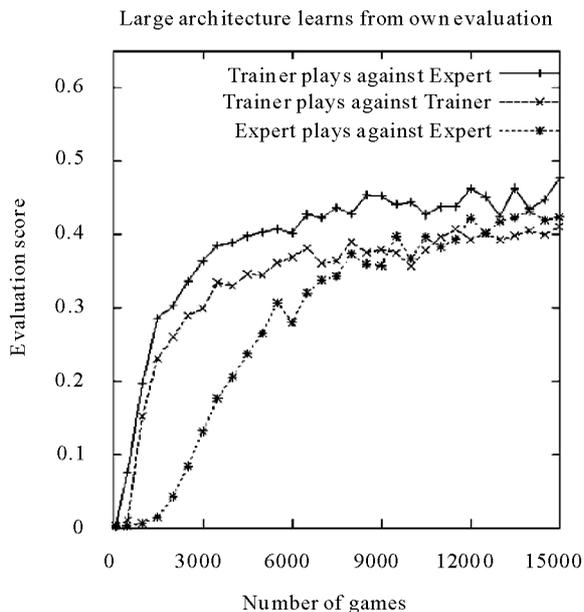
has already been obtained after only 5,000 training games.

In **Table 2** we can see that if we let the learning program learn from games played against the expert, in the beginning it almost always loses (its average test-result or equity after 100 training games is 0.007), but already after 500 training games the equity has increased to an average value of 0.26. We can conclude that the learning program can learn its evaluation function by learning from the good positions of its opponent. This good learning performance can be attributed to the minimax TD-learning rule, since otherwise always losing will quickly result in a simple evaluation function that always returns a negative result. However, using the minimax TD-learning rule, the program does not need to win many games in order to learn the evaluation function. Learning by self-play performs almost as good as learning from playing against the expert. If we use the expert's evaluation function then learning progresses much faster in the beginning, although after 10,000 training games almost the same results are obtained. Learning by observing the expert playing against itself progresses slower and reaches worse results if the learning program learns from its own evaluation function. If we look at the learning curve, we can still see that it is improving however.

We repeated the same simulations for the large architecture consisting of 9 modules. The results are shown in **Figures 5 and 6**. The results show that learning with the large network architecture progresses much slower, which can be explained by the much larger number of

**Table 2. Results for the three different methods for generating training games with learning from the own or the expert's evaluation function. The results are averages of 5 simulations**

Method	Eval-function	100	500	1000	5000	10,000
Self-play	Own	0.006	0.20	0.36	0.41	0.46
Self-play	Expert	0.15	0.33	0.38	0.46	0.46
Against expert	Own	0.007	0.26	0.36	0.45	0.46
Against expert	Expert	0.20	0.35	0.39	0.47	0.47
Observing expert	Own	0.003	0.01	0.16	0.41	0.43
Observing expert	Expert	0.05	0.22	0.32	0.45	0.46

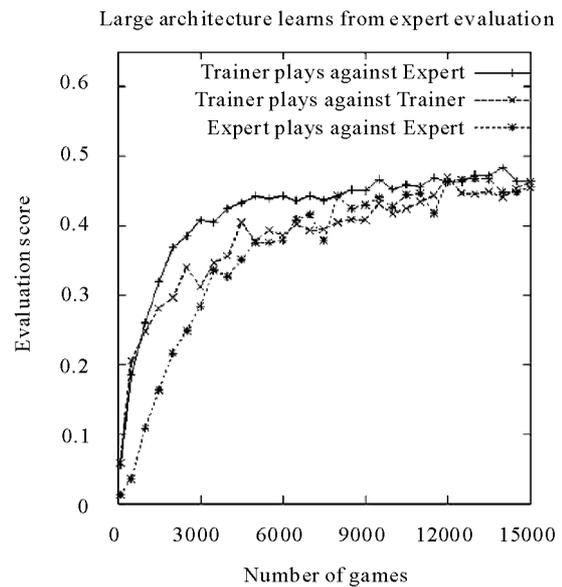


**Figure 5. Results for the large architecture when using a particular method for generating games. The evaluation on which the agent learns is its own**

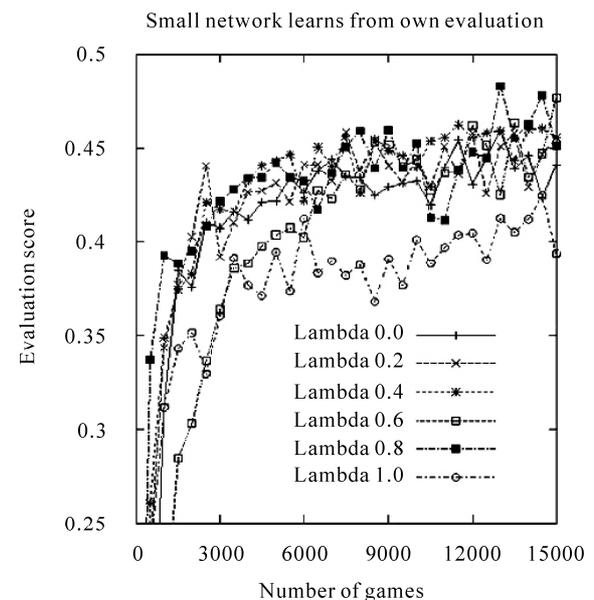
parameters which need to be trained and the fewer examples for each individual network. The results also show that learning from observing the expert play against itself performs worse than the other methods, although after 15,000 games this method also reaches quite high equities, comparable with the other methods. The best method for training the large architecture is when games are generated by playing against the expert. **Figure 6** shows faster progress if the expert's evaluations are used.

**Effect of  $\lambda$ .** Finally, we examine what the effect of different values for  $\lambda$  is when the small architecture learns by playing against the expert. We tried values for  $\lambda$  of 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0. When using  $\lambda = 1$  we needed to use a smaller learning-rate, since otherwise initially the

weights became much too large. Therefore we used a learning rate of 0.001 for  $\lambda = 1.0$  and a learning rate of 0.01 for the other values for  $\lambda$ . **Figure 7** shows the results averaged over 5 simulations. It can be seen that a  $\lambda$ -value of 1.0 works much worse and that values of 0.6 or 0.8 perform the best. **Table 3** shows the results after 100, 500, 1000, 5000, and 10,000 games. We can see that higher values of  $\lambda$  initially result in faster learning which



**Figure 6. Results for the large architecture when using a particular method for generating games. Results when the expert gives the evaluations**



**Figure 7. Results for the small architecture when using different values for  $\lambda$ . The games are generated by self-play**

**Table 3. Results for different values of  $\lambda$  when the small architecture learns against the expert**

$\lambda$	100	500	1000	5000	10,000
0.0	0.004	0.13	0.31	0.42	0.43
0.2	0.002	0.24	0.34	0.43	0.45
0.4	0.002	0.26	0.35	0.44	0.44
0.6	0.007	0.26	0.36	0.45	0.46
0.8	0.06	0.34	0.39	0.44	0.45
1.0	0.12	0.23	0.31	0.39	0.40

can be explained by the fact that bootstrapping from the initially random evaluation function does not work too well and therefore larger eligibility traces are profitable. After a while  $\lambda$  values between 0.2 and 0.8 perform all similarly.

### 4.3 Discussion

Learning a good evaluation function for backgammon with temporal difference learning appears to succeed very well. Already within few thousands of games which can be played in less than one hour a good playing level is learned with equity of around 0.45 against the expert program. We expect this equity to be similar to a human player who regularly plays backgammon. The results show that learning by self-play and by playing against the expert obtain the same performance.

Learning by observing an expert play progresses approximately two or three times slower than the other methods. In our current experiments the learning program observed another program that still needed to select moves. Therefore there was no computational gain in generating training games. However, if we would have used a database, then in each position also one-step look-ahead would not be needed. Since the branching factor for a one-step look-ahead search is around 16 for backgammon, we would gain 94% of the computational time for generating and learning from a single game. Therefore learning from database games could still be advantageous compared to learning by self-play or playing against an expert. A problem of using a (small) database is that overfitting the evaluation function may occur. This may be solved by combining this approach with learning by self-play. In the large experiment, the learning behavior of the method that learns by observing the expert is a bit more fluctuating, but it still obtained equity a bit larger than 0.5 during one of the test-games in the large experiment and additional tests indicated that its playing strength at that point was equal to the expert player.

We also noted that training large architectures initially takes longer which can be simply explained by the larger number of parameters which need to be learned and fewer examples for individual modules. After training for a longer time, such bigger architectures can reach higher performance levels than smaller architectures. We note that since the agent learns on the same problem as on which it is tested, in these cases overfitting does not occur. A large value for  $\lambda$  (larger than 0.8) initially helps to improve the learning speed, but after some time smaller values for  $\lambda$  (smaller than 0.8) perform better. An annealing schedule for  $\lambda$  may therefore be useful. Finally we observed in all experiments that the learning programs are not always improving by playing more games. This can be explained by the fact that there is no convergence guarantee for RL and neural networks. Therefore testing the learning program against other fixed programs on a regular basis is necessary to be able to save the best learning program. It is interesting to note the similarity to evolutionary algorithms evolving game playing programs which also use tests. However, we expect that temporal difference learning and gradient descent is better for fine-tuning the evaluation function than a more randomized evolutionary search process.

Another approach that receives a lot of attention in recent RL research and good results for particular control problems is kernel-based least policy iteration (LSPI) learning [31]. However, it is unlikely that RBF kernels will generalize well to the huge state space of backgammon and that therefore kernel based LSPI is not likely to be successful. In fact, we implemented Support vector machines with RBF kernels for the game of Othello, and this showed indeed that RBF kernels are not good for games involving huge state-spaces. For this sigmoid functions are needed, but they are difficult to use as kernels, since they require a lot of structural design. The use of neural networks with sigmoid activation functions is therefore the current method of choice for difficult games.

## 5. Conclusions

In this paper different strategies for obtaining training examples for learning game evaluation functions have been examined. The possible advantage of playing against or observing an expert, namely that games are initially played at a high level was not clearly shown in the experimental results. We will now return to our research questions and answer them here.

1) Question 1. Which method combined with temporal difference learning results in the best performance after a fixed number of games? Is observing an expert player, playing against an expert, or self-play the best method?

**Answer.** The results indicate that observing an expert play is the worst method. The reason can be that the learning program is never actively involved in playing

and therefore can not learn to penalize particular moves that it may have overestimated. Learning by playing against an expert seems to be the best strategy. Another approach that could be useful is learning from the expert combined with learning by self-play.

2) Question 2. When the learning program immediately receives accurate evaluations of encountered board positions, will it then learn faster than when it uses its initially randomized function approximator and TD-learning to estimate the board evaluations?

**Answer.** Initially, learning goes much faster when accurate evaluations are given. However, after 10,000 training games, the disadvantage of the initially randomized function approximator has almost disappeared.

3) Question 3. Is a function approximator with more trainable parameters more efficient for learning to play the game of backgammon than a smaller representation?

**Answer.** Yes, in general the larger function approximators obtain better performance levels, although in the beginning they learn at a slower rate. Since the agent is tested on exactly the same problem as on which it is trained (different from supervised learning), overfitting does not occur in reinforcement learning.

4) Question 4. Which value for  $\lambda$  in TD( $\lambda$ ) works best for obtaining the best performance after a fixed number of games?

**Answer.** Initially larger values for  $\lambda$  result in a faster learning rate. However, the final performance is best for intermediate values of  $\lambda$  around 0.6. It should be noted that this observation is quite problem specific.

**Future work.** Although in this paper it was demonstrated that learning from observing an expert is not profitable to learn to play backgammon, we also mentioned some advantages of using an expert or a database. Advantages of learning from experts are that the system does not explore the whole huge state-space and that in some applications it is a safer method for obtaining experiences than learning by trial-and-error. Furthermore, learning game evaluation functions from databases has the advantage that no look-ahead during game-play is necessary.

Learning from experts or databases can also be used for other applications, such as learning in action or strategic computer games for which human games played with a joystick can be easily recorded. Furthermore, for therapy planning in medicine, databases of therapies may be available and could therefore be used for learning policies. For robotics, behavior may be steered by humans and these experiences can be recorded and then learned by the robot [32]. Thus, we still think that learning from observing an expert has many advantages and possibilities for learning control knowledge, although care should be taken that the learner tries out its own behavior during learning.

## REFERENCES

- [1] L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 237-285.
- [2] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," The MIT press, Cambridge, MA, 1998.
- [3] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, Vol. 3, 1988, pp. 9-44.
- [4] J. B. Pollack and A. D. Blair, "Why Did TD-Gammon Work," In: D. S. Touretzky, M. C. Mozer and M. E. Hasselmo, Ed., *Advances in Neural Information Processing Systems* 8, MIT Press, Cambridge, MA, 1996, pp. 10-16.
- [5] D. B. Fogel, "Evolving a Checkers Player without Relying on Human Experience," *Intelligence*, Vol. 11, No. 2, 2000, pp. 20-27.
- [6] D. E. Moriarty, "Symbiotic Evolution of Neural Networks in Sequential Decision Tasks," PhD thesis, Department of Computer Sciences, The University of Texas at Austin, USA, 1997.
- [7] G. Tesauro, "Practical Issues in Temporal Difference Learning," In: D. S. Lippman, J. E. Moody and D. S. Touretzky, Ed., *Advances in Neural Information Processing Systems* 4, Morgan Kaufmann, San Mateo, CA, 1992, pp. 259-266.
- [8] G. J. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, Vol. 38, 1995, pp. 58-68.
- [9] S. Thrun, "Learning to Play the Game of Chess," In: G. Tesauro, D. Touretzky and T. Leen, Ed., *Advances in Neural Information Processing Systems* 7, Morgan Kaufmann, San Francisco, CA, 1995, pp. 1069-1076.
- [10] J. Baxter, A. Tridgell and L. Weaver, "Knightcap: A Chess Program that Learns by Combining TD( $\lambda$ ) with Minimax Search," Technical report, Australian National University, Canberra, 1997.
- [11] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal on Research and Development*, Vol. 3, No. 3, 1959, pp. 210-229.
- [12] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers II—Recent Progress," *IBM Journal on Research and Development*, Vol. 11, No. 6, 1967, pp. 601-617.
- [13] J. Schaeffer, M. Hlynka and V. Hussila, "Temporal Difference Learning Applied to a High-Performance Game," *In Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, WA, USA, 2001, pp. 529-534.
- [14] N. N. Schraudolph, P. Dayan and T. J. Sejnowski, "Temporal Difference Learning of Position Evaluation in the Game of Go," In: J. D. Cowan, G. Tesauro and J. Alsppector, Ed., *Advances in Neural Information Processing Systems*, Morgan Kaufmann, San Francisco, CA, 1994, pp. 817-824.
- [15] J. Furnkranz, "Machine Learning in Games: A Survey," In: J. Furnkranz and M. Kubat, Ed., *Machines that learn to Play Games*, Nova Science Publishers, Huntington,

- NY, 2001, pp. 11-59.
- [16] A. Plaat, "Research Re: search and Re-search," PhD thesis, Erasmus University Rotterdam, Holland, 1996.
- [17] J. Schaeffer, "The Games Computers (and People) Play," *Advances in Computers*, Vol. 50, 2000, pp. 189-266.
- [18] J. Schaeffer and A. Plaat, "Kasparov versus Deep Blue: The Re-match," *International Computer Chess Association Journal*, Vol. 20, No. 2, 1997, pp. 95-102.
- [19] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," *Proceedings of the fifth International Conference on Computers and Games*, Turin, Italy, 2006, pp. 72-83.
- [20] R. Bellman, "Dynamic Programming," Princeton University Press, USA, 1957.
- [21] J. N. Tsitsiklis, "Asynchronous Stochastic Approximation and Q-learning," *Machine Learning*, Vol. 16, 1994, pp. 185-202.
- [22] A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike Adaptive Elements that Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, 1983, pp. 834-846.
- [23] M. A. Wiering and J. H. Schmidhuber, "Fast Online  $Q(\lambda)$ ," *Machine Learning*, Vol. 33, No. 1, 1998, pp. 105-116.
- [24] C. M. Bishop, "Neural Networks for Pattern Recognition," Oxford University, New York, 1995.
- [25] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation," In: D. E. Rumelhart and J. L. McClelland, Ed., *Parallel Distributed Processing*, MIT Press, USA, 1986, pp. 318-362.
- [26] L.-J. Lin, "Reinforcement Learning for Robots Using Neural Networks," PhD thesis, Carnegie Mellon University, Pittsburgh, 1993.
- [27] H. Berliner, "Experiences in Evaluation with BKG—A Program that Plays Backgammon," *In Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 1, 1977, pp. 428-433.
- [28] J. A. Boyan, "Modular Neural Networks for Learning Context-Dependent Game Strategies," Master's thesis, University of Chicago, USA, 1992.
- [29] R. Caruana and V. R. de Sa, "Promoting Poor Features to Supervisors: Some Inputs Work Better as Outputs," In: M. C. Mozer, M. I. Jordan and T. Petsche, Ed., *Advances in Neural Information Processing Systems 9*, Morgan Kaufmann, San Mateo, CA, 1997, pp.246-252.
- [30] A. Sperduti and A. Starita, "Speed up Learning and Network Optimization with Extended Backpropagation," *Neural Networks*, Vol. 6, 1993, pp. 365-383.
- [31] X. Xu, D. Hu and X. Lu, "Kernel-Based Least Squares Policy Iteration for Reinforcement Learning," *IEEE Transactions on Neural Networks*, Vol. 18, No. 4, 2007, pp. 973-992.
- [32] W. Smart and L. Kaelbling, "Effective Reinforcement Learning for Mobile Robots," *Proceedings of the IEEE International Conference on Robotics and Automation*, Washington, DC, USA, 2002, pp. 3404-3410.