Learning to Play Board Games using Temporal Difference Methods

Marco A. Wiering

Jan Peter Patist

Henk Mannen

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-048

www.cs.uu.nl

Learning to Play Board Games using Temporal Difference Methods

Marco A. Wiering
Intelligent Systems Group
Utrecht University
marco@cs.uu.nl

Jan Peter Patist
Artificial Intelligence
Vrije Universiteit Amsterdam
jpp@few.vu.nl

Henk Mannen
Defense Security and Safety
TNO
henk.mannen@tno.nl

December 2, 2005

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 themselves. 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. We compared these three methods using temporal difference methods to learn the game of backgammon. For particular games such as draughts and chess, learning from a large database containing games played by human experts has as a large advantage that during the generation of (useful) training games, no expensive lookahead planning is necessary for move selection. Experimental results in this paper show how useful this method is for learning to play chess and draughts.

Keywords: Board Games, Reinforcement Learning, TD(λ), 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 [18, 41] techniques, in particularly temporal difference learning [39, 41] for learning a backgammon evaluation function from training games generated by letting the program play against itself. This had 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 [25], checkers [14], and othello [22] and were quite successful. Reinforcement learning has been applied to learn a variety of games, including backgammon [43, 44], chess [46, 2], checkers [28, 29, 31], and Go [34]. 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 [16] 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 lookahead 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 higher level representations of the input is used.

A simple strategy for evolving an evaluation function for a game using EAs is by learning a neural network using coevolution [14, 25]. Here initially two neural networks are initialized which play some number of games against themselves. After this the winner is selected and is allowed to mutate its neural network weights to create a new opponent. Then the previous winner and its mutated clone play a number of test games and this is repeated many times. More complex forms of coevolution in which multiple tests are evolved and used for evaluating players also exist. In this approach the aim is to evolve the ideal set of tests for perfectly evaluating and comparing a number of different learners. It has been shown that co-evolutionary approaches that evolve the tests can learn to approximate this ideal set of tests [13].

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 [39, 43] uses the difference between two successive positions for backpropagating 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 feedforward neural networks will find or converge to an optimal value function.

Although temporal difference learning is quite well understood [39, 41], for its use for control problems instead of only prediction, we have to deal with the exploration/exploitation dilemma [45] — the learning agent should trade-off the goals of obtaining as much reward as possible and exploring the state space at the same time. In games this is an issue on its own, since in deterministic games such as draughts or Go we need to have exploration, but since TD-learning is used this will have consequences for the learning updates.

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.

One advantage of learning from games provided by another expert or a database is that games are immediately played at a high level instead of completely random when the agent would play its own games. Another advantage is that for particular games such as draughts, chess, and Go, usually expensive lookahead searches are necessary to choose a good move. These games have in common that there are many tactical lines of play which can force the opponent's move so that the player can reach an advantageous position. Since lookahead is expensive, learning from a database of played games would save a huge amount of computation time. E.g., if 1000 positions are evaluated to select a move, training from recorded games provided by some database would save 99.9% of the computational cost of generating the training games and learning from them. Since generating a new game is much more expensive than learning from a game, using recorded games can be very useful. A disadvantage of learning from database games or from observing an expert program play is that the learning agent is never allowed to try the action which it would prefer. Basically, the exploration is governed by human decisions and there is no exploitation. Therefore, the agent might remain biased to particular moves which the experts would never select and are therefore never punished. This is different from the case where the agent selects its own moves so that it can observe that moves currently preferred are not as good as expected. Another possible problem when learning from database games without search is that it is questionable whether TD learning must be integrated with search such as in Knightcap [2] and TDLeaf-learning [5]. For example, a problem when learning from successive moves is when there is a queen trade. In that case the TD method will experience a large difference in material and thereby introduce variance in the update. When learning from the leaves of the principal variations, which are usually at the end of quiescence search, the evaluations of the positions will be much smoother. However, it still may be well possible to learn an evaluation function without search as long as the evaluation function is used with search during tournament games. We will examine in this article whether learning from game demonstrations for the games backgammon, draughts, and chess is fruitful.

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 presents experiments with the games of draughts and chess for which we used databases containing many human expert games. Finally, section 6 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:

- A set of possible board positions
- A set of legal moves in a position
- Rules for carrying out moves

- Rules for deciding upon termination and the result of a game

A game playing program consists of a move generator, a lookahead algorithm, and an evaluation function. The move generator just generates all legal moves, possibly in some specific order (taking into account some priority). The lookahead algorithm deals with inaccurate evaluation functions. If the evaluation function would be completely accurate, lookahead 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 [24, 30] 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 [33], then it relies heavily on the use of very fast computers and lookahead 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 lookahead algorithms for comparing a large number of positions. Expert backgammon playing programs only use 3-ply lookahead, 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 lookahead the dice are no computational problem, this makes the branching factor for computing possible positions after two or more moves much larger (since then lookahead needs to take into account the 21 outcomes of the two dice). This is the reason why looking ahead many moves in stochastic games is infeasible for human experts or computers. For this Monte Carlo simulations 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 more smooth 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 results 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.

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 loosing 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 [44] relies on this smoothness of the evaluation function. It is well known that learning smooth functions requires less parameters for a machine learning algorithm and therefore faster search for a good solution and better generalization.

In order not to bias our results too much towards one game, we analysed using temporal difference (TD) learning for three difficult games; backgammon, chess, and draughts. In the next section we will explain how we can use TD methods for learning to play games.

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 $S = \{s^1, s^2, \ldots, s^n\}$, 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 the optimal policy for mapping states to actions. We denote the optimal deterministic policy as $\pi^*(s) \rightarrow a^*|s$. It is well known that for each MDP, one or more optimal deterministic policies exist. The optimal policy is defined as the policy which receives the highest possible cumulative discounted rewards in its future from all states.

In order to learn the optimal policy, value-function based reinforcement learning [39, 18, 41] 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 π:

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

The optimal policy is the one which has the largest state-value in all states. In many cases reinforcement learning algorithms used for learning to control an agent also make use of a Q-function for evaluating state-action pairs. Here $Q^\pi(s, a)$ is defined as the expected cumulative discounted future reward if the agent is in state s, executes action a, and follows policy π afterwards:

$$Q^\pi(s, a) = E(\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, a_0 = a, \pi)$$

It is easy to see that if the optimal Q-function, $Q^*(.)$ is known, that the agent can select optimal actions by selecting the action with the largest value in a state:

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

and furthermore the optimal value of a state should correspond to the highest action value in that state according to the optimal Q-function:

$$V^*(s) = \max_a Q^*(s, a)$$

It is also well-known that there exists a recursive equation known as the Bellman optimality equation [6] 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:

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

And the same holds for the optimal Q-function:

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

The Bellman equation has led to very efficient dynamic programming (DP) techniques for solving known MDPs [6, 41]. One of the most used DP algorithms is value iteration which uses the Bellman equation as an update:

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

Where $V^k(s) = \max_a Q^k(s,a)$. In each step the Q-function looks ahead one step, using this recursive update rule. It can be easily shown that $\lim_{k\to\infty} Q^k = Q^*$, when starting from an arbitrary Q-value function Q^0 containing only finite values.

In a similar way we can use value iteration to compute the optimal V-function without storing the Q-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:

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

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 modelled as an MDP; secondly, for large or continuous state-spaces, a 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) [39] 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 Q-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:

$$\pi(s) = \arg\max_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V(s'))$$

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 [43, 44].

There exists a whole family of temporal difference learning algorithms known as TD(λ)-algorithms [39] which are parametrized by the value λ which makes the agent look further in the future for updating its value function. It has been proved [47] 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(λ)-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(λ) uses a factor $\lambda \in [0, 1]$ to discount TD-errors of future time steps:

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

where the TD(λ)-error δ_t^λ 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 (Barto et al., 1983; Sutton 1988). 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 [50]. The value of λ determines how much the updates are influenced by events that occurred much later in time. The extremes are TD(0) and TD(1) where (offline) 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 λ depends on the length of an epoch and varies between applications, although often a value between 0.6 and 0.9 works best.

Although temporal difference learning algorithms are very useful for evaluating a policy or for control if a model is available, we often also want to use reinforcement learning for learning optimal control in case no model of the environment is available. To do this, we need to learn Q-functions. One particular algorithm for learning a Q-function is Q-learning [48, 49]. Q-learning makes an update after an experience (s_t, a_t, r_t, s_{t+1}) as follows:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Q-learning is an off-policy reinforcement learning algorithm [41], which means that the agent learns about the optimal value function while following another behavioural policy which usually includes exploration steps. This has as advantage that it does not matter how much exploration is used, as long as the agent visits all state-action pairs an infinite number of times, Q-learning (with appropriate learning rate adaptation) will converge to the optimal Q-function [49, 17, 47, 42]. On the other hand, Q-learning does not learn

about its behavioural policy, so if the behavioural policy always receives low cumulative discounted rewards, then the agent does not try to improve it.

Instead of Q-learning, the on-policy algorithm SARSA for learning Q-values has been proposed in [27, 40]. SARSA makes the following update after an experience $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Tabular SARSA converges to the optimal policy under some conditions on the learning rate after an infinite number of steps if the exploration policy is GLIE (greedy in the limit of infinite exploration), which means that the agent should always explore, but stop exploring after an infinite number of steps [35]. Q-learning and SARSA can also be combined with eligibility traces [48, 41, 50].

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 feedforward neural networks [9].

In this paper we only consider fully-connected feedforward 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): $I_1, \ldots, I_{|I|}$, where $|I|$ is the number of input units, one hidden layer H with hidden units: $H_1, \ldots, H_{|H|}$, and one output layer with output units: $O_1, \ldots, O_{|O|}$. 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:

$$H_h = \sigma(\sum_{i=1}^{|I|} w_{ih} I_i + b_h)$$

Where $\sigma(x)$ is the Sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$
3) Compute the values for all output units O_o:

$$O_o = \sum_{h=1}^{|H|} w_{ho} H_h + b_o$$

Backpropagation. For training the system we can use the backpropagation algorithm [26]. 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 α. 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:

$$w_{ho} = w_{ho} + \alpha \delta_O(o) H_h$$

And finally we change all input-hidden weights:

$$w_{ih} = w_{ih} + \alpha \delta_H(h) I_i$$

So all we need is a desired output and then backpropagation can be used to compute weight updates to minimize the errorfunction on every different example. To get the desired output, we can simply use offline temporal difference learning [19] 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(λ) 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).

$$V'(s_T) = r_T \tag{1}$$

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 backpropagation 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:

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

In our experiments, we only used the minimax TD-update rule for backgammon, for draughts and chess we used the normal TD-update rule. An online version of TD(λ)

with neural networks would also be possible. Say that the neural networks consists of parameters w, we get the following update rule after each state transition:

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

where the eligibility traces of the parameters are updated by:

$$e_t = \gamma\lambda e_{t-1} + \frac{\partial V(s_t)}{\partial w}$$

Note that it does not matter how games are generated for the TD(λ) algorithm, the algorithm just learns from a played game starting at the initial position and ends in a final results. Therefore, for learning from demonstrations no adaptions have to be made to the learning algorithm.

3.4 Discussion

For learning to play board games, we usually do not want to use state-action or Q-functions, but prefer to use state or V-functions. The reason is that a model is available which can compute the resulting position for each move. Since it does not matter for a position which move was selected to arrive there, using the state function saves a lot of information about actions which needs to be learned and stored otherwise.

Dynamic programming could in theory be used to compute a perfect evaluation function using a tabular representation, but in practice this is infeasible due to the huge number of positions which need to be stored and evaluated. Therefore we apply reinforcement learning with function approximators that can generalize. The advantage is that RL only learns good evaluations for board positions which occur often and since it can generalize, not every possible position needs to be encountered. A problem with using function approximators such as neural networks, is that convergence to an optimal value function cannot be guaranteed. In practice we observed that longer learning did not always improve the quality of the evaluation function.

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 lookahead strategy making 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 [7] which was implemented using human designed weighted symbolic rules to get an evaluation function.

4.1 Learning an Expert Backgammon Playing 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 a 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 was 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 a modular neural network architecture, since different strategical positions require different knowledge for evaluating the positions [10]. Therefore we used a neural network architecture consisting of the following 9 neural networks for different strategical position classes, and we also show how many learning examples these networks received during training this architecture by self-play:

- One network for the endgame; all stones are in the inner-board for both players or taken out (10,7 million examples).
- One network for the racing game or long endgame; the stones cannot be beaten anymore by another stone (10.7 million examples).
- 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).
- 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).
- 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)
- 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).
- One network if the player has all its stones further than the 8 point (3.3 million examples).
- One network if the opponent has all its stones further than the 8 point (3.2 million examples).
- 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:

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

points or loosing 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 [11]. 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 (with values between 0 and 1) times their predicted results. 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(λ) learning with a learning rate of 0.01, a discount factor γ of 1.0, and a value for λ 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 that only have a small influence on the evaluation of a position.

In order to evaluate the quality of this trained large architecture we let it play against Frank Berger's BGBlitz[1], the champion of the world computer Olympics of backgammon in 2002 and 2003. We let them play two times 30 games where doubling was allowed. We made a very simple doubling strategy in which our player doubled if the evaluation was between 0.5 and 1.0, and the program took a double if the evaluation was higher than -0.5. The scores are given in Table 1.

Table 1: *Results from testing our trained expert against BGBlitz and results for BGBlitz with one move lookahead against BGBlitz with two moves lookahead.*

Player 1	Player 2	Tournament 1	Tournament 2
Large Expert	BGBlitz (1)	31 - 33	20 - 38
BGBlitz (2)	BGBlitz (1)	39 - 18	23 - 39

Unfortunately, the variance of the results in backgammon is very large and therefore testing two times 30 games is not enough to compare both programs. From the second tournament result it looks like BGBlitz plays better than the large architecture we trained, but we would need to play much more testing games to find out whether the difference is significant. Since we needed to test the games manually, testing much more games was infeasible since playing one test game costs 5 minutes. If we examine the results between BGBlitz looking ahead one or two moves, we also see a large variance and cannot easily compare both programs, even though we would expect BGBlitz to profit from the deeper game-tree search. What we observed, however, is that our program almost always played the same moves as BGBlitz in the same positions. Furthermore, if we only look at the number of games won, then it was (12 - 18) in the second tournament in the advantage of BGBlitz, but (17 - 13) in the first tournament won by the large architecture. It might therefore be the case that the doubling strategy based on the evaluations of BGBlitz worked better than ours, but even that is questionable to conclude with the amount of test games we played. Looking at the games as a human observer, it can be said that the games were of a high level containing many logical moves in the opening and middle-game and only some small mistakes in the endgame. Moves in the endgame of the large expert did not always take out most stones, which is usually the best move. The large architecture sometimes preferred to keep a nicer distribution on the board at the cost of

[1] See http://www.bgblitz.com.

taking pieces slower out. Although this does not really matter for positions with a clear advantage or disadvantage, for some endgames this is not the best thing to do. Although such mistakes did not occur often and most moves in the endgame were also optimal, we can easily circumvent this behavior by adding some rule which tells that the architecture should only consider moves which take out the maximal amount of stones.

Now we obtained an expert program, we can use it for our experiments in analysing 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 endgame (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: $H_j = e^{-(\sum_i w_{ij} I_i + b_j)^2}$. 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 backpropagation [38] and TD(λ)-learning with a learning rate of 0.01 and an eligibility trace factor λ of 0.6 that gave the best results in preliminary experiments. Figure 1 shows 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 an equity of almost 0.5. Table 2 shows that all architectures, except for the architecture using RBF neurons, obtained an equity higher than 0.5 in at least one of 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 1(Right), we can see that the large architecture with many modules 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 backpropagation [38] and TD(λ) with a learning rate of 0.01 and set $\lambda = 0.6$.

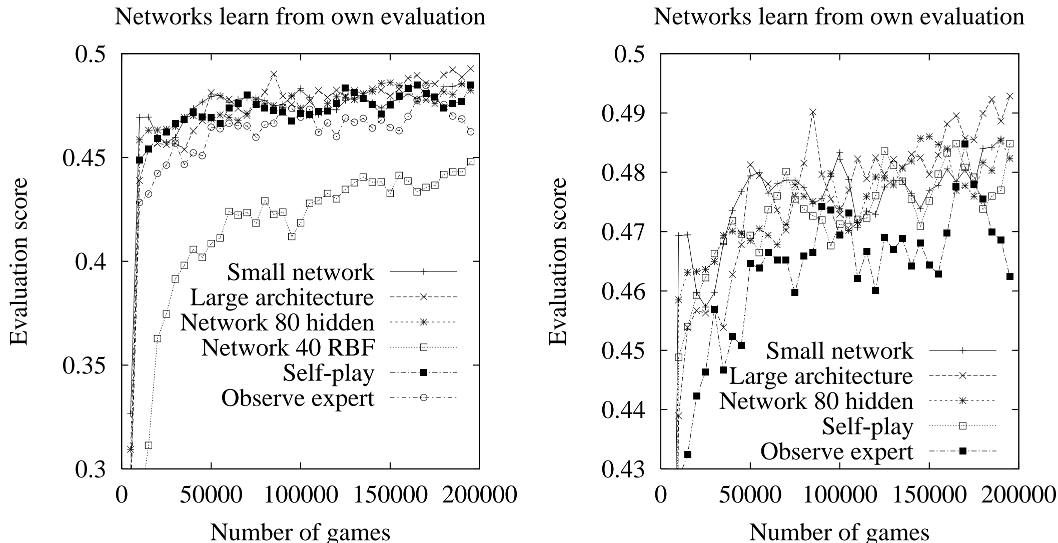

Figure 1: *(Left) Results for different architectures from learning against the expert, and the small architecture that learns by self-play or by observing games of the expert. (Right) More detailed plot without the architecture with RBF hidden units.*

Table 2: *Results for the different methods as averages of 6 matches of 5,000 games 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 expert	0.283	0.469	0.469	110,000	0.510

In Figure 2(Left), 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 2(Right) 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 (expert plays against expert) progresses slower and reaches slightly worse results within 15,000 games if the program learns from its own evaluation function. In Figure 2(Right) 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 has already been obtained after only 5,000 training games.

In Table 3 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

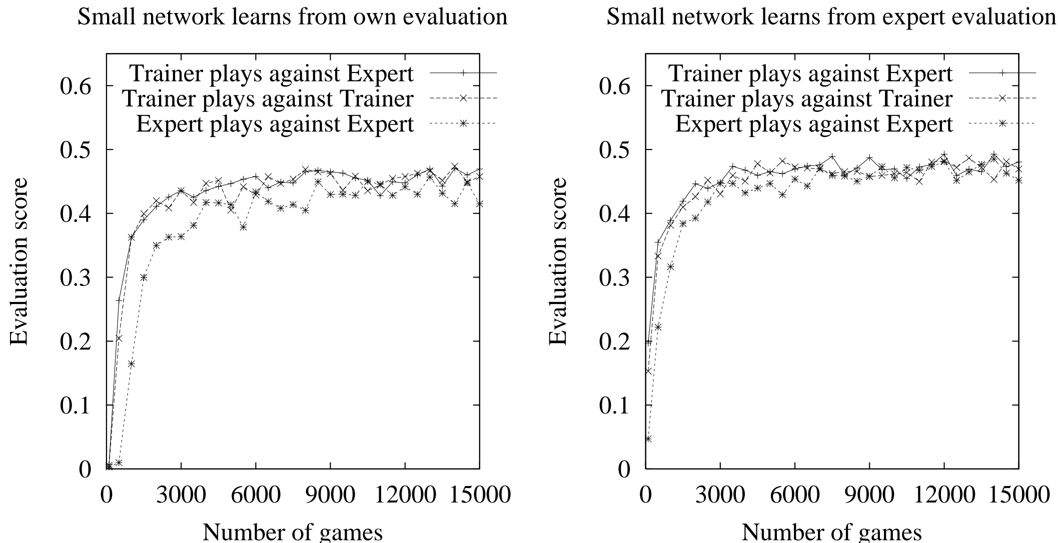

Figure 2: *(Left) Results for the small architecture when using a particular method for generating games. The evaluation on which the agent learns is its own. (Right) Results when the expert gives the evaluations of positions.*

Table 3: *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

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 loosing with 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 Figure 3. The results show that learning with the large network architecture progresses much slower, which can be explained by the much larger number of parameters which need to be trained and the fewer examples for each individual network.

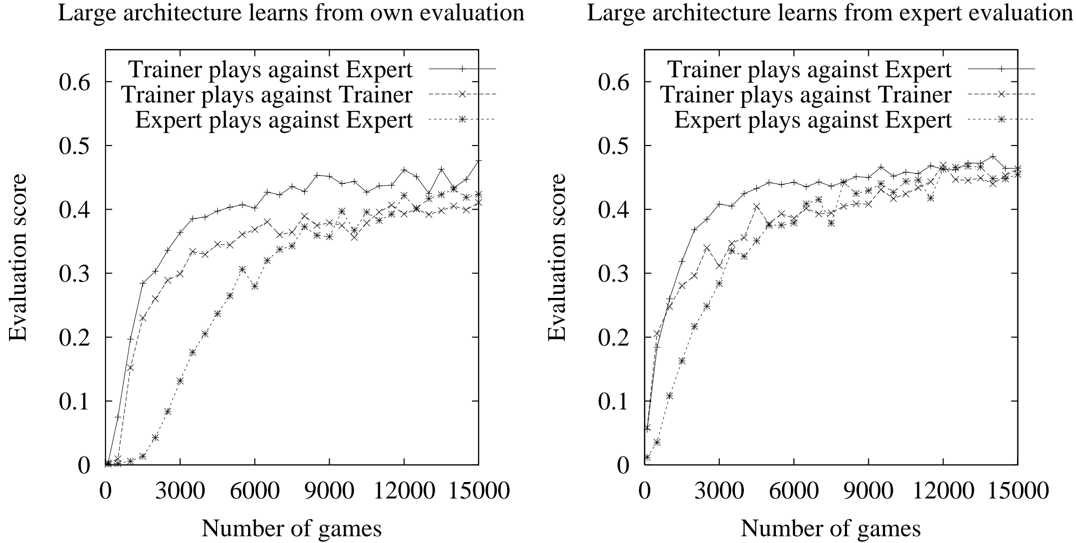

Figure 3: *(Left) Results for the large architecture when using a particular method for generating games. The evaluation on which the agent learns is its own. (Right) Results when the expert gives the evaluations.*

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 3(Right) shows faster progress if the expert's evaluations are used.

Effect of λ. Finally, we examine what the effect of different values for λ is when the small architecture learns by playing against the expert. We tried values for λ 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 λ. Figure 4 shows the results averaged over 5 simulations. It can be seen that a λ-value of 1.0 works much worse and that values of 0.6 or 0.8 perform the best. Table 4 shows the results after 100, 500, 1000, 5000, and 10,000 games. We can see that higher values of λ initially result in faster learning which 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 λ values between 0.2 and 0.8 perform all similarly.

Table 4: *Results for different values of λ when the small architecture learns from playing against the expert.*

λ	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

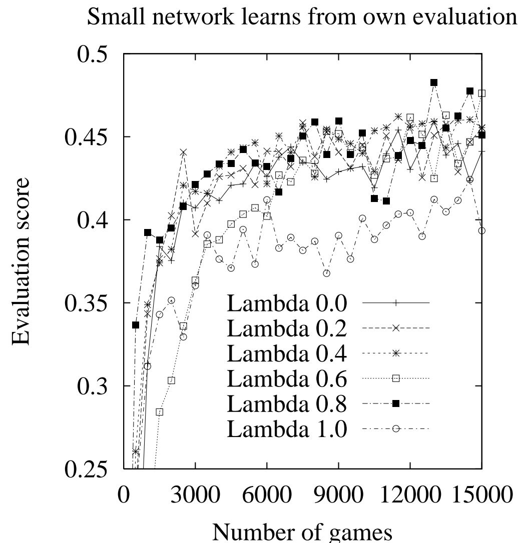

Figure 4: *Results for the small architecture when using different values for λ. The games are generated by self-play.*

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 an 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 lookahead would not be needed. Since the branching factor for a one-step lookahead 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. 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 an 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. A large value for λ (larger than 0.8) initially helps to improve the learning speed, but after some time smaller values for λ (smaller than 0.8) perform better. An annealing schedule for λ 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.

5 Learning to Play Chess and Draughts

In the previous section we noted that learning from observing games played by others is an alternative to learning by self-play or playing against an expert, especially if a database of games is available. For games such as draughts or chess, learning from database games has as a huge advantage that the time to generate training games is significantly reduced, since looking ahead many moves resulting in evaluations of thousands of positions is not necessary. We will examine in this section whether learning from database games using temporal difference learning leads to good learning programs for the games of chess and draughts. Furthermore, we will compare different neural network architectures consisting of a single or multiple networks and using different kinds of board features.

5.1 Learning to Play Chess

In recent years much progress has been made in the field of chess computing [15, 16]. Today's strongest chess programs are already playing at grand-master level. The evaluation function of these programs are programmed by translating available human chess knowledge into the function that is sometimes further optimized by adapting the function to prefer moves played in recorded human expert games. Notions such as *material balance*, *mobility*, *board control* and *connectivity* can be used to give an evaluation value for a board position.

Gary Kasparov was beaten in 1997 by the computer program Deep Blue in a match over six games by 3,5-2,5 [33]. Despite this breakthrough, world class human players are still considered playing better chess than computer programs. Chess programs still suffer problems with positions where the evaluation depends mainly on long-term positional features (e.g. pawn structure). This is rather difficult to solve because the positional character often leads to a clear advantage in a much later stadium than within the search depth of the chess program.

The programs can look very deep ahead nowadays, so they are quite good at calculating tactical lines. Winning material in chess usually occurs within a few moves and most chess programs have a search depth of at least 8 ply. Deeper search can occur for instance, when a tactical line is examined or a king is in check after normal search or if there are only a few pieces on the board. Humans are able to recognize patterns in positions and have therefore important information on what a position is about. Expert players are quite good at grouping pieces together into chunks of information, as was pointed out in the psychological studies by de Groot [12]. Computers analyze a position with the help of their chess knowledge. The more chess knowledge it has, the longer it takes for a single position to be evaluated. So the playing strength not only depends on the amount of knowledge, it also depends on the time it takes to evaluate a position, because less evaluation-time leads to deeper searches. It is a question of finding a balance between chess knowledge and search depth which is also called the search/knowledge trade-off [8]. Deep Blue for instance, relied mainly on a high search depth. Other programs focus more on chess knowledge and therefore have a relatively lower search depth.

A reinforcement learning chess program is Sebastian Thrun's NeuroChess [46]. NeuroChess has two separate neural networks. The explanation-based neural network [21] that predicts the value of an input vector or board position two ply (half moves) later and was trained on 120,000 expert games. Another network is the evaluation function that gives an output value for the input vector of 175 hand-coded chess features and which was

trained by TD-learning using the explanation-based neural network. NeuroChess uses the framework of the chess program GNU-Chess. The evaluation function of GNU-Chess was replaced by the trained evaluation function. NeuroChess defeated GNU-Chess in about 13% of the games, but was able to score 25% in the last 400 games.

Another RL chess program is KnightCap, which was developed by Baxter et al. [2, 3, 4]. It uses TDLeaf-learning [5], which is an enhancement of Sutton's TD(λ)-learning [39] for game learning programs. KnightCap uses a linear evaluation function and also uses a book learning algorithm that enables it to learn opening lines and end-games. This learning program learned from a 1650 player to a 2150 player in just 308 games against opponents on a chess server on internet. Many other machine learning approaches for chess are discussed in [15, 16].

Input features. To characterize a chess position we convert it into some important features. An example of such a feature is connectivity. The connectivity of a piece is the amount of pieces it defends. In figure 5 the total connectivity of the white pieces is 7.

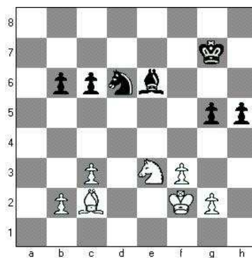

Figure 5: Connectivity is one feature for evaluating chess positions. In the given board-position, the connectivity of white's pieces is 7 and of black's pieces it is 0.

The pawn on b2 defends the pawn on c3. The pawn on g2 defends the pawn on f3. The knight on e3 defends the pawn on g2 and the bishop on c2. The king on f2 defends the pawn on g2, the pawn on f3 and the knight on e3. There is no connectedness between the black pieces.

We use the following general features: number of queens, rooks, bishops, knights, pawns for both players, material balance, Queen's mobility, Rook's horizontal mobility, Rook's vertical mobility, Bishop's mobility, Knight's mobility, Center control, Isolated pawns, Doubled pawns, Passed pawns, Pawn forks, Knight forks, Light pieces on first rank, Horizontally connected rooks, Vertically connected rooks, Rooks on seventh rank, Board control, Connectivity, King's distance to center. A more extensive description of the features we used in our experiments can be found in [20].

Parameters. In our experiments we made use of the open source chess program *tscp 1.81*[2] which was written by Tom Kerrigan in C. The parameters of the networks are: learning rate = 0.001, $\lambda = 0.9$, $\gamma = 1.0$, the number of hidden units is 80 and they use the sigmoid activation function. The networks have a single output unit with a linear activation function and are trained with normal backpropagation.

Architectures. For our chess-experiment we experimented with seven different neural network architectures:

A = a network with general features (71 inputs)

B = 3 separated networks with general features (71 inputs)

C = a network with general features and partial raw board features (kings and pawns, 311 inputs)

[2]Tscp 1.81 can be downloaded from: http://home.comcast.net/~tckerrigan

D = 3 separated networks with general features and partial raw board features (kings and pawns, 311 inputs)

E = a network with general features and full raw board features (831 inputs)

F = 3 separated networks with general features and full raw board features (831 inputs)

G = a linear network with general features and partial raw board features (311 inputs)

H = a hand-coded evaluation function which is the a-priori given evaluation function of tscp 1.81

All networks were trained a single time on 50,000 different database games. The linear network was a network without a hidden layer. The full raw board features contained information about the position of all pieces. The partial raw board features only informed about the position of kings and pawns. The separated networks consisted of three networks. They had a different network for positions in the opening, middle-game and endgame. Discriminating among these three stages of a game was done by looking at the amount of material present. Positions with a material value greater than 65 points are classified as opening positions. Middle-game positions have a value between 35 and 65 points. All other positions are labelled as endgame positions.

The hand-coded evaluation function sums up the scores for similar features as the general features of the trained evaluation functions. A tournament was held in which every trained network architecture played 5 games with the white pieces and 5 games with the black pieces against the other architectures. The search depth of all programs was set to 2 ply. Programs only searched deeper if a side was in check in the final position or if a piece was captured. This was done to avoid the overseeing of short tactical tricks.

Table 5: Performance of the different trained architectures

Rank	Program	Separated networks	Inputs	won-lost	Score
1	**D**	Yes	General, kings and pawns	51,5-18,5	+33
2	**C**	No	General, kings and pawns	42-28	+14
3	**B**	Yes	General	39,5-30,5	+9
4	**A**	No	General	39-31	+8
5	**G**	No	General, kings and pawns	37,5-32,5	+5
6	**H**	No	General tcsp 1.81	34,5-35,5	-1
7	**F**	Yes	General and full board	20,5-49,5	-29
8	**E**	No	General and full board	15,5-54,5	-39

Experimental results. The results are reported in Table 5. The three possible results of a game are: 1-0(win), 0-1(loss) and 0,5-0,5(draw). The best results were obtained by the separated networks with general features and partial raw board information. The single network with general features and partial raw board also performed well, but its 'big brother' yielded a much higher score. This is because it is hard to generalize over the position of kings and pawns during the different stages of the game (opening, middle-game and endgame). During the opening and middle-game the king often seeks protection in a corner behind its pawns. While in the endgame the king can become a strong piece, often marching on to the center of the board. Pawns also are moved further in endgame positions than they are in the opening and middle-game. Because the separated networks are trained on the different stages of the game, they are more capable of making this positional distinction.

The networks with general features (**A** and **B**) also yielded a positive result. They lacked knowledge of the exact position of the kings and pawns on the board. Therefore awkward looking pawn and king moves were sometimes made in their games.

Table 6: Results against hand-coded evaluation function

	A	**B**	**C**	**D**	**E**	**F**	**G**
H	5,5-4,5	4-6	4-6	3-7	7-3	7,5-2,5	3,5-6,5

The linear network made a nice result, but because it is just a linear function it is not able to learn some important non-linear characteristics of board positions. Its result was better than the result obtained by the hand-coded evaluation function, which scored slightly less than 50%.

The networks with the greatest amount of input features (**E** and **F**) scored not very well. This is because they have to be trained on much more games before they can generalize well on the input they get from the positions of all pieces.

The separated networks yielded a better result than their single network counterparts. This can be explained by the fact that the separated networks version was better in the positioning of its pieces during the different stages of the game. For instance, the single network often put its queen in play too soon. During the opening it is normally not very wise to move a queen to the center of the board. The queen may have a high mobility score in the center, but it often gives the opponent the possibility for rapid development of its pieces by attacking the queen.

In Table 6 we can see that four of the seven learned evaluation functions played better chess than the hand-coded evaluation function after just 50,000 training games. The linear evaluation function defeated the hand-coded evaluation function so we may conclude that the hand-coded function used inferior features or weighted them in a worse manner. Four non-linear functions booked better results than this linear function (see table 5). It took 7 hours to train evaluation function **D** on a PentiumII 450mhz. This illustrates the attractiveness of database training in order to create a quite good evaluation function in a short time interval. If we realize that a usual human game costs 4 hours to play, a game of that quality may also consume so much time by a computer, and therefore learning from databases is much faster. Note that learning from a single game costs 420 minutes divided by 50,000 which is about 0,01 minute. If we would have used lookahead for learning by self-play, one game would have cost at least 5 minutes, so that only about 80 games could have been played in the same 7 hours.

5.2 Learning to Play Draughts

In international draughts several grand-masters were defeated by the programs Buggy[2003] and Flits[2002]. The success of most of the strongest programs in draughts is due to the use of the possibility to look upon many thousands of positions per second by using computer power and efficient search algorithms. In most of the board games the evaluation function is tuned by the system designer and this may cost a lot of time. Machine learning can result in better evaluation functions in less time.

It should be noted that with draughts we mean international draughts played on a 10x10 squared board, which is more complicated than the game of checkers for which very good programs exist. In fact, the checkers program Chinook was the first to win a human world championship in any game [32]. The early work of Samuel (1959, 1967) showed the first approach for learning a game evaluation function. For this Samuel used a kind of temporal difference learning approach to evolve a checkers playing program. The resulting program was the first computer game playing program which was able to beat its programmer.

Learning a game evaluation function for checkers was also done by Lynch and Griffith (1997). Their program Neurodraughts used a cloning strategy to test, select and train networks with different input features. It is not known how strong the best resulting

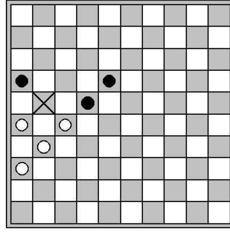

Figure 6: The structural feature 'Hekstelling'. The given pieces should be on the indicated fields and on the field with a cross there should not be any piece.

program plays against other checkers playing programs or humans.

Experimental setup. The games used in training are extracted from Turbo Dambase[3], the only draughts game-database. This database contains more than 260.000 games between players of all different levels. For testing purposes the neural network was incorporated in the draughts program Buggy[4] which is probably the best draughts program of the world and uses state-of-the-art search algorithms. Because it is almost impossible to learn from the raw board position alone, we represented the board position by additional features.

Input features. We used 3 different kinds of features namely global-features, structural features and raw board information. In the raw board representation every field of the game position is decoded to represent the presence of the kind of piece. Every square is represented by two bits. One bit to represent the occupancy of a white single checker and one for a single black checker. We did not use raw-board presentation for kings. Structural features are boolean combinations of occupied or empty squares. An example of a structural feature is 'Hekstelling' (translated 'Fence position') shown in Figure 6. Unlike structural features and the raw-board representation, global features model knowledge which is not directly available from a board position. An example of a global-feature is material balance which is the difference in checkers plus 3 times the difference in kings.

The total amount of inputs, which are 23 global plus 97 structural features plus 100 raw board features, is 220 (for an extensive description of the used input features, see [23]). Five neural networks were trained. The difference between the first four neural networks lies only in the representation of a board position. We define the four neural network players in respect to the input, namely:

- NN1: Global features (23 inputs)
- NN2: Global features plus all possible 'three-on-a-diagonal' (87 inputs)
- NN3: All features except the raw board representation (120 inputs)
- NN4: All features (220 inputs)

The fifth neural network consists of three separated neural networks. The neural networks were used in training or playing on the basis how many pieces were on the board. One for the position with more than 25 pieces, one for more than 15 till 25 pieces and one for the remaining positions, for which all three were equipped with all features. All the neural networks were trained a single time over the first 200.000 games of Turbo Dambase. Positions in which a player was obliged to hit were eliminated from the game. Also all positions were eliminated which are a part of a tactical combination or a local discontinuity in material balance. This is for example when a player successively sacrifices two checkers in two moves and regains material balance on the third move by hitting two checkers.

[3]For information on Turbo Dambase, see http://www.turbodambase.com/

[4]For information on Buggy, see http://www.buggy-online.com/

Table 7: *The result of the matches against the program GWD. A win is awarded with 2 points, a draw with 1 point.*

program:	*GWD*				
NN-...	Inputs	WIN	LOSS	DRAW	RESULT
NN1	Global features	3	4	3	9-11
NN2	Global and 3-on-a-diagonal	4	1	5	13-7
NN3	All features except raw board info	7	3	0	14-6
NN4	All features	8	1	1	17-3

The reason why we did this is because learning on these positions distorted the learning process. The neural network has great difficulty in learning specific board settings in order to compensate for the temporal difference in material.

Learning parameters. The number of hidden units is 80, the learning rate for NN1 and NN2 is 0.001, the learning rate for NN3, NN4, and NN5 is 0.0005. We set $\lambda = 0.9$. In our experiments we used the activation function $\beta x/(1 + abs(\beta x))$ with neuron sensitivity β. The neuron sensitivity is adjusted according to the partial derivative of the error to the neuron sensitivity using extended back-propagation [37, 38] which is used to speed up learning. The initial hidden neuron sensitivity β is 3.0. The learning rate of the sensitivity for hidden neurons for NN1 and NN2 is 0.01, and the learning rate of the sensitivity for hidden neurons for NN3, NN4, and NN5 is 0.005.

Testing. The four neural networks, created by learning from 200.000 database games, were tested against two draughts programs available on the internet. DAM 2.2[5] is a very strong player and GWD[6] a strong player on the scale of very weak, weak, medium, strong and very strong.[7] Each network played one match of 10 games, 5 with white and 5 with black against both computer programs. In all the games both players got 4 minutes time. In the programs GWD and DAM 2.2. it was only possible to give the computer the amount of time per move. So GWD and DAM 2.2 got 4 seconds per move. Furthermore a round tournament was held with only the neural network players. So each player played 10 games against another player using 4 minutes per game per player. No program made use of an opening book.

Experimental results. The results of matches against the strong program GWD are shown in Table 7. All neural networks, except NN1 were able to beat GWD after training from 200,000 database games. We can see that NN4 which uses the largest amount of features performs best against GWD and is able to beat this program 8 times out of 10 test-games. Therefore it can be concluded that the learned programs play quite strong.

The results of the trained architectures against the very strong computer program DAM 2.2 are shown in Table 8. The results show that DAM 2.2 is still much stronger than the learned evaluation functions, although it is not able to always win against them. Since the programs obtained the same amount of time (only about 4 seconds per move), we expect that DAM 2.2 could make deeper lookahead searches due to its faster evaluation function. The learning programs use a large neural network which takes much longer to evaluate. Therefore it was not possible to lookahead the same number of moves. This is a drawback when the search is not very deep, e.g. a lookahead search of 3 or 5 ply can make a huge difference. If the computers would be much faster, a lookahead search of 12 compared to 14 ply would be much less important, and thus more knowledge in the

[5]For information and download, see http://www.xs4all.nl/ hjetten/dameng.html

[6]For information and download, see http://www.wxs.nl/ gijsbert.wiesenekker/gwd4distrib.zip

[7]According to http://perso.wanadoo.fr/alemanni/apage76_e.html

Table 8: The result of the matches against the program DAM 2.2. A win is awarded with 2 points, a draw with 1 point.

program:		*DAM 2.2*		
NN-...	WIN	LOSS	DRAW	RESULT
NN1	0	9	1	1-19
NN2	0	9	1	1-19
NN3	0	7	3	3-17
NN4	0	9	1	1-19

Table 9: The results of the round tournament between the neural networks. The outcome 3-3-4 means the row player won 3 games, lost 3 games and drew 4 games.

x	NN1	NN2	NN3	NN4
NN1	x	3-3-4	2-6-2	4-6-0
NN2	3-3-4	x	1-6-3	3-5-2
NN3	6-2-2	6-1-3	x	4-5-1
NN4	6-4-0	5-3-2	5-4-1	x

evaluation function could become more profitable. Unfortunately it still takes a lot of time until computers can lookahead many moves using large and knowledge-rich evaluation functions, and thus it is currently difficult to measure the utility of learned evaluation functions for obtaining world class level play.

In Table 9, the results of the round tournament of the network architectures is shown. It is clear that NN3 and NN4 are stronger than NN1 and NN2, and that more input features is therefore useful for learning good evaluation functions. The networks were trained over a whole game. However because an endgame has nothing to do with the middle-game inference can be a problem. We also tried an experiment with 3 neural networks for the opening, the middle-game and endgame. However the network was not able to learn the material function in the opening. The reason is that there are too few examples of positions in the opening with piece (dis)advantage in the database games.

Discussion. The networks NN2, NN3 and NN4 were able to defeat GWD on a regular basis. Most of the time NN1 had a superb position against GWD. In many of the games the advantage in development for NN1 was huge and NN1 was controlling all the center fields. However sometimes NN1 made it possible for GWD to get a clear way to promotion and sometimes built uncomfortable formations. The reason why it did this is because it does not have enough input features for accurately describing the position. All networks were not able to compete against Dam 2.2. Some of the networks were able to draw sometimes. NN3 and NN4 are much better playing networks than NN1 and NN2. A problem of NN3 and NN4 is the evaluation of positions with structures where break off and rebuilding of the structure plays an important role. Probably this is because of the input features and the lack of an opening book. The networks NN3 and NN4 only have one structural feature for these kind of positions. That is why it is very difficult to learn using these features whether the positions are good or bad. It is to be noted that this is of course when lookahead does not reach further than the disappearance of the structure.

Probably the biggest problem in the play is the opening. In some openings the networks want to go for a quick attacking strategy, but in the opening this is most of the time

not a good idea. One simple counter strategy is to attack this piece a couple of times and exchange it. This results immediately in disadvantage in development and in an unconnected (split) position. The problem of the opening is that the positions are the most far positions from the end position. Because of this, the variance in the value of the positions is very small, but later some small differences on the board can lead to a considerable advantage. The opening has been a problem too for other draughts playing computers which therefore often use an opening book.

5.3 Discussion

The experiments with learning to play chess and draughts using database games indicated that quite good evaluation functions can be obtained within a short time. To enhance the evaluation functions, we can let the architectures train multiple times from the database games. Although it may be useful to switch to learning by self-play at some moment in time so that the learning program can generate experiences with moves never played in the database games, generating games by self-play would cost much more time. Especially if we want to have a large neural network architecture that could learn a lot of game knowledge, many games need to be played and therefore initially learning from given games may be a very good method. It should be mentioned that Baxter et al.'s Knightcap [2] was able to learn to play chess at human expert level from only about 300 games played on internet, but it used a linear network and its initial playing strength was already much better than random.

The experiments indicated that using more features and therefore larger architectures can work better, but a problem of having a large neural network architecture as evaluation function is that it takes much more time to evaluate a position than with a set of symbolic rules or a linear network. E.g., we used 80 hidden units and therefore the evaluation time is roughly 80 times higher. This means that search depth in the same time interval is at least 3 ply less which is a significant drawback if we cannot use very fast computers which obtain a search depth of more than 10 ply anyway. Therefore, we expect that learning game evaluation functions for difficult games requiring a lot of lookahead is most fruitful if faster computers become available.

The experiments also showed that raw board information is useful for learning better game evaluation functions. Although the architectures become larger and therefore more training games need to be observed, the raw board inputs give useful information which cannot be obtained solely from higher level features. For chess separate networks for the opening, middle-game and endgame resulted in better evaluation functions, but for draughts this was not the case. The problem with draughts was that there were too few examples of differences in material in the opening game. This could be easily overcome by adding more games played by worse players or by sometimes allowing the learning programs to learn from self-play. We could improve the results by adding more higher-level features, modules, and increasing the number of training games. Furthermore, tournaments can be held to store the best learned evaluation function.

6 Conclusion

In this paper we looked at the advantages of using database games for learning to play games using temporal difference methods. The results indicated that this approach has as large advantage that the learning program can train on much more games than using self-play without a large penalty for doing so. The other possible advantage, namely that games are initially played at a high level was not clearly shown in the experimental results. In fact, learning form observations seems to require more training games than learning by self-play or playing against an expert. We only studied a single machine learning ap-

proach, however, namely temporal difference methods. If we looked at the games played by the trained programs, we observed that opening and endgame play was not always optimal. In the opening and endgame the evaluations of positions often do not differ very much (e.g., in the endgame there can be many winning positions, but this does not lead to an optimal strategy for winning the game). Since the evaluations are so close, a function approximator has large difficulties in preferring the best moves. We believe therefore that the opening or endgame should be learned by other machine learning approaches such as case-based reasoning or root learning. During the middle-game, the number of positions is usually largest and therefore it is infeasible to store all positions. We therefore think that learning evaluation functions for middle-game positions using temporal difference learning methods is very helpful. Other possibilities to improve the level of games consist of learning search control, learning to add higher-level features automatically, or learning different strategical position classes. Using a symbolic categorization module has as advantage than we can use more parameters for learning the evaluation function, whereas we still have fast propagation. We believe that combining a variety of machine learning methods and using a large number of database games will make it possible to learn world class playing programs for a large number of games.

Learning from databases can also be used for other applications, such as learning in action or strategical 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 using a joystick and these experiences can be recorded and then learned by the robot [36]. Thus, we think that learning from observing an expert has many advantages and possibilities for learning control knowledge, and can be more fruitfully used for many applications than learning from trial and error, whereas the same reinforcement learning algorithms can still be applied.

References

[1] 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*, SMC-13:834–846, 1983.

[2] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: A chess program that learns by combining TD(λ) with minimax search. Technical report, Australian National University, Canberra, 1997.

[3] J. Baxter, A. Tridgell, and L. Weaver. A chess program that learns by combining td(lambda) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning*, pages 28–36, 1998.

[4] J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

[5] D. Beal. Learning piece values using temporal differences. *International Computer Chess Association Journal*, 20:147–151, 1997.

[6] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[7] H. Berliner. Experiences in evaluation with BKG - a program that plays backgammon. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 428–433, 1977.

[8] H. Berliner. Search vs. knowledge: An analysis from the domain of games. In A. Elithorn and R. Banerji, editors, *Artificial and Human Intelligence.* Elsevier, New York, 1984.

[9] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University, New York, 1995.

[10] J. A. Boyan. Modular neural networks for learning context-dependent game strategies. Master's thesis, University of Chicago, 1992.

[11] 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, editors, *Advances in Neural Information Processing Systems 9*. Morgan Kaufmann, San Mateo, CA, 1997.

[12] A. de Groot. *Thought and Choice in Chess*. Mouton & Co, The Hague, The Netherlands, 1965.

[13] E.D. de Jong and J.B. Pollack. Ideal evaluation from coevolution. *Evolutionary Computation*, 12(2):159–192, 2004.

[14] D.B. Fogel. Evolving a checkers player without relying on human experience. *Intelligence*, 11(2):217, 2000.

[15] J. Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–160, 1996.

[16] J. Fürnkranz. Machine learning in games: A survey. In *Machines that learn to Play Games, chapter 2*, pages 11–59. Nova Science Publishers, Huntington, NY, 2001.

[17] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[18] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[19] L-J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

[20] H. Mannen. *Learning to play chess using reinforcement learning with database games*, 2003. Master's thesis, Cognitive Artificial Intelligence, Utrecht University.

[21] T.M. Mitchell and S. Thrun. Explanation based learning: A comparison of symbolic and neural network approaches. In *Proceddings of the tenth International Conference on Machine Learning*, pages 197–204, 1993.

[22] D.E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.* PhD thesis, Department of Computer Sciences, The university of Texas at Austin, 1997.

[23] J-P. Patist. *Learning to play draughts using reinforcement learning with neural networks*, 2003. Master's thesis, Cognitive Artificial Intelligence, Utrecht University.

[24] A. Plaat. *Research Re:search and Re-search.* PhD thesis, Erasmus University Rotterdam, 1996.

[25] J.B. Pollack and A.D. Blair. Why did TD-Gammon work. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 10–16, Cambridge MA, 1996. MIT Press.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

[27] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist sytems. Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK, 1994.

[28] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3(3):210–229, 1959.

[29] A. L. Samuel. Some studies in machine learning using the game of checkers II - recent progress. *IBM Journal on Research and Development*, 11(6):601–617, 1967.

[30] J. Schaeffer. The games computers (and people) play. In *Advances in Computers*, volume 50, pages 189–266. Academic Press, 2000.

[31] J. Schaeffer, M. Hlynka, and V. Hussila. Temporal difference learning applied to a high-performance game. In *Seventeenth International Joint Conference on Artificial Intelligence*, pages 529–534, 2001.

[32] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The world man-machine checkers champion. *Artificial Intelligence Magazine*, 17(1):21–29, 1996.

[33] J. Schaeffer and A. Plaat. Kasparov versus deep blue: The re-match. *International Computer Chess Association Journal*, 20(2):95–102, 1997.

[34] N. N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann, San Francisco, 1994.

[35] S.P. Singh, T. Jaakkola, M.L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[36] W. Smart and L. Kaelbling. Effective reinforcement learning for mobile robots, 2002.

[37] A. Sperduti. Speed up learning and network optimization with extended back propagation. Technical Report TR-10/92, University of Pisa, 1992.

[38] A. Sperduti and A. Starita. Speed up learning and network optimization with extended back propagation. *Neural Networks*, 6:365–383, 1993.

[39] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[40] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1045. MIT Press, Cambridge MA, 1996.

[41] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT press, Cambridge MA, A Bradford Book, 1998.

[42] C. Szepesvari and M.L. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 11(8):2017–2059, 1999.

[43] G. Tesauro. Practical issues in temporal difference learning. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 4*, pages 259–266. San Mateo, CA: Morgan Kaufmann, 1992.

[44] G.J. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[45] S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie-Mellon University, January 1992.

[46] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. San Fransisco, CA: Morgan Kaufmann, 1995.

[47] J. N Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202, 1994.

[48] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, England, 1989.

[49] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[50] M. A. Wiering and J. H. Schmidhuber. Fast online Q(λ). *Machine Learning*, 33(1):105–116, 1998.