Machine Learning, 40, 243–263, 2000

Learning to Play Chess Using Temporal Differences

JONATHAN BAXTER jonathan.baxter@anu.edu.au
Department of Systems Engineering, Australian National University 0200, Australia

ANDREW TRIDGELL andrew.tridgell@cs.anu.edu.au
LEX WEAVER lex.weaver@cs.anu.edu.au
Department of Computer Science, Australian National University 0200, Australia

Editor: Sridhar Mahadevan

Abstract. In this paper we present TDLEAF(λ), a variation on the TD(λ) algorithm that enables it to be used in conjunction with game-tree search. We present some experiments in which our chess program "KnightCap" used TDLEAF(λ) to learn its evaluation function while playing on Internet chess servers. The main success we report is that KnightCap improved from a 1650 rating to a 2150 rating in just 308 games and 3 days of play. As a reference, a rating of 1650 corresponds to about level B human play (on a scale from E (1000) to A (1800)), while 2150 is human master level. We discuss some of the reasons for this success, principle among them being the use of on-line, rather than self-play. We also investigate whether TDLEAF(λ) can yield better results in the domain of backgammon, where TD(λ) has previously yielded striking success.

Keywords: temporal difference learning, neural network, TDLEAF, chess, backgammon

1. Introduction

Temporal Difference learning, first introduced by Samuel (Samuel, 1959) and later extended and formalized by Sutton (Sutton, 1988) in his TD(λ) algorithm, is an elegant technique for approximating the expected long term future cost (or *cost-to-go*) of a stochastic dynamical system as a function of the current state. The mapping from states to future cost is implemented by a parameterized function approximator such as a neural network. The parameters are updated online after each state transition, or possibly in batch updates after several state transitions. The goal of the algorithm is to improve the cost estimates as the number of observed state transitions and associated costs increases.

Perhaps the most remarkable success of TD(λ) is Tesauro's TD-Gammon, a neural network backgammon player that was trained from scratch using TD(λ) and simulated self-play. TD-Gammon is competitive with the best human backgammon players (Tesauro, 1994). In TD-Gammon the neural network played a dual role, both as a predictor of the expected cost-to-go of the position and as a means to select moves. In any position the next move was chosen greedily by evaluating all positions reachable from the current state, and then selecting the move leading to the position with smallest expected cost. The parameters of the neural network were updated according to the TD(λ) algorithm after each game.

Although the results with backgammon are quite striking, there is lingering disappointment that despite several attempts, they have not been repeated for other board games such

as othello, Go and the "drosophila of AI"—chess (Thrun, 1995; Walker, Lister, & Down, 1993; Schraudolph, Dayan, & Sejnowski, 1994).

Many authors have discussed the peculiarities of backgammon that make it particularly suitable for Temporal Difference learning with self-play (Tesauro, 1992; Schraudolph, Dayan, & Sejnowski, 1994; Pollack, Blair, & Land, 1996). Principle among these are *speed of play*: TD-Gammon learnt from several hundred thousand games of self-play, *representation smoothness*: the evaluation of a backgammon position is a reasonably smooth function of the position (viewed, say, as a vector of piece counts), making it easier to find a good neural network approximation, and *stochasticity*: backgammon is a random game which forces at least a minimal amount of exploration of search space.

As TD-Gammon in its original form only searched one-ply ahead, we feel this list should be appended with: *shallow search is good enough against humans*. There are two possible reasons for this; either one does not gain a lot by searching deeper in backgammon (questionable given that recent versions of TD-Gammon search to three-ply and this significantly improves their performance), or humans are simply incapable of searching deeply and so TD-Gammon is only competing in a pool of shallow searchers. Although we know of no psychological studies investigating the depth to which humans search in backgammon, it is plausible that the combination of high branching factor and random move generation makes it quite difficult to search more than one or two-ply ahead. In particular, random move generation effectively prevents selective search or "forward pruning" because it enforces a lower bound on the branching factor at each move.

In contrast, finding a representation for chess, othello or Go which allows a small neural network to order moves at one-ply with near human performance is a far more difficult task. It seems that for these games, reliable tactical evaluation is difficult to achieve without deep lookahead. As deep lookahead invariably involves some kind of minimax search, which in turn requires an exponential increase in the number of positions evaluated as the search depth increases, the computational cost of the evaluation function has to be low, ruling out the use of expensive evaluation functions such as neural networks. Consequently most chess and othello programs use linear evaluation functions (the branching factor in Go makes minimax search to any significant depth nearly infeasible).

Our goal is to develop techniques for using TD(λ) in domains dominated by search. In this paper we introduce TDLEAF(λ), a variation on the TD(λ) algorithm, that can be used to learn an evaluation function for use in deep minimax search. TDLEAF(λ) differs from TD(λ) in that instead of operating on positions that occur during the game, it operates on the leaf nodes of the *principal variation* of a minimax search from each position (also known as the *principal leaves*).

To test the effectiveness of TDLEAF(λ), we incorporated it into our own chess program—*KnightCap*. KnightCap has a particularly rich board representation facilitating computation of sophisticated positional features, although this is achieved at some cost in speed (KnightCap is about 10 times slower than *Crafty*—the best public-domain chess program—and 6,000 times slower than *Deep Blue*). We trained KnightCap's linear evaluation function using TDLEAF(λ) by playing it on the Free Internet Chess Server (FICS, `fics.onenet.net`) and on the Internet Chess Club (ICC, `chessclub.com`). Internet play was used to avoid the premature convergence difficulties associated with self-play. The main success story we

report is that starting from an evaluation function in which all coefficients were set to zero except the values of the pieces, KnightCap went from a 1650-rated player to a 2150-rated player in just three days and 308 games. KnightCap is an ongoing project with new features being added to its evaluation function all the time. We use TDLEAF(λ) and Internet play to tune the coefficients of these features.

Simultaneously with the work presented here, Beal and Smith (Beal & Smith, 1997) reported positive results using essentially TDLEAF(λ) and self-play (with some random move choice) when learning the parameters of an evaluation function that only computed material balance in chess. However, they were not comparing performance against on-line players, but were primarily investigating whether the weights would converge to "sensible" values at least as good as the naive (1, 3, 3, 5, 9) values for (pawn, knight, bishop, rook, queen) (they did, in about 2000 games).

Sutton and Barto (Sutton and Barto, 1998) have outlined, but not implemented, a scheme for combining TD-style backups with deep minimax search. Their method would calculate all the one-step differences seen during the construction of the search tree.

The remainder of this paper is organized as follows. In Section 2 we describe the TD(λ) algorithm as it applies to games. The TDLEAF(λ) algorithm is described in Section 3. Experimental results for Internet-play with KnightCap are given in Section 4, while Section 5 looks at applying TDLEAF(λ) to backgammon where TD(λ) has had its greatest success. Section 7 contains some discussion and concluding remarks.

2. The TD(λ) algorithm applied to games

In this section we describe the TD(λ) algorithm as it applies to playing board games. We discuss the algorithm from the point of view of an *agent* playing the game.

Let S denote the set of all possible board positions in the game. Play proceeds in a series of moves at discrete time steps $t = 1, 2, \ldots$. At time t the agent finds itself in some position $x_t \in S$, and has available a set of moves, or *actions* A_{x_t} (the legal moves in position x_t). The agent chooses an action $a \in A_{x_t}$ and makes a transition to state x_{t+1} with probability $p(x_t, x_{t+1}, a)$. Here x_{t+1} is the position of the board after the agent's move and the opponent's response. When the game is over, the agent receives a scalar reward, typically "1" for a win, "0" for a draw and "-1" for a loss.

For ease of notation we will assume all games have a fixed length of N (this is not essential). Let $r(x_N)$ denote the reward received at the end of the game. If we assume that the agent chooses its actions according to some function $a(x)$ of the current state x (so that $a(x) \in A_x$), the expected reward from each state $x \in S$ is given by

$$J^*(x) := E_{x_N|x} r(x_N), \tag{1}$$

where the expectation is with respect to the transition probabilities $p(x_t, x_{t+1}, a(x_t))$ and possibly also with respect to the actions $a(x_t)$ if the agent chooses its actions stochastically.

For very large state spaces S it is not possible store the value of $J^*(x)$ for every $x \in S$, so instead we might try to approximate J^* using a parameterized function class $\tilde{J}: S \times \mathbb{R}^k \to \mathbb{R}$, for example linear functions, splines, neural networks, etc. $\tilde{J}(\cdot, w)$ is assumed to be a

differentiable function of its parameters $w = (w_1, \ldots, w_k)$. The aim is to find a parameter vector $w \in \mathbb{R}^k$ that minimizes some measure of error between the approximation $\tilde{J}(\cdot, w)$ and $J^*(\cdot)$. The TD(λ) algorithm, which we describe now, is designed to do exactly that.

Suppose $x_1, \ldots, x_{N-1}, x_N$ is a sequence of states in one game. For a given parameter vector w, define the *temporal difference* associated with the transition $x_t \to x_{t+1}$ by

$$d_t := \tilde{J}(x_{t+1}, w) - \tilde{J}(x_t, w). \tag{2}$$

Note that d_t measures the difference between the reward predicted by $\tilde{J}(\cdot, w)$ at time $t+1$, and the reward predicted by $\tilde{J}(\cdot, w)$ at time t. The true evaluation function J^* has the property

$$E_{x_{t+1}|x_t}[J^*(x_{t+1}) - J^*(x_t)] = 0,$$

so if $\tilde{J}(\cdot, w)$ is a good approximation to J^*, $E_{x_{t+1}|x_t} d_t$ should be close to zero. For ease of notation we will assume that $\tilde{J}(x_N, w) = r(x_N)$ always, so that the final temporal difference satisfies

$$d_{N-1} = \tilde{J}(x_N, w) - \tilde{J}(x_{N-1}, w) = r(x_N) - \tilde{J}(x_{N-1}, w).$$

That is, d_{N-1} is the difference between the true outcome of the game and the prediction at the penultimate move.

At the end of the game, the TD(λ) algorithm updates the parameter vector w according to the formula

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}(x_t, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right] \tag{3}$$

where $\nabla \tilde{J}(\cdot, w)$ is the vector of partial derivatives of $\tilde{J}$ with respect to its parameters. The positive parameter α controls the learning rate and would typically be "annealed" towards zero during the course of a long series of games. The parameter $\lambda \in [0, 1]$ controls the extent to which temporal differences propagate backwards in time. To see this, compare Eq. (3) for $\lambda = 0$:

$$\begin{aligned} w &:= w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}(x_t, w) d_t \\ &= w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}(x_t, w) [\tilde{J}(x_{t+1}, w) - \tilde{J}(x_t, w)] \end{aligned} \tag{4}$$

and $\lambda = 1$:

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}(x_t, w) [r(x_N) - \tilde{J}(x_t, w)]. \tag{5}$$

Consider each term contributing to the sums in Eqs. (4) and (5). For $\lambda = 0$ the parameter vector is being adjusted in such a way as to move $\tilde{J}(x_t, w)$—the predicted reward at time t—closer to $\tilde{J}(x_{t+1}, w)$—the predicted reward at time $t + 1$. In contrast, TD(1) adjusts the parameter vector in such away as to move the predicted reward at time step t closer to the final reward at time step N. Values of λ between zero and one interpolate between these two behaviours. Note that (5) is equivalent to gradient descent on the error function $E(w) := \sum_{t=1}^{N-1}[r(x_N) - \tilde{J}(x_t, w)]^2$.

Successive parameter updates according to the TD(λ) algorithm should, over time, lead to improved predictions of the expected reward $\tilde{J}(\cdot, w)$. Provided the actions $a(x_t)$ are independent of the parameter vector w, it can be shown that for *linear* $\tilde{J}(\cdot, w)$, the TD(λ) algorithm converges to a near-optimal parameter vector (Tsitsiklis & Roy, 1997). Unfortunately, there is no such guarantee if $\tilde{J}(\cdot, w)$ is non-linear (Tsitsiklis & Roy, 1997), or if $a(x_t)$ depends on w (Bertsekas & Tsitsiklis, 1996).

3. Minimax search and TD(λ)

For argument's sake, assume any action a taken in state x leads to predetermined state which we will denote by x'_a. Once an approximation $\tilde{J}(\cdot, w)$ to J^* has been found, we can use it to choose actions in state x by picking the action $a \in A_x$ whose successor state x'_a minimizes the opponent's expected reward:[1]

$$a^*(x) := \arg\min_{a \in A_x} \tilde{J}(x'_a, w). \tag{6}$$

This was the strategy used in TD-Gammon. Unfortunately, for games like othello and chess it is very difficult to accurately evaluate a position by looking only one move or *ply* ahead. Most programs for these games employ some form of *minimax* search. In minimax search, one builds a tree from position x by examining all possible moves for the computer in that position, then all possible moves for the opponent, and then all possible moves for the computer and so on to some predetermined depth d. The leaf nodes of the tree are then evaluated using a heuristic evaluation function (such as $\tilde{J}(\cdot, w)$), and the resulting scores are propagated back up the tree by choosing at each stage the move which leads to the best position for the player on the move. See figure 1 for an example game tree and its minimax evaluation. With reference to the figure, note that the evaluation assigned to the root node is the evaluation of the leaf node of the *principal variation*; the sequence of moves taken from the root to the leaf if each side chooses the best available move.

In practice many engineering tricks are used to improve the performance of the minimax algorithm, $\alpha - \beta$ search being the most famous.

Let $\tilde{J}_d(x, w)$ denote the evaluation obtained for state x by applying $\tilde{J}(\cdot, w)$ to the leaf nodes of a depth d minimax search from x. Our aim is to find a parameter vector w such that $\tilde{J}_d(\cdot, w)$ is a good approximation to the expected reward J^*. One way to achieve this is to apply the TD(λ) algorithm to $\tilde{J}_d(x, w)$. That is, for each sequence of positions $x_1, \ldots, x_N$ in a game we define the temporal differences

$$d_t := \tilde{J}_d(x_{t+1}, w) - \tilde{J}_d(x_t, w) \tag{7}$$

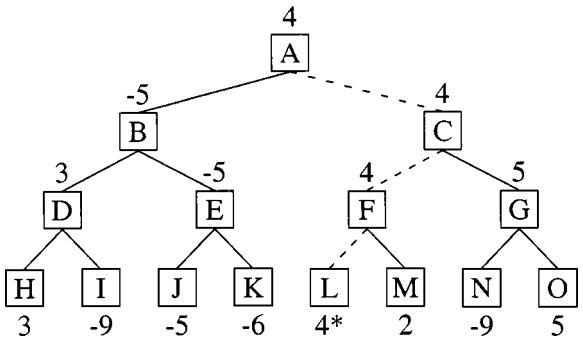

Figure 1. Full breadth, 3-ply search tree illustrating the minimax rule for propagating values. Each of the leaf nodes (H–O) is given a score by the evaluation function, $\tilde{J}(\cdot, w)$. These scores are then propagated back up the tree by assigning to each opponent's internal node the minimum of its children's values, and to each of our internal nodes the maximum of its children's values. The principle variation is then the sequence of best moves for either side starting from the root node, and this is illustrated by a dashed line in the figure. Note that the score at the root node A is the evaluation of the leaf node (L) of the principal variation. As there are no ties between any siblings, the derivative of A's score with respect to the parameters w is just $\nabla \tilde{J}(L, w)$.

as per Eq. (2), and then the TD(λ) algorithm (3) for updating the parameter vector w becomes

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}_d(x_t, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right]. \tag{8}$$

One problem with Eq. (8) is that for $d > 1$, $\tilde{J}_d(x, w)$ is not necessarily a differentiable function of w for all values of w, even if $\tilde{J}(\cdot, w)$ is everywhere differentiable. This is because for some values of w there will be "ties" in the minimax search, i.e. there will be more than one best move available in some of the positions along the principal variation, which means that the principal variation will not be unique (see figure 2). Thus, the evaluation assigned to the root node, $\tilde{J}_d(x, w)$, will be the evaluation of any one of a number of leaf nodes.

Fortunately, under some mild technical assumptions on the behaviour of $\tilde{J}(x, w)$, it can be shown that for each state x, the set of $w \in \mathbb{R}^k$ for which $\tilde{J}_d(x, w)$ is not differentiable has Lebesgue measure zero. Thus for all states x and for "almost all" $w \in \mathbb{R}^k$, $\tilde{J}_d(x, w)$ is a differentiable function of w. Note that $\tilde{J}_d(x, w)$ is also a continuous function of w whenever $\tilde{J}(x, w)$ is a continuous function of w. This implies that even for the "bad" pairs (x, w), $\nabla \tilde{J}_d(x, w)$ is only undefined because it is multi-valued. Thus we can still arbitrarily choose a particular value for $\nabla \tilde{J}_d(x, w)$ if w happens to land on one of the bad points. One final point to note is that as we search deeper, discontinuities in the gradient are likely to become more dense and so most steps in parameter space are likely to step clear across several discontinuities. However, this did not seem to hurt us in our experiments with chess, probably because the gradient does not change all that radically between adjacent regions in parameter space (of constant gradient).

Based on these observations we modified the TD(λ) algorithm to take account of minimax search in an almost trivial way: instead of working with the root positions $x_1, \ldots, x_N$, the

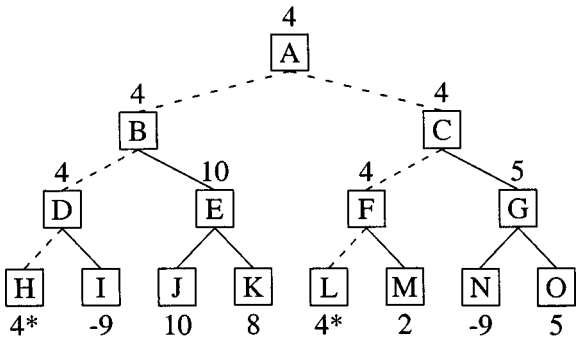

Figure 2. A search tree with a non-unique principal variation (PV). In this case the derivative of the root node A with respect to the parameters of the leaf-node evaluation function is multi-valued, either $\nabla \tilde{J}(H, w)$ or $\nabla \tilde{J}(L, w)$. Except for transpositions (in which case H and L are identical and the derivative is single-valued anyway), such "collisions" are likely to be extremely rare, so in TDLEAF(λ) we ignore them by choosing a leaf node arbitrarily from the available candidates.

TD(λ) algorithm is applied to the leaf positions found by minimax search from the root positions. We call this algorithm TDLEAF(λ). Full details are given in figure 3.

4. TDLEAF(λ) and chess

In this section we describe the outcome of several experiments in which the TDLEAF(λ) algorithm was used to train the weights of a linear evaluation function in our chess program "KnightCap". For details about the program itself, see Appendix A.

4.1. Experiments with KnightCap

In our main experiment we took KnightCap's evaluation function and set all but the material parameters to zero. The material parameters were initialized to the standard "computer" values: 1 for a pawn, 4 for a knight, 4 for a bishop, 6 for a rook and 12 for a queen. With these parameter settings KnightCap (under the pseudonym "WimpKnight") was started on the Free Internet Chess server (FICS, `fics.onenet.net`) against both human and computer opponents. We played KnightCap for 25 games without modifying its evaluation function so as to get a reasonable idea of its rating. After 25 games it had a blitz (fast time control) rating of 1650 ± 50,[2] which put it at about B-grade human performance (on a scale from E (1000) to A (1800)), although of course the kind of game KnightCap plays with just material parameters set is very different to human play of the same level (KnightCap makes no short-term tactical errors but is positionally completely ignorant). We then turned on the TDLEAF(λ) learning algorithm, with $\lambda = 0.7$ and the learning rate $\alpha = 1.0$. The value of λ was chosen heuristically, based on the typical delay in moves before an error takes effect, while α was set high enough to ensure rapid modification of the parameters. A couple of minor modifications to the algorithm were made:

Let $\tilde{J}(\cdot, w)$ be a class of evaluation functions parameterized by $w \in \mathbb{R}^k$. Let $x_1, \ldots, x_N$ be N positions that occurred during the course of a game, with $r(x_N)$ the outcome of the game. For notational convenience set $\tilde{J}(x_N, w) := r(x_N)$.

1. For each state x_i, compute $\tilde{J}_d(x_i, w)$ by performing minimax search to depth d from x_i and using $\tilde{J}(\cdot, w)$ to score the leaf nodes. Note that d may vary from position to position.

2. Let x_i^l denote the leaf node of the principle variation starting at x_i. If there is more than one principal variation, choose a leaf node from the available candidates at random. Note that

$$\tilde{J}_d(x_i, w) = \tilde{J}(x_i^l, w). \tag{9}$$

3. For $t = 1, \ldots, N-1$, compute the temporal differences:

$$d_t := \tilde{J}(x_{t+1}^l, w) - \tilde{J}(x_t^l, w). \tag{10}$$

4. Update w according to the TDLEAF(λ) formula:

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla \tilde{J}(x_t^l, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right]. \tag{11}$$

Figure 3. The TDLEAF(λ) algorithm.

– The raw (linear) leaf node evaluations $\tilde{J}(x_i^l, w)$ were converted to a score between -1 and 1 by computing

$$v_i^l := \tanh\left[\beta \tilde{J}(x_i^l, w)\right].$$

This ensured small fluctuations in the relative values of leaf nodes did not produce large temporal differences (the values v_i^l were used in place of $\tilde{J}(x_i^l, w)$ in the TDLEAF(λ) calculations). The outcome of the game $r(x_N)$ was set to 1 for a win, -1 for a loss and

0 for a draw. β was set to ensure that a value of $\tanh[\beta \tilde{J}(x_i^l, w)] = 0.25$ was equivalent to a material superiority of 1 pawn (initially).

- The temporal differences, $d_t = v_{t+1}^l - v_t^l$, were modified in the following way. Negative values of d_t were left unchanged as any decrease in the evaluation from one position to the next can be viewed as mistake. However, positive values of d_t can occur simply because the opponent has made a blunder. To avoid KnightCap trying to learn to predict its opponent's blunders, we set all positive temporal differences to zero unless KnightCap predicted the opponent's move.[3]
- The value of a pawn was kept fixed at its initial value so as to allow easy interpretation of weight values as multiples of the pawn value (we actually experimented with not fixing the pawn value and found it made little difference: after 1764 games with an adjustable pawn its value had fallen by less than 7 percent).

Within 300 games KnightCap's rating had risen to 2150, an increase of 500 points in three days, and to a level comparable with human masters. At this point KnightCap's performance began to plateau, primarily because it does not have an opening book and so will repeatedly play into weak lines. We have since implemented an opening book learning algorithm and with this KnightCap now plays at a rating of 2400–2500 (peak 2575) on the other major Internet chess server: ICC, `chessclub.com`.[4] It often beats International Masters at blitz. Also, because KnightCap automatically learns its parameters we have been able to add a large number of new features to its evaluation function: KnightCap currently operates with 5872 features (1468 features in four stages: opening, middle, ending and mating).[5] With this extra evaluation power KnightCap easily beats versions of Crafty restricted to search only as deep as itself. However, a big caveat to all this optimistic assessment is that KnightCap routinely gets crushed by faster programs searching more deeply. It is quite unlikely this can be easily fixed simply by modifying the evaluation function, since for this to work one has to be able to predict tactics statically, something that seems very difficult to do. If one could find an effective algorithm for "learning to search selectively" there would be potential for far greater improvement.

Note that we have twice repeated the original learning experiment and found a similar rate of improvement and final performance level. The rating as a function of the number of games from one of these repeat runs is shown in figure 4 (we did not record this information in the first experiment). Note that in this case KnightCap took nearly twice as long to reach the 2150 mark, but this was partly because it was operating with limited memory (8Mb) until game 500 at which point the memory was increased to 40Mb (KnightCap's search algorithm—MTD(f) (Plaat et al., 1996)—is a memory intensive variant of α–β and when learning KnightCap must store the whole position in the hash table so small memory significantly impacts upon performance). Another reason may also have been that for a portion of the run we were performing parameter updates after every four games rather than every game.

We also repeated the experiment using another variant of TD(λ), in which the temporal differences calculated were those between the positions actually occurring in the game, even though these positions had been selected by a deep minimax search rather than the usual one-ply search associated with TD(λ). We have dubbed this variant "TD-DIRECTED(λ)".

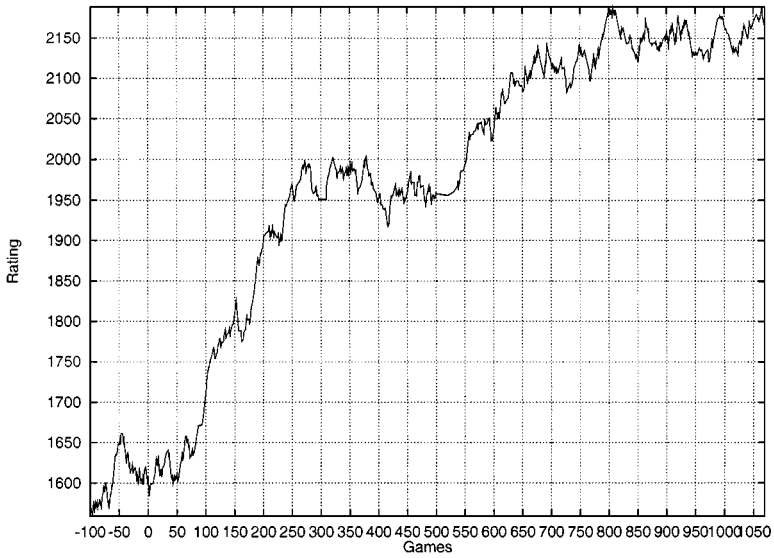

Figure 4. KnightCap's rating as a function of games played (second experiment). Learning was turned on at game 0.

With it we observed a 200 point rating rise over 300 games. A significant improvement, but much slower than TDLEAF(λ) and a lower peak. Its performance on backgammon is discussed in Section 5.

Plots of various parameters as a function of the number of games played are shown in figure 5 (these plots are from the same experiment in figure 4). Each plot contains three graphs corresponding to the three different stages of the evaluation function: opening, middle and ending.[6]

Finally, we compared the performance of KnightCap with its learnt weights to KnightCap's performance with a set of hand-coded weights, again by playing the two versions on ICC. The hand-coded weights were close in performance to the learnt weights (perhaps 50–100 rating points worse). We also tested the result of allowing KnightCap to learn starting from the hand-coded weights, and in this case it seems that KnightCap performs better than when starting from just material values (peak performance was 2632 compared to 2575, but these figures are very noisy). We are conducting more tests to verify these results. However, it should not be too surprising that learning from a good quality set of handcrafted parameters is better than just learning from material parameters. In particular, some of the handcrafted parameters have very high values (the value of an "unstoppable pawn", for example) which can take a very long time to learn under normal playing conditions, particularly if they are rarely active in the principal leaves. It is not yet clear whether given a sufficient number of games this dependence on the initial conditions can be made to vanish.

4.2. Discussion

There appear to be a number of reasons for the remarkable rate at which KnightCap improved.

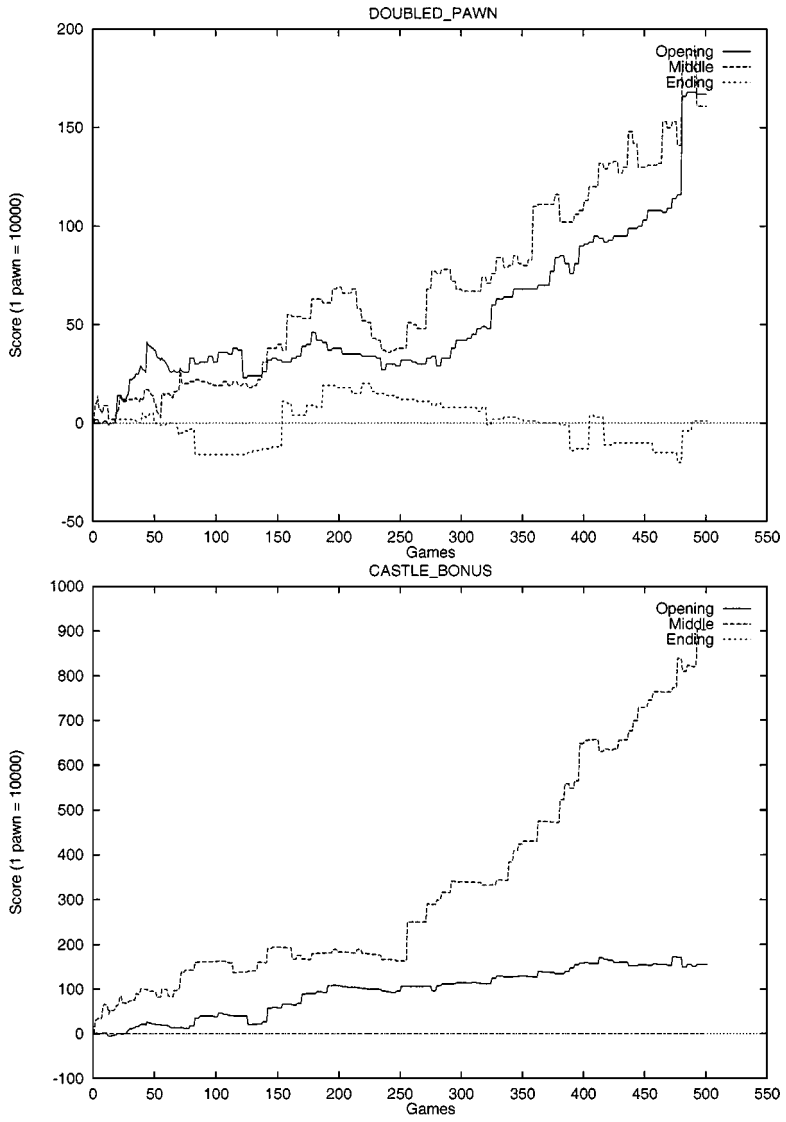

Figure 5. Evolution of two parameters (bonus for castling and penalty for a doubled pawn) as a function of the number of games played. Note that each parameter appears three times: once for each of the three stages in the evaluation function.

1. As all the non-material weights were initially zero, even small changes in these weights could cause very large changes in the relative ordering of materially equal positions. Hence even after a few games KnightCap was playing a substantially better game of chess.
2. It seems to be important that KnightCap started out life with intelligent material parameters. This put it close in parameter space to many far superior parameter settings.

3. Most players on FICS prefer to play opponents of similar strength, and so KnightCap's opponents improved as it did. This may have had the effect of *guiding* KnightCap along a path in weight space that led to a strong set of weights.
4. KnightCap was learning on-line, not by self-play. The advantage of on-line play is that there is a great deal of information provided by the opponent's moves. In particular, against a stronger opponent KnightCap was being shown positions that 1) could be forced (against KnightCap's weak play) and 2) were mis-evaluated by its evaluation function. Of course, in self-play KnightCap can also discover positions which are misevaluated, but it will not find the kinds of positions that are relevant to strong play against other opponents. In this setting, one can view the information provided by the opponent's moves as partially solving the "exploration" part of the *exploration/exploitation* tradeoff.

To further investigate the importance of some of these reasons, we conducted several more experiments.

Good initial conditions. A second experiment was run in which KnightCap's coefficients were all initialised to the value of a pawn. The value of a pawn needs to be positive in KnightCap because it is used in many other places in the code: for example we deem the MTD search to have converged if $\alpha < \beta + 0.07 * \texttt{PAWN}$. Thus, to set all parameters equal to the same value, that value had to be a pawn.

Playing with the initial weight settings KnightCap had a blitz rating of around 1250. After more than 1000 games on FICS KnightCap's rating has improved to about 1550, a 300 point gain. This is a much slower improvement than the original experiment.[7] We do not know whether the coefficients would have eventually converged to good values, but it is clear from this experiment that starting near to a good set of weights is important for fast convergence. An interesting avenue for further exploration here is the effect of λ on the learning rate. Because the initial evaluation function is completely wrong, there would be some justification in setting $\lambda = 1$ early on so that KnightCap only tries to predict the outcome of the game and not the evaluations of later moves (which are extremely unreliable).

Self-play. Learning by self-play was extremely effective for TD-Gammon, but a significant reason for this is the randomness of backgammon which ensures that with high probability different games have substantially different sequences of moves, and also the speed of play of TD-Gammon which ensured that learning could take place over several hundred-thousand games. Unfortunately, chess programs are slow, and chess is a deterministic game, so self-play by a deterministic algorithm tends to result in a large number of substantially similar games. This is not a problem if the games seen in self-play are "representative" of the games played in practice, however KnightCap's self-play games with only non-zero material weights are very different to the kind of games humans of the same level would play.

To demonstrate that learning by self-play for KnightCap is not as effective as learning against real opponents, we ran another experiment in which all but the material parameters were initialised to zero again, but this time KnightCap learnt by playing against itself. After 600 games (twice as many as in the original FICS experiment), we played the resulting

version against the good version that learnt on FICS for a further 100 games with the weight values fixed. The self-play version scored only 11% against the good FICS version.

5. Experiment with backgammon

For our backgammon experiment we were fortunate to have Mark Land (Computer Science Department, University of California, San Diego) provide us with the source code for his LGammon program which uses self-play and TD(λ) to train a backgammon playing neural network. The code has served as both a base on which to implement TDLEAF(λ)-based training, and as a benchmark for measuring the success of this training.

5.1. LGammon

Land's LGammon program has been implemented along the lines of Tesauro's TD-Gammon (Tesauro, 1992, 1994). Like Tesauro, Land uses a raw board representation coupled with some hand-coded features, and uses self-play based upon one-ply search to generate training data. During each game, the positions encountered and their evaluations are recorded, with error signals and consequent weight updates being calculated and applied after the game.

Along with the code for LGammon, Land also provided a set of weights for the neural network. The weights are those which LGammon has used for most of the time it has been playing on the First Internet Backgammon Server (FIBS, fibs.com). With these weights LGammon achieved a rating on FIBS which ranged from 1600 to 1680, significantly above the mean rating across all players of about 1500. For convenience, we refer to the weights simply as the *FIBS weights*.

5.2. Experiment with LGammon

The stochasticity inherent in backgammon complicates the implementation of TD-DIRECTED(λ) and TDLEAF(λ). Using minimax search to a depth of one-ply with backgammon is simple, because the set of legal moves is fully determined by the board position and the dice roll. Searching to two-ply however, requires considering for each position reached in one-ply, the twenty-one distinct dice rolls which could follow, and the subsequent moves which the opponent may choose. Consequently, we have defined the two-ply evaluation of a position in the obvious manner, using the expected value across the dice rolls, of the positions reachable from each one-ply position. Adapting the notation defined in Section 3 such that x'_{adb} refers to board position x subject to action a, dice roll d, and action b, we choose action a in accordance with

$$a^*(x) := \arg\min_{a \in A_x} \left(E_{x'_{ad}|x'_a} \arg\min_{b \in A_{x'_{ad}}} \tilde{J}(x'_{adb}, w) \right). \tag{12}$$

where the expectation is with respect to the transition probabilities $p(x'_a, x'_{ad})$.

TD-DIRECTED(λ) then stores and trains using the one-ply positions, even though these are chosen by the two-ply search just described. Since the averaging across dice rolls for depth two means there is not an explicit principal variation, TDLEAF(λ) approximates the leaf node with the expectation term of Eq. (12) which corresponds to the branch of the game tree selected.

Similarly for the derivative of a two-ply terminal position under TDLEAF(λ), we calculate the expected value of the derivative with respect to these transition probabilities.

Limit of learning. Our experiment sought to determine whether TDLEAF(λ) or TD-DIRECTED(λ) could find better weights than standard TD(λ). To test this, we took two copies of the FIBS weights, the end product of a standard TD(λ) training run, and trained one with each of our variants and self-play.

The networks were trained for 50000 games, and check-pointed every 5000 games. To test the effectiveness of the training, the check-point networks were played against the unmodified FIBS weights for 1600 games, with both sides searching to two-ply and the match score being recorded.

The results fluctuated around parity with the FIBS weights (the result of training with standard TD(λ) for the duration of the training, with no consistent or statistically significant change in relative performance being observed.

If the optimal networks for two-ply and one-ply play are not the same, we would expect our variants to achieve some improvement over the course of 50000 games of training. That this didn't happen, suggests that the solution found by standard TD(λ), which only searches to one-ply in training, is either at or near the optimal for two-ply play.

6. Future work

TDLEAF(λ) is a general method for combining search and TD(λ). As such, it should be applicable to domains where search is beneficial and an evaluation function needs to be learnt. This includes games such as othello, shogi, checkers, and Go. However, there are also many non-game domains requiring deep search which may benefit from on-line learning of an evaluation function. These include agent planning, automated theorem proving, and instruction scheduling in optimising compilers.

We also need note of the backgammon result of Section 5, which shows that deeper searching TDLEAF(λ) and TD-DIRECTED(λ) don't always improve on the solution of one-step look-ahead TD(λ). This begs the question of whether our variants will, in general, converge to solutions of the same quality as TD(λ). Obviously domain specific characteristics can influence this,[8] so empirically it is impossible to prove, but a theoretical result would be useful.

For domains where both normal TD(λ) and TDLEAF(λ) are feasible, the important question of which converges faster remains open. Backgammon may be an unusual case, because the branching factor, induced by the stochasticity at each turn, is quite large and makes searching an additional ply expensive. Thus it is possible that TD(λ) converges faster in terms of CPU time, though we suspect that TDLEAF(λ) may converge faster in terms of games played.

7. Conclusion

We have introduced TDLEAF(λ), a variant of TD(λ) suitable for training an evaluation function used in minimax search. The only extra requirement of the algorithm is that the leaf-nodes of the principal variations be stored throughout the game.

We presented some experiments in which a chess evaluation function was trained from B-grade to master level using TDLEAF(λ) by on-line play against a mixture of human and computer opponents. The experiments show both the importance of "on-line" sampling (as opposed to self-play) for a deterministic game such as chess, and the need to start near a good solution for fast convergence, although just how near is still not clear.

We also demonstrated that in the domain of backgammon, TDLEAF(λ) and TD-DIRECTED(λ) were unable to improve upon a good network trained by TD(λ). This suggests that the optimal network to use in 1-ply search is close to the optimal network for 2-ply search.

KnightCap is freely available on the web from `http://wwwsyseng.anu.edu.au/lsg/ knightcap.html`.

Acknowledgments

Jonathan Baxter was supported by an Australian Postdoctoral Fellowship. Lex Weaver was supported by an Australian Postgraduate Award.

Appendix A: KnightCap

KnightCap is a reasonably sophisticated computer chess program for Unix systems. It has all the standard algorithmic features that modern chess programs tend to have as well as a number of features that are much less common. This section is meant to give the reader an overview of the type of algorithms that have been chosen for KnightCap. Space limitations prevent a full explanation of all of the described features, an interested reader should be able find explanations in the widely available computer chess literature (see for example (Marsland & Schaeffer, 1990)) or by examining the source code: `http://wwwsyseng.anu.edu.au/lsg`.

A.1. Board representation

This is where KnightCap differs most from other chess programs. The principal board representation used in KnightCap is the *topieces* array. This is an array of 32 bit words with one word for each square on the board. Each bit in a word represents one of the 32 pieces in the starting chess position (8 pieces + 8 pawns for each side). Bit i on square j is set if piece i is attacking square j.

The topieces array has proved to be a very powerful representation and allows the easy description of many evaluation features which are more difficult or too costly with other representations. The array is updated dynamically after each move in such a way that for

the vast majority of moves only a small proportion of the topieces array need be directly examined and updated.

A simple example of how the topieces array is used in KnightCap is determining whether the king is in check. Whereas an in_check ( ) function is often quite expensive in chess programs, in KnightCap it involves just one logical AND operation in the topieces array. In a similar fashion the evaluation function can find common features such as connected rooks using just one or two instructions.

The topieces array is also used to drive the move generator and obviates the need for a standard move generation function.

A.2. Search algorithm

The basis of the search algorithm used in KnightCap is MTD(f) (Plaat et al., 1996). MTD(f) is a logical extension of the minimal-window alpha-beta search that formalizes the placement of the minimal search window to produce what is in effect a bisection search over the evaluation space.

The variation of MTD(f) that KnightCap uses includes some convergence acceleration heuristics that prevent the very slow convergence that can sometimes plague MTD(f) implementations. These heuristics are similar in concept to the momentum terms commonly used in neural network training.

The MTD(f) search algorithm is applied within a standard iterative deepening framework. The search begins with the depth obtained from the transposition table for the initial search position and continues until a time limit is reached in the search. Search ordering at the root node ensures that partial ply search results obtained when the timer expires can be used quite safely.

A.3. Null moves

KnightCap uses a recursive null move forward pruning technique. Whereas most null move using chess programs use a fixed R value (the number of additional ply to prune when trying a null move) KnightCap instead uses a variable R value in an asymmetric fashion. The initial R value is 3 and the algorithm then tests the result of the null move search. If it is the computers side of the search and the null move indicates that the position is "good" for the computer then the R value is decreased to 2 and the null move is retried.

The effect of this null move system is that most of the speed of a $R = 3$ system is obtained, while making no more null move defensive errors than an $R = 2$ system. It is essentially a pessimistic system.

A.4. Search extensions

KnightCap uses a large number of search extensions to ensure that critical lines are searched to sufficient depth. Extensions are indicated through a combination of factors including check, null-move mate threats, pawn moves to the last two ranks and recapture extensions. In addition KnightCap uses a single ply razoring system with a 0.9 pawn razoring threshold.

A.5. Asymmetries

There are quite a number of asymmetric search and evaluation terms in KnightCap, with a leaning towards pessimistic (i.e. careful) play. Apart from the asymmetric null move and search extensions systems mentioned above, KnightCap also uses an asymmetric system to decide what moves to try in the quiescence search and several asymmetric evaluation terms in the evaluation function (such as king safety and trapped piece factors).

When combined with the TDLEAF(λ) algorithm KnightCap is able to learn appropriate values for the asymmetric evaluation terms.

A.6. Transposition tables

KnightCap uses a standard two-deep transposition table with a 128 bit transposition table entry. Each entry holds separate depth and evaluation information for the lower and upper bound.

The ETTC (enhanced transposition table cutoff) technique is used both for move ordering and to reduce the tree size. The transposition table is also used to feed the book learning system and to initialize the depth for iterative deepening.

A.7. Move ordering

The move ordering system in KnightCap uses a combination of the commonly used history (Schaeffer, 1989), killer, refutation and transposition table ordering techniques. With a relatively expensive evaluation function KnightCap can afford to spend a considerable amount of CPU time on move ordering heuristics in order to reduce the tree size.

A.8. Parallel search

KnightCap has been written to take advantage of parallel distributed memory multi-computers, using a parallelism strategy that is derived naturally from the MTD(f) search algorithm. Some details on the methodology used and parallelism results obtained are available in Tridgell (1997). The results given in this paper were obtained using a single CPU machine.

A.9. Evaluation function

The heart of any chess program is its evaluation function. KnightCap uses quite a slow evaluation function that evaluates a number of computationally expensive features. The evaluation function also has four distinct stages: Opening, Middle, Ending and Mating, each with its own set of parameters (but the same features). We have listed the names of all KnightCap's features in Table A.1. Note that some of the features have more than one parameter associated with them, for example there are 64 parameters associated with rook position, one for each square. These features all begin with "I". To summarize just a few of the more obscure features: IOPENING_KING_ADVANCE is a bonus for the rank of the king in the opening, it has 8 parameters, one for each rank. IMID_KING_ADVANCE

Table A.1. KnightCap's features and the number of parameters corresponding to each. Most of the features are self-explanatory, see the text for a description of the more obscure ones. Note that KnightCap's large number of parameters is obtained by summing all the numbers in this table and then multiplying by the number of stages (four).

Feature	#	Feature	#
BISHOP_PAIR	1	CASTLE_BONUS	1
KNIGHT_OUTPOST	1	BISHOP_OUTPOST	1
SUPPORTED_KNIGHT_OUTPOST	1	SUPPORTED_BISHOP_OUTPOST	1
CONNECTED_ROOKS	1	SEVENTH_RANK_ROOKS	1
OPPOSITE_BISHOPS	1	EARLY_QUEEN_MOVEMENT	1
IOPENING_KING_ADVANCE	8	IMID_KING_ADVANCE	8
IKING_PROXIMITY	8	ITRAPPED_STEP	8
BLOCKED_KNIGHT	1	USELESS_PIECE	1
DRAW_VALUE	1	NEAR_DRAW_VALUE	1
NO_MATERIAL	1	MATING_POSITION	1
IBISHOP_XRAY	5	IENDING_KPOS	8
IROOK_POS	64	IKNIGHT_POS	64
IPOS_BASE	64	IPOS_KINGSIDE	64
IPOS_QUEENSIDE	64	IKNIGHT_MOBILITY	80
IBISHOP_MOBILITY	80	IROOK_MOBILITY	80
IQUEEN_MOBILITY	80	IKING_MOBILITY	80
IKNIGHT_SMOBILITY	80	IBISHOP_SMOBILITY	80
IROOK_SMOBILITY	80	IQUEEN_SMOBILITY	80
IKING_SMOBILITY	80	IPIECE_VALUES	6
THREAT	1	OPPONENTS_THREAT	1
IOVERLOADED_PENALTY	15	IQ_KING_ATTACK_COMPUTER	8
IQ_KING_ATTACK_OPPONENT	8	INOQ_KING_ATTACK_COMPUTER	8
INOQ_KING_ATTACK_OPPONENT	8	QUEEN_FILE_SAFETY	1
NOQUEEN_FILE_SAFETY	1	IPIECE_TRADE_BONUS	32
IATTACK_VALUE	16	IPAWN_TRADE_BONUS	32
UNSUPPORTED_PAWN	1	ADJACENT_PAWN	1
IPASSED_PAWN_CONTROL	21	UNSTOPPABLE_PAWN	1
DOUBLED_PAWN	1	WEAK_PAWN	1
ODD_BISHOPS_PAWN_POS	1	BLOCKED_PASSED_PAWN	1
KING_PASSED_PAWN_SUPPORT	1	PASSED_PAWN_ROOK_ATTACK	1
PASSED_PAWN_ROOK_SUPPORT	1	BLOCKED_DPAWN	1
BLOCKED_EPAWN	1	IPAWN_ADVANCE	7
IPAWN_ADVANCE1	7	IPAWN_ADVANCE2	7
KING_PASSED_PAWN_DEFENCE	1	IPAWN_POS	64
IPAWN_DEFENCE	12	ISOLATED_PAWN	1
MEGA_WEAK_PAWN	1	IWEAK_PAWN_ATTACK_VALUE	8

is the same but applies in the middle game (the fact that we have separate features for the opening and middle games is a hangover from KnightCap's early days when it didn't have separate parameters for each stage). IKING_PROXIMITY is the number of moves between our king and the opponents king. It is very useful for forcing mates in the ending. Again there is one parameter for each of the 8 possible values. IPOS_BASE is the base score for controlling each of the squares. IPOS_KINGSIDE and IPOS_QUEENSIDE are modifications added to IPOS_BASE according as KnightCap is castled on the king or queen sides respectively. The MOBILITY scores are the number of moves available to a piece, thresholding at 10. There is a separate score for each rank the piece is on, hence the total number of parameters of 80. The SMOBILITY scores are the same, but now the square the piece is moving to has to be safe (i.e. controlled by KnightCap). THREAT and OPPONENTS_THREAT are computed by doing a minimax search on the position in which only captures are considered and each piece can move only once. Its not clear this helps the evaluation much, but it certainly improves move ordering (the best capture is given a high weight in the ordering). IOVERLOADED_PENALTY is a penalty that is applied to each piece for the number of otherwise hung pieces it is defending. There is a separate penalty for each number, thresholding at 15 (this could be done better: we should have a base score times by the number of pieces, and have KnightCap learn the base score and a perturbation on the base score for each number). IQ_KING_ATTACK_OPPONENT and INOQ_KING_ATTACK_OPPONENT are bonuses for the number of pieces KnightCap has attacking the squares around the enemy king, both with and without queens on the board. IQ_KING_ATTACK_COMPUTER and INOQ_KING_ATTACK_COMPUTER are the same thing for the opponent attacking KnightCap's king. Note that this asymmetry allows KnightCap the freedom to learn to be cautious by assigning greater weight to opponent pieces attacking its own king that it does to its own pieces attacking the opponent's king. It can of course also use this to be aggressive. For more information on the features, see eval.c in KnightCap's source code.

The most computationally expensive part of the evaluation function is the "board control". This function evaluates a control function for each square on the board to try to determine who controls the square. Control of a square is essentially defined by determining whether a player can use the square as a flight square for a piece, or if a player controls the square with a pawn.

Despite the fact that the board control function is evaluated incrementally, with the control of squares only being updated when a move affects the square, the function typically takes around 30% of the total CPU time of the program. This high cost is considered worthwhile because of the flow-on effects that this calculation has on other aspects of the evaluation and search. These flow-on effects include the ability of KnightCap to evaluate reasonably accurately the presence of hung, trapped and immobile pieces which is normally a severe weakness in computer play. We have also noted that the more accurate evaluation function tends to reduce the search tree size thus making up for the decreased node count.

A.10. Modification for TDLEAF(λ)

The modifications made to KnightCap for TDLEAF(λ) affected a number of the program's subsystems. The largest modifications involved the parameterization of the evaluation

function so that all evaluation parameters became part of a single long weight vector. All tunable evaluation knowledge could then be described in terms of the values in this vector.

The next major modification was the addition of the full board position in all data structures from which an evaluation value could be obtained. This involved the substitution of a structure for the usual scalar evaluation type, with the evaluation function filling in the evaluated position and other board state information during each evaluation call. Similar additions were made to the transposition table entries so that the result of a search would always have available to it the position associated with the leaf node in the principal variation. This significantly enlarges the transposition table and means that to operate effectively with the MTD(f) search algorithm (itself a memory-hungry α-β variant), KnightCap really needs at least 30Mb of hash table when learning.

The only other significant modification that was required was an increase in the bit resolution of the evaluation type so that a numerical partial derivative of the evaluation function with respect to the evaluation coefficient vector could be obtained with reasonable accuracy.

Notes

1. If successor states are only determined stochastically by the choice of a, we would choose the action minimizing the expected reward over the choice of successor states.
2. The standard deviation for all ratings reported in this section is about 50.
3. In a later experiment we only set positive temporal differences to zero if KnightCap did not predict the opponent's move *and* the opponent was rated less than KnightCap. After all, predicting a stronger opponent's blunders is a useful skill, although whether this made any difference is not clear.
4. There appears to be a systematic difference of around 200–250 points between the two servers, so a peak rating of 2575 on ICC roughly corresponds to a peak of 2350 on FICS. We transferred KnightCap to ICC because there are more strong players playing there.
5. In reality there are not 1468 independent "concepts" per stage in KnightCap's evaluation function as many of the features come in groups of 64, one for each square on the board (like the value of placing a rook on a particular square, for example).
6. KnightCap actually has a fourth and final stage "mating" which kicks in when all the pawns are off, but this stage only uses a few of the coefficients (opponent's king mobiliity and proximity of our king to the opponent's king).
7. We ran this experiment three times, with the result reported being the best achieved. Since the main experiment succeeded on all three occasions it was run, it is unlikely that the slower ratings improvement in this experiment is due to vagaries in the training environment.
8. A chess program using only one-step look-ahead would lose most games against reasonable quality opponents and would thus learn to value all positions as lost. This contrasts with KnightCap whose deep search makes competing with better players possible.

References

Beal, D. F. & Smith, M. C. (1997). Learning piece values using temporal differences. *Journal of The International Computer Chess Association*.

Bertsekas, D. P. & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Marsland, T. A. & Schaeffer, J. (1990). *Computers, Chess and Cognition*. Springer Verlag.

Plaat, A., Schaeffer, J., Pijls, W., & de Bruin, A. (1996). Best-first fixed-depth minmax algorithms. *Artificial Intelligence*, *87*, 255–293.

Pollack, J., Blair, A., & Land, M. (1996). Coevolution of a backgammon player. In *Proceedings of the Fifth Artificial Life Conference*, Nara, Japan.

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

Schaeffer, J. (1989). The history of heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Learning*, *11*(11), 1203–1212.

Schraudolph, N., Dayan, P., & Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of go. In J. Cowan, G. Tesauro, & J. Alspector (Eds.), *Advances in Neural Information Processing Systems 6*. San Fransisco: Morgan Kaufmann.

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, *3*, 9–44.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge MA: MIT Press. ISBN 0-262-19398-1.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257–278.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*, 215–219.

Thrun, S. (1995). Learning to play the game of chess. In G.Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*. San Fransisco: Morgan Kaufmann.

Tridgell, A. (1997). KnightCap—A parallel chess program on the AP1000+. In *Proceedings of the Seventh Fujitsu Parallel Computing Workshop*, Canberra, Australia. ftp://samba.anu.edu.au/tridge/knightcap_pcw97.ps.gz source code: http://wwwsysneg.anu.edu.au/lsg.

Tsitsiklis, J. N. & Roy, B. V. (1997). An analysis of temporal difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*(5), 674–690.

Walker, S., Lister, R., & Downs, T. (1993). On self-learning patterns in the othello board game by the method of temporal differences. In C. Rowles, H. Liu, & N. Foo (Eds.), *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence* (pp. 328–333). Melbourne: World Scientific.

Received June 19, 1998
Revised November 29, 1999
Final manuscript November 29, 1999