KnightCap: A chess program that learns by combining TD(λ) with minimax search

Jonathan Baxter
Department of Systems Engineering
Australian National University
Canberra 0200, Australia

Andrew Tridgell
Department of Computer Science
Australian National University
Canberra 0200, Australia

Lex Weaver
Department of Computer Science
Australian National University
Canberra 0200, Australia
{Jon.Baxter,Andrew.Tridgell,Lex.Weaver}@anu.edu.au

November 27, 1997

Abstract

In this paper we present TDLeaf(λ), a variation on the TD(λ) algorithm that enables it to be used in conjunction with minimax search. We present some experiments in which our chess program, "KnightCap," used TDLeaf(λ) to learn its evaluation function while playing on the Free Ineternet Chess Server (FICS, `fics.onenet.net`). It improved from a 1650 rating to a 2100 rating in just 308 games and 3 days of play. We discuss some of the reasons for this success and also the relationship between our results and Tesauro's results in backgammon.

1 Introduction

Temporal Difference learning or TD(λ), first introduced by Sutton [9], is an elegant algorithm 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 transistion, 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 [11]. 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 chess [12, 15, 8].

Many authors have discussed the peculiarities of backgammon that make it particularly suitable for Temporal Difference learning with self-play [10, 8, 7]. Principle among these are *speed of play*: TD-Gammon learnt from several hundered 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 (high branching factor alone cannot be the reason because humans search very deeply in Go, a game with a similar branching factor to backgammon).

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 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).

In this paper we introduce TDLeaf(λ), a variation on the TD(λ) theme that can be used to learn an evaluation function for use in deep minimax search. TDLeaf(λ) is identical to TD(λ) except that instead of operating on the positions that occur during

the game, it operates on the leaf nodes of the *principal variation* of a minimax search from each position.

To test the effectiveness of TDLeaf(λ), we incorporated it into our own chess program — KnightCap. KnightCap has a particularly rich board representation enabling relatively fast computation of sophisticated positional features. 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 self-play[1].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 2100-rated player in just three days and 308 games. KnightCap is an ongoing project with new features being added to its evaluation function continually. We use TDLeaf(λ) and Internet play to tune the coefficients of these features

As this paper was going to press we discovered [1] in which the same idea was used to tune the material values in a chess program that only operated with material values. The learnt values came out very close to the "traditional" chess values of 1:3:3;5:9 for pawn,knight,bishop,rook,queen, but were learnt by self-play rather than on-line play. With KnightCap, we found self-play to be worse than on-line learning (see section 4), but KnightCap's performance was being measured against the on-line players, not against a fixed program as in [1].

The remainder of this paper is organised 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. Some details of Knightcap are given in section 4.1. Experimental results for Internet-play with chess are given in sections 4. Section 5 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

[1]Randomizing move choice is another way of avoiding problems associated with self-play (this approach has been tried in Go [8]), but the advantage of the Internet is that more information is provided by the opponents play.

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 chosses its actions stochastically.

For vary 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 function, 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

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

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. It is this observation that motivates the TD(λ) algorithm.

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 λ controls the extent to which temporal differences propagate backwards in time. To see this,

compare equation (3) for $\lambda = 0$:

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

and $\lambda = 1$:

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

Consider each term contributing to the sums in equations (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 a way 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.

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 [14]. Unfortunately, there is no such guarantee if $\tilde{J}(\cdot, w)$ is non-linear [14], or if $a(x_t)$ depends on w [2]. However, despite the lack of theoretical guarantees there have been many successful applications of the TD(λ) algorithm [3].

3 Minimax Search and TD(λ)

For most games, any action a taken in state x will lead 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[2]:

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

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

[2] 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.

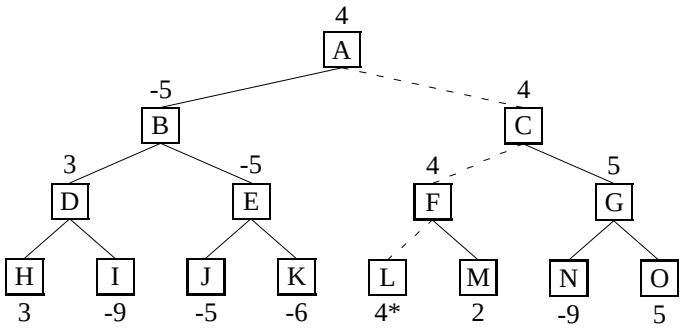

Figure 1: Full breadth, 3-ply search tree illustrating the minimax rule for propagating values. Each of the leaf nodes (H–O) is given a score by the evaluation function, $\tilde{J}(\cdot, w)$. These scores are then propagated back up the tree by assigning to each opponent's internal node the minimum 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 $\nabla \tilde{J}(L, w)$.

$\tilde{J}(\cdot, w)$), 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, $\alpha\beta$ 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

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

as per equation (2), and then the TD(λ) algorithm 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 equation (8) is that for $d > 1$, $\tilde{J}_d(x, w)$ is not a necessarily a differentiable function of w for all values of w, even if $\tilde{J}(\cdot, w)$ is everywhere differentiable.

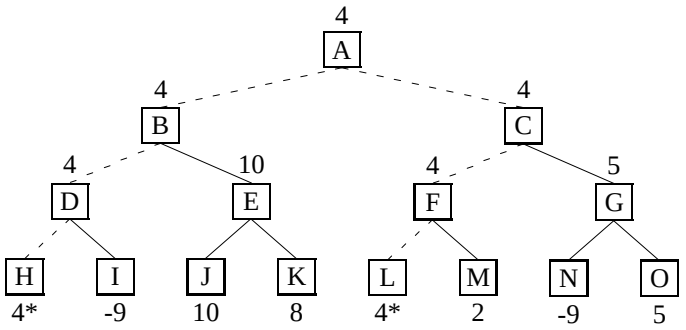

Figure 2: A search tree with a non-unique principal variation (PV). In this case the derivative of the root node A with respect to the parameters of the leaf-node evaluation function is multi-valued, either $\nabla \tilde{J}(H, w)$ or $\nabla \tilde{J}(L, w)$. Except for transpositions (in which case 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.

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.

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

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

4.1 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 [5] and [4]) or by examining the source code: `http://wwwsyseng.anu.edu.au/lsg`.

4.1.1 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.

4.1.2 Search algorithm

The basis of the search algorithm used in KnightCap is MTD(f) [6]. MTD(f) is a logical extension of the minimal-window alpha-beta search that formalises 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.

4.1.3 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 plys 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.

4.1.4 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.

4.1.5 Asymmetries

There are quite a number of asymmetric search and evaluation terms in KnightCap, with a leaning towards pessimistic (ie. 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 quiesce search and several asymmetric evaluation terms in the evaluation function (such as king safety and trapped piece factors).

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

4.1.6 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 initialise the depth for iterative deepening.

4.1.7 Move ordering

The move ordering system in KnightCap uses a combination of the commonly used history, 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.

4.1.8 Parallel search

KnightCap has been written to take advantage of parallel distributed memory multicomputers, 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 [13]. The results given in this paper were obtained using a single CPU machine.

4.1.9 Evaluation function

The heart of any chess program is its evaluation function. KnightCap uses a quite slow evaluation function that evaluates a number of quite computationally expensive features.

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.

4.1.10 Modification for TDLeaf

The modifications made to KnightCap for TDLeaf affected a number of the program's subsystems. The largest modifications involved the parameterisation of the evaluation function so that all evaluation coefficients became part of a single long weight vector. All 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 and book learning data so that the result of a search would always have available to it the position associated with the leaf node in the principal variation.

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.

4.2 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 initialised 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 pseodonym "WimpKnight") was set playing 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, which put it at about C-level human performance (the standard deviation for all ratings reported in this section is about 50). Figure 4 gives an example of the kind of game it plays against itself with just a material evaluation function. 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 changes to the algorithm were made:

- 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 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 $\tanh\left[\beta \tilde{J}(x_i^l, w)\right] = 0.25$ was equivalent to a material superiority of 1 pawn.

- 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.

- 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.

Within 300 games KnightCap's rating had risen 2100, an increase of 450 points in three days. 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 2500 on the other major internet chess server (ICC, `chessclub.com`). It regularly beats International Masters at blitz, and has a similar performance to Crafty (the best public domain program) at longer time controls.

We repeated the experiment using TD(λ) applied to the root nodes of the search (i.e. the actual positions as they occurred in the game), rather than the leaf nodes of the

principal variation, and observed a 200 point rating rise over 300 games. A significant improvement, but much slower than TDLeaf(λ) and a lower peak.

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

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 not "thrown in the deep end" against much stronger opponents, hence it receievd both positive and negative feedback from its games.

5. KnightCap was not learning by self-play.

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. 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).

Positive and negative feedback
To further investigate the importance of balanced feedback, we again initialised every parameter in KnightCap's evaluation function to the value of a pawn, and this weak version was played against another version with much stronger parameter settings. Apart from a few draws by repetition in the opening, the weak version lost every single one

of 2811 games. Although the weak version after training was stronger than the original pawn-weight version, it was still a very weak player—for example, it would sacrifice a queen for a knight and two pawns in most positions. The reason for the improvement over the pawn-weight version was that the trained version had learnt increased material weight and hence would tend to hang on to its material whereas the pawn-weight version quite happily threw its material away.

On face value it seems remarkable that the trained version learnt anything useful at all by losing 2811 games. If every game is lost, the only fixed point of the TD(λ) algorithm (and the TDLeaf(λ) algorithm) is to value every position as a loss from the outset. In fact this is close to the solution found by KnightCap: its evaluation of the starting position was -9 pawns by the end of the simulation (which gives a value very close to -1 after squashing). It achieved this primarily by exploiting the material weights and two asymmetric positional features: whether there are zero of the opponent's pieces attacking its own king, and whether there are zero of its own pieces attacking the opponent's king. KnightCap learnt coefficients for these features of -5.35 pawns and -3.23 pawns repectively, which meant that any position in which there was no attack on either king (like the starting position), these features contributed -8.58 pawns to KnightCap's evaluation function. The material weights increased because its opponent was a strong player so in most games the weak version would very soon be down in material, which encourages positive material parameters because that gives a negative value for the position.

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. However, chess is a deterministic game and 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, but one can see from the example game in figure 4 that 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.

In [1] positive results using essentially TDLeaf and self-play (with some random move choice) were reported for only learning the material weights. However, they were not comparing performance against on-line players, but were primarily investigating whether the weights would converge to "sensible" values as least as good as the naive (1,3,3,5,9) values for (pawn,knight,bishop,rook,queen) (they did). In view of these positive results, and our negative results with self-play, further investigation is required to determine how critical on-line learning is for good play.

KNIGHTCAP VS. KNIGHTCAP

1. Na3 Nc6 2. Nb5 Rb8 3. Nc3 Ra8 4. Rb1 d6 5. Ra1 Na5 6. Nf3 Bd7 7. d3 Be6 8. Qd2 Nf6 9. a3 Nc6 10. Qe3 Nb8 11. Nd4 Bc8 12. Qf3 Qd7 13. Nb3 Nc6 14. Kd1 Ne5 15. Qf4 Ng6 16. Qg3 c5 17. Ne4 Nd5 18. Kd2 Qb5 19. Nc3 Nxc3 20. bxc3 Qa6 21. Rb1 Kd8 22. Qf3 Ke8 23. Kd1 Qc6 24. Ke1 Be6 25. Kd2 Kd7 26. Na1 b6 27. Ke1 Kc7 28. Rb2 Ne5 29. Qxc6+ Kxc6 30. Be3 Kd5 31. Kd1 Nd7 32. Rb1 Kc6 33. c4 d5 34. cxd5+ Bxd5 35. Bf4 Rd8 36. Rb2 Nf6 37. Nb3 Be6 38. Ke1 Kd5 39. a4 Kc6 40. c4 Kd7 41. Bd2 Ne8 42. Ra2 Kc6 43. e4 Kd7 44. Be2 Rc8 45. Kd1 Kd8 46. Bf4 Nc7 47. Na1 Na6 48. Kc1 Ke8 49. Be3 Rb8 50. Kb1 Ra8 51. Bf1 Nb4 52. Ra3 Rb8 53. Nc2 Na6 54. Ra2 Bd7 55. Bf4 Rb7 56. Be3 Nb8 57. d4 cxd4 58. Nxd4 Na6 59. Nb5 Nb4 60. Ra1 Bxb5 61. cxb5 Rd7 62. Kc1 Kd8 63. Bd2 Rc7+ 64. Kd1 Nc2 65. Ra2 Nd4 66. Be3 Ne6 67. Bd3 Kd7 68. Bc2 Kc8 69. Ra3 Kd8 70. Kc1 Rd7 71. Rc3 Nd4 72. Rd3 e5 73. Rhd1 Bc5 74. Bxd4 exd4 75. Kd2 f6 76. Rb3 Rb7 77. Bd3 g6 78. g3 Kc7 79. Ke2 Kd7 80. Kf3 Rbb8 81. Kg4 Kc7 82. f3

Figure 4: The first 82 moves of a game by KnightCap against itself, where both sides were using an evaluation function in which the only non-zero parameters were the material coefficients. Such a configuration gives KnightCap a blitz rating of about 1650 on FICS, but as can be seen from the game it plays nothing like a human of similar rating.

5 Discussion and 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 linear chess evaluation function was trained 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), and the need to start near a good solution for fast convergence.

We compared training using leaf nodes (TDLeaf(λ)) with training using root nodes, and found a significant improvement training on the leaf nodes in chess.

We are currently investigating how well TDLeaf(λ) works for Backgammon, with a view to understanding the differences between chess and backgammon and how they affect learning.

On the theoretical side, it has recently been shown that TD(λ) converges for linear evaluation functions[14]. An interesting avenue for further investigation would be to determine whether TDLeaf(λ) has similar convergence properties.

References

[1] D. F. Beal and M. C. Smith. Learning Piece values Using Temporal Differences. *Journal of The International Computer Chess Association*, 1997.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

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

[4] D. Levy and M. Newborn. *How Computers Play Chess*. W. H. Freeman and Co., 1990.

[5] T. A. Marsland and J. Schaeffer. *Computers, Chess and Cognition*. Springer Verlag, 1990.

[6] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minmax Algorithms. *Artificial Intelligence*, 87:255–293, 1996.

[7] J. Pollack, A. Blair, and M. Land. Coevolution of a Backgammon Player. In *Proceedings of the Fifth Artificial Life Conference*, Nara, Japan, 1996.

[8] N. Schraudolph, P. Dayan, and T. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, San Fransisco, 1994. Morgan Kaufmann.

[9] R. Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[10] G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–278, 1992.

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

[12] 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*, San Fransisco, 1995. Morgan Kaufmann.

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

[14] J. N. Tsitsikilis and B. V. Roy. An Analysis of Temporal Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[15] S. Walker, R. Lister, and T. Downs. On Self-Learning Patterns in the Othello Board Game by the Method of Temporal Differences. In C. Rowles, H. liu, and N. Foo, editors, *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence*, pages 328–333, Melbourne, 1993. World Scientific.