REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Experiments with Automatic Weight Tuning in Heuristic Evaluation Functions

Jónas Tryggvi Jóhannsson
Master of Science
January 2007

Supervisor:
Yngvi Björnsson
Associate Professor

Reykjavík University - Department of Computer Science

M.Sc. Project Report

EXPERIMENTS WITH AUTOMATIC WEIGHT TUNING IN HEURISTIC EVALUATION FUNCTIONS

by

Jónas Tryggvi Jóhannsson

Project report submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of **Master of Science**

January 2007

Project Committee:

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University, Iceland

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University, Iceland

Dr. Hannes Högni Vilhjálmsson
Assistant Professor, Reykjavík University, Iceland

The undersigned hereby certify that they recommend to the Department of Computer Science at Reykjavík University for acceptance this project report entitled **Experiments with Automatic Weight Tuning in Heuristic Evaluation Functions** submitted by Jónas Tryggvi Jóhannsson in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Date

_____

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University, Iceland

_____

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University, Iceland

_____

Dr. Hannes Högni Vilhjálmsson
Assistant Professor, Reykjavík University, Iceland

Abstract

Tuning weights coefficients in Heuristic Evaluation Function is a non-trivial task, often done by hand by game-playing program developers. The time and effort required to achieve good results has recently driven developers to utilize Reinforcement Learning techniques to tune these weight coefficients automatically. In this masters project, we have developed an general framework for automatic weight tuning. Secondly, we used the framework to conduct a comparisons study between the two main methods of learning from game data, TD(λ) and TD-Leaf(λ). A world-class chess program called Fruit was modified to enable learning within the framework, and then used to perform the experiments. The results from the study are interesting, as they show that although TD-Leaf(λ) is generally believed to be superior and is indeed more robust in respect to different parameter settings, TD(λ) can be just as effective when given proper parameter settings.

Útdráttur

Þetta verkefnið snérist um að læra betri vigtir fyrir matsföll (e. Heuristic Evaluation Functions), sem notuð eru í forritum sem spila borðleiki eins og t.d. skák, en til þess var notað skilyrt viðbragð (e. Reinforcement Learning). Verkefnið var tvíþætt; annars vegar að búa til almennt forritasafn til að auðvelda ferlið og umstangið í kringum það að læra vigtir fyrir matsföll. Hinsvegar var forritasafn þetta svo notað til að læra vigtir fyrir mjög sterkt skákforrit sem heitir Fruit-Chess með tveimur mismunandi lærdóms aðferðum sem eru nefndar TD(λ) og TD-Leaf(λ). Lýsingu á hönnun forritasafnsins er að finna í þessari skýrslu eftir að lesendum hefur verið kynnt nauðsynlegt bakgrunnsefni. Niðurstöður tilrauna og samanburð milli aðferðanna er svo að finna í niðurstöðukafla þessarar skýrslu.

Acknowledgements

I would like to start by thanking my supervisor Yngvi for being so patient during my countless moments of stupidity. Also, all the wonderful universities that I have studied at during my Masters program deserve a lot of credit: Reykjavík University, DTU in Denmark, and University of Alberta in Canada. On a more personal note, my brother Pétur deserves a lot of gratitude for his time spent on reading this report and giving me insightful comments. But most of all I would like to thank my girlfriend Steinunn María who has helped me so much with everything during my studies. Without her by my side I would never have the patience to go this far.

Contents

List of Figures

Chapter 1

Introduction

The history of computer chess is as old as the history of computing science itself, as the innovators of the field were already speculating on the success of chess programs in the beginning of the computing age. For example, Alan Turing who is often considered the father of computer science designed the first chess program in 1947 for his Turing Machine model of a computer. Claude Elwood Shannon, the father of information theory, wrote an influential paper on creating chess programs back in the 1950s [15] which along with John von Neumann's Minimax theorem [22] laid the groundwork for all modern chess playing programs.

Even though modern chess programs are still built on the same basic principles, they have come a long way since Shannon and Turing introduced their ideas. Computers started gaining the upper hand on humans at board games, ranging from backgammon and checkers to even more complex games such as chess towards the end of the last century. Today, there are numerous different chess programs that are unbeatable for most humans, requiring only the hardware of normal personal computers. The competition of the strongest chess entities is therefore shifting into the virtual world of programs, as the computer players are getting gradually stronger each year. In that virtual world there are two things that matter the most: (1) how fast the programs are, resulting in how many different games they can explore, and (2) how accurately the programs can predict the true outcome of the game using a heuristic evaluation function.

Writing a heuristic evaluation function however is a non-trivial task. Many different board configurations must be considered to decide which features of the chess board are important enough to be a part of the function and then their relative importance weights must be decided, which by itself is a difficult task. The weights are therefore often estimated by the programs developers and then tuned through months of playing experience against other programs. Having to rely on human intuition and months of tediously benchmarking their programs has driven developers into applying machine learning methods to automatically tune the weights of their evaluation functions.

Reinforcement Learning techniques have been applied to this problem successfully, mainly using two different learning methods called TD(λ) [16] and TD-Leaf(λ) [1]. In this project we develop a framework for automatic weight tuning and then use this frame-

work to do a comparison of these learning methods. The contributions of this project are twofold: first, a general framework for solving the tedious task of hand tuning heuristic evaluation functions is developed, and secondly a comparison study of the two previously mentioned learning methods is performed. The main result of the comparison study is that TD(λ) does perform just as well as the more complex TD-Leaf(λ) algorithm given the right λ parameter settings. This a bit surprising given that the latter algorithm was developed to overcome the inefficiencies of TD(λ).

The report is structured as follows. After this short introduction, Chapter 2 covers the necessary background material. In Chapter 3 we describe the design of the learning framework. In Chapter 4, the results of the comparison study is presented. Chapter 5 contains conclusions and discussions on some of the implementation issues and ideas that arose during the work on this project.

Chapter 2

Background

This Chapter covers the background material for this project. Chapter 2.1 describes how games are modeled as *game trees* and how these trees can be searched for the best move at each point in the game. Chapter 2.2 will then go on to explain how different game positions can be rated and given a numeric value by an *evaluation function*. This value is then used in the search to choose which move is the best at some given point in the game. Chapter 2.3 will explain how these evaluation functions are best viewed as *functional approximation* of the state space[1] of the game and how the *weight coefficients* of the evaluation function can be automatically tuned by using *Reinforcement Learning*.

2.1 Two Player Search

The board games where artificial intelligence has beens the most successful all fall into the same game theoretic category: perfect information two-player zero-sum games. These games, such as Chess, Othello and Checkers, have no stochastic elements such a dice rolls or hidden variables and they all have two players that are competing for a win, where one players gain are the others loss. This is why all high performance game-playing programs can all rely on *search algorithms* that make practical use of the Minimax theorem introduced by John von Neumann in 1928 [22].

To enable search algorithms to find good moves in a board game, the game must be explicitly modeled as a graph representing the state space of the game. This state space, sometimes referred to as search space, is the collection of all possible board configuration, or states, that can arise in that type of board game. Every state is represented by a node in the graph, where all the possible moves in each state are represented by *edges* to the successive nodes. The number of possible moves at a given node is called the *branching factor*. These graphs are then treated as search trees by the search algorithms, where the current state of the game becomes the root node of the tree, which is then expanded to each level by traversing the edges to each of the node's siblings.

[1] A state space in games is the collection of all the possible legal board configuration in a game

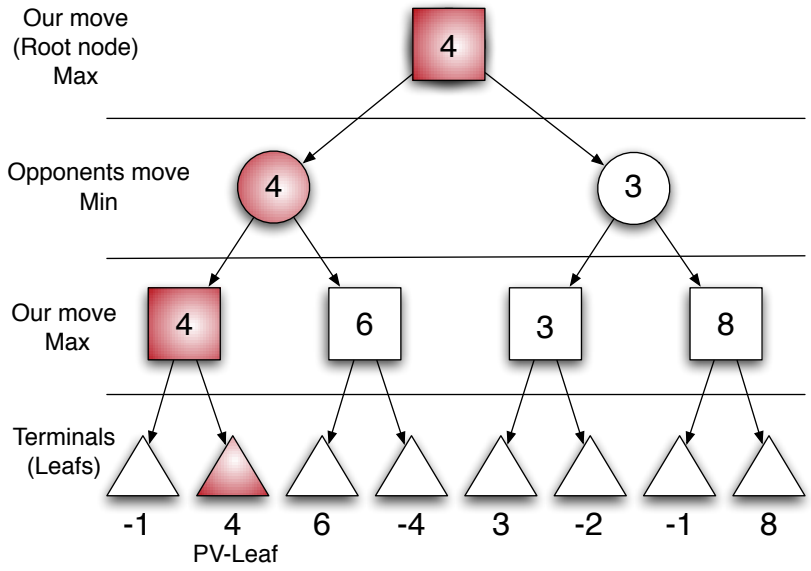

Figure 2.1: A backup diagram for a Minimax search, showing the *Principal Variation* (PV) in red, and the PV-Leaf. Note that the as the opponent is minimizing the values, he is in practice maximizing his own values as one players gain is a loss for the other.

2.1.1 The Minimax Theorem

Minimax search finds the best move possible at a given state by searching the whole game tree. A Minimax search tree, as shown in Figure 2.1, is constructed by the player that is about to move, expanding all nodes and their siblings recursively until it reaches a terminal node where no moves are possible. At a terminal node the game is over and the true outcome of the game can be observed and converted to a numerical value, such as -1 for losing, 0 for draws and 1 for winning. This value is then used to perform a backup on the tree, selecting the maximum value for the player that is performing the search but the minimum value for its opponent. The two players are therefore often called Min and Max, which stems from the name and nature of the Minimax theorem. The colored nodes in Figure 2.1 is the *principal variation* (PV), or the sequence of moves from the root node to the terminal leaf that is responsible for the root value after the backup has been performed. The PV is then the strongest line of play known at that point of the game, and the terminal of the PV is called the *PV-Leaf*.

Each Minimax search returns the indisputable result of the game, as every possible outcome of the game is explored. As the search is exhaustive, it follows that if the player is playing a perfect game then any mistakes made by the opponent can only lead to equal or better values. Constructing the whole game tree is only achievable in games with small state spaces as the game trees grow by a factor of b^d where b is the *branching factor* of the game and d is the depth of the search tree. Exhaustive search can solve simple games like Tic-Tac-Toe but will never return a value in more complex games, such as chess that has 10^{50} possible legal game positions with game length averaging 40 moves and an average branching factor around 35 [11].

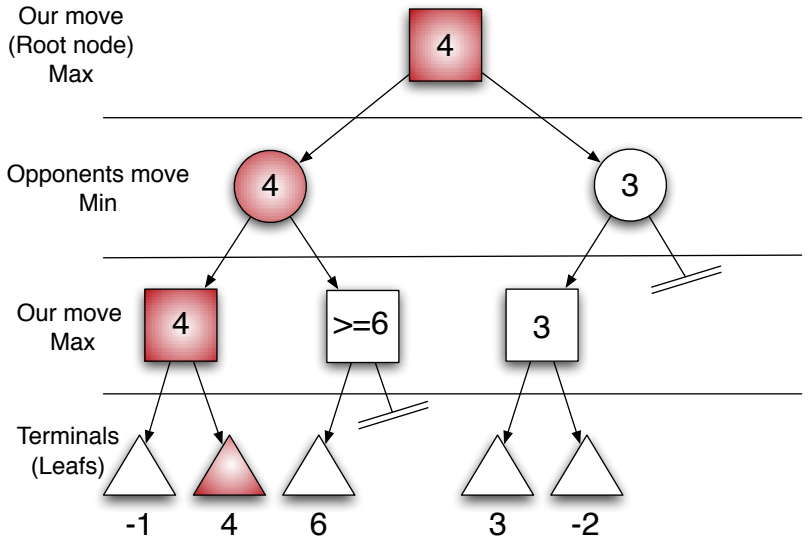

Figure 2.2: The pruned version of the Minimax tree in figure 2.1. On the left side, notice that when MIN has to minimize from 4 and a value that will not be smaller then 6, then there is no reason to expand the last node on the left branch. On the right branch we have an example of a *deep pruning*, as MIN can already choose the value 3 giving us no reason to explore that branch further as MAX will always choose the value 4 from the left branch. The colored nodes in the backup tree is the *principal variation* as described in Figure 2.1

2.1.2 AlphaBeta Pruning

AlphaBeta [8] search is a pruning version of a Minimax search, meaning that it does not search the entire game tree but only the parts that the players will be able to reach. Pruning is a technique that is made possible by the fact that a large proportion of the game tree are moves that are so beneficial for either of the player that the opponent will never allow those moves to be made, if there is a better alternative. For example; in chess, Black will never let White capture the queen unless faced with only worse options, so given that the black player has a better alternative then there is no reason to look deeper into that move.

This pruning technique cuts the search tree from b^d to $b^{d/2} = \sqrt{b^d}$ in the best case, by only pruning unnecessary parts of the game tree and without sacrificing any certainty of correctness of the backed up value. The search algorithms utilizing this pruning technique have been closing in on the theoretical best case running times [10] of the pruning technique, leaving little room for improvement in search techniques without sacrificing certainty and optimality by using probabilistic methods.

2.2 Heuristic Evaluation Functions

Search on its own is not enough to play any non-trivial board games because of the vastness of their state spaces. Therefore Minimax based search algorithms are typically coupled with a heuristic evaluation function to approximate the goodness of any non-terminal state. That allows the search to evaluate the merit of any node within the search tree, enabling search of partial game trees by using the evaluation values for the backup process. As the evaluation function provides the value for the backup process in the search, it has become responsible for what parts of the game tree are cut off by AlphaBeta pruning. It is therefore vital that the evaluation function returns a reliable and good heuristic that consistently reflects the true outcome of the game. Because of that, developers of game-playing programs often spend majority of their time on designing and tuning the evaluation function.

These heuristic evaluation functions serve as approximations of the state space, as learning a value for each state of the game is impossible as the number of states is too large. Evaluation functions are most often implemented using linear combinations which is explained later in this section. The approximation is composed of features of the game that the program designer believes to be relevant, and weight coefficients for each feature which indicates how much that feature should contribute to the total evaluation of the state. These features can be simple, such as number of pawns on a chess board, or difference of the sums of each player's piece values. But in some cases higher level features are used, such as analysis of the pawn structure, safety of the king or measurements of which player controls the center of the board.

2.2.1 Linear Combination

The previously described combination of features and weights is called linear combination, and is a simple yet powerful functional approximation technique commonly used in game-playing programs because of how efficient it is to compute. Given a set of features $\vec{\phi}$ and weights, $\vec{w}$, for each of the feature it can be calculated as follows:

$$f(\vec{\phi}, \vec{w}) = \sum_{n=1}^{N} \phi_n w_n \tag{2.1}$$

The value of a state is calculated as the sum of the numeric value of each feature ϕ, multiplied by its weight coefficient w. The gradient of linear combinations can be easily computed as it is simply the feature vector as shown in Equation 2.2. Another good feature of linear combinations is *locality*, meaning that only the features that contribute to the evaluation of of states will have a value in the gradient of the function.

$$\nabla f(\vec{\phi}, \vec{w}) = \left(\frac{\partial \phi_1}{\partial w_1}, ..., \frac{\partial \phi_{n-1}}{\partial w_{n-1}}, \frac{\partial \phi_n}{\partial w_n} \right) = \vec{\phi} \tag{2.2}$$

2.2.2 Gradient Descent

Gradient Descent is an optimization technique used to minimize the prediction error of a given weight vector $\vec{w}$. By use of calculus we can find the slope of the error for each dimension, called the *gradient vector* or simply the gradient, and minimize the error by traversing down the negative gradient slope of the error. The size of the step along the negative gradient is controlled by the prediction error δ of the function that is being minimized, in proportion to the step size parameter α which is set to a small number so the steps along the gradient do not overshoot the minima but kept big enough for the algorithm to converge at some point. The weights are updates as follows in each iteration;

$$\vec{w_t} = \vec{w} + \alpha\delta(-\nabla f(\vec{\phi}, \vec{w})) \tag{2.3}$$

This optimization method allows us to start out with some arbitrary vector $\vec{w}$, to approximate a state space, which is then gradually improved with each iteration until reaching a local minima. As Gradient Decent is a very efficient optimization algorithm, it serves as a the perfect companion for linear combination functions which have an easily computable gradient.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a field of research within AI where an *agent* learns by interacting with an *environment*. An agent can choose which *actions* to take, which will move him from one *state* to another. The agent receives a *reward* from the environment at any given point during its exploration though the environment, which indicates how well the agent is doing. The rewards are used by the agents to learn to predict its future payoffs, and in turn create a *policy* to maximize its future payoffs. This reevaluation of the policy which is performed by constantly updating the values of each action is the central idea of Reinforcement Learning. By that, RL is essentially trying to solve the *Credit Assignment Problem* [17] or the problem of identifying which state/actions should be given credit for the received rewards.

What makes Reinforcement Learning unique is that it does not require constant feedback during its training phase, as a periodic reward signal from the environment suffices for learning. This is why episodic problems, or problems that naturally break down into finite episodes with observable outcomes, can be easily modeled as Reinforcement Learning problems. For example: a particular board configuration in chess can be viewed as a state within the chess board environment. All the possible moves at that state are then the actions, and after each game we receive the reward value, which can be used to reevaluate the moves performed during the game. That reevaluation process is called a *backup* and should gradually improve the predicted final outcome of the game with more experience within the environment.

Reinforcement Learning can either be done in an *online* fashion, where the agent updates the value function at each step during the episode, or in an *offline* fashion where the agent

updates its values after each episode or even using large batches of episodes. Also, the process of constant reevaluation of the policy, called *policy iteration*, can be performed as *on policy* learning, where the agent uses its own episodes to improve its decisions, or as *off policy* learning where the agent uses external data to learn how to improve its policy by observing how other policies perform within the environment, e.g. learning *off* some external policy.

2.3.1 TD(λ)

Temporal Difference Learning, or TD(λ), is one of the fundamental Reinforcement Learning algorithm and was first used by Arthur Samuel in his checkers playing program in the year 1959 [12], but later formalized as a class of algorithms ranging from Dynamic Programming to Monte Carlo methods by Richard Sutton [16]. TD(λ)'s use in games took off with Tesauros TD-Gammon [20], which in 1995 became the strongest backgammon player in the world and even taught the strongest human players at the time new and better openings. TD-Gammons success sprang series of optimistic predictions for AI in games and drew attention to the use of Reinforcement Learning for the learning part of the game-playing programs.

TD(λ) learns how to improve its predictions of future rewards using the *temporal differences* between two subsequent states, often noted as the greek symbol δ. This is most usually done in an online fashion, but as the focus of this project is on the offline case of TD learning it will be demonstrated here in an offline fashion. In case of off-line learning, the prediction errors δ between subsequent states can be calculated for a whole episode and stored in a vector $\vec{\delta}$ as follows:

$$\vec{\delta}_t \leftarrow f(\vec{\phi}_{t+1}, \vec{w}) - f(\vec{\phi}_t, \vec{w}), t \in \{1..N\} \tag{2.4}$$

In this equation, $f(\vec{\phi}_t, \vec{w})$ denotes the approximation function for the current state at time t, calculated using the feature vector $\vec{\phi}$ and the weight vector $\vec{w}$. In the update process, the δ value for each state is propagated back to the history of previously visited states in proportion to the recency factor λ. This way of utilizing the temporal differences between subsequent states is then coupled with the gradient descent optimization technique in TD(λ) update algorithm;

$$\vec{w} \leftarrow \vec{w} + \alpha \sum_{t=1}^{N-1} \nabla f(\vec{\phi}_t, \vec{w}) \left[\sum_{j=t}^{N-1} \lambda^{j-t} \delta_j \right] \tag{2.5}$$

In the update equation, the weight vector is updated in proportion to the learning parameter α of the two sums. The first sum is multiplying the latter with the gradient of each state traversed during the episode, while the inner sum is propagating the prediction error (δ) of previous states scaled by the recency factor λ, which gets smaller as the difference between *t* and *j* increases. When λ is set to 0, then the prediction error of only the next state is used during the update, as done in Dynamic Programming algorithms. With λ set to 1, then updates are performed to the whole trajectory of states visited, as in Monte

Carlo methods. Setting λ to some value between 0 and 1 gives us an opportunity to use updating strategies ranging between these two classes of algorithms.

2.3.2 TD-Leaf(λ)

D.F. Beal and M.C. Smith experimented with learning piece values by using Temporal Difference Learning in 1997 [4], where they learnt piece values which were at least as good as those used in standard elementary chess books. One of the most interesting aspects of their work is that they use the features of the *principal variation leaves* rather than using the the features of the root nodes of each search made during the game. Independently, Jonathan Baxter et al [2] came up with the same strategy of applying temporal difference learning to game trees and coined the method as TD-Leaf(λ).

The strategy is that for every move in the sequence of moves made during the game, $x_1, x_2, ..., x_{n-1}, x_n$, we have a leaf node denoted x_n^l which is the node that was rated by the evaluation function $f(\vec{\phi}_n^l, \vec{w})$ using the features of the leaf x and the weight vector $\vec{w}$. When each game finishes we can view the true outcome and compare it with how this evaluation was predicting the outcome, and calculate the prediction errors ($\vec{\delta}$) in an offline manner like described in Equation 2.4. The weights are then updated as described in Equation 2.5. Instead of using the temporal differences and features of the root nodes of each search like TD(λ) would normally do, TD-Leaf(λ) uses the features and temporal differences of the PV-Leafs.

Neither Beal and Smith nor Baxter et al give an explanation of why they decided on using the features of the PV-Leaves in the learning process, but a plausible reason is that the search engines are often designed to use a stable node for evaluation so that there are for example no capture moves unexplored at the PV-Leaves. Using the features of the PV-Leaf can therefore be viewed as a strategy to improve the quality of the learning data.

2.4 Related Studies

As previously mentioned, Samuel's Checker player was the first major success of using Reinforcement Learning for learning in board games. Since then, the Reinforcement Learning techniques have formalized better and the algorithms have matured. There have been numerous successful attempts of applying these techniques to weight learning in numerous games. This section discusses the work that is relevant to this project.

2.4.1 KnightCap

KnightCap is the chess-playing program that Jonathan Baxter et al used when they implemented TD-Leaf(λ) [2]. It went from being an average rated player to a human master level in 308 games by playing against gradually stronger human players via Internet play,

which according to the authors was one of the key element of its success. KnightCap's use of TD-Leaf(λ) enabled it to learn from the features of the Principal Variation Leaves, instead of using the features of the positions that arose on the chess board during the game as would be the normal strategy when using TD(λ).

The importance of each of KnighCap's feature is different depending on what phase of the game it is, e.g. in the beginning and mid game we might care more about king safety while in the end we want the king to participate in the game. Therefore a weight is not a single number that can be learnt, but should rather be viewed as a set of vectors to be used in different phases of the game. This is done in KnighCap by dividing the game into a number of phases and learning different weights for each of them. TD-Leaf(λ) authors propose not to update the weights if their program does not predict the opponents move, unless they are certain that the opponent is a stronger player than the program. This is especially important against human players, because they are more likely to make dreadful mistakes in their games which would make the learner think it has found a good way to win.

They compared TD-Leaf(λ) to TD(λ) with $\lambda = 0.70$ where each method had equally deep searches for each move in the training data. They concluded that TD-Leaf(λ) learned faster than TD(λ), but the values learnt by both methods should be as efficient.

2.4.2 Weight Learning in Chinook

Chinook [14] started out as a research project in 1989 for a better understanding of heuristic search, but quickly became one of the strongest Checkers player in the work. It was created by Jonathan Scaeffer and his colleagues at the University of Alberta and it became the first computer program to hold a world championship in a board game in the year 1994. Since then it has been withdrawn from all competitions and is currently being used to solve the game of checkers.[2] As the strongest Checkers player in the world, with weights that had been manually tuned for a period of over five years, Chinook served as the perfect platform for testing how the weights that are learnt by Temporal Difference Learning perform compared to those tuned by experts.

Schaeffer and his colleagues implemented a TD-Learner [13] for Chinook that automatically tuned the weights by using self play against the tournament version of Chinook, where both versions used a six piece endgame database. To minimize the modifications to Chinook, they implemented the TD Learner as a separate program and kept all communications between Chinook and the TD Learner at a text file level. An opening book was used to prevent the programs from replaying the same game over and over again.

They concluded that the weights learnt in a few days by the TD Learner for Chinook were at least as good as those hand tuned for years by experts. The weights were learnt from no previous knowledge, using fixed depth search of 5, 9 and 13 plies. They also found that weights learnt from training data from shallow searches, such as 5 ply searches, did not work well when used for deeper searches, such as the 13 ply search.

[2] Solving a game means that for every possible state in the game the true outcome of the game will be known given that both players are playing a perfect game

Chapter 3

Framework

The purpose of the framework is to allow automatic tuning of evaluation functions as well as to serve as a platform for our comparison study. For the sake of generality, the framework was split into three different modules:

- **Game-playing program** which generates a log of the moves that it performs during the game. The program uses the learnt weights in its evaluation function.
- **Learning-data extractor** module that takes the logged games generated by the game-playing program, and creates learning data by replaying the games and extracting the relevant features of the evaluated nodes. This process is highly dependent on the game-playing program.
- **TD-Learner**, or Temporal-Difference learning module, that performs the tuning of the weights used in the evaluation function given the information extracted from the game logs.

The separation of modules into a game-playing program, learning-data extraction and the learning agent enables easy experiments with new learning algorithms or new means of learning-data extraction just by rewriting that module. The separation of the game-playing program and the learning data generation allows us to change learning strategies, from i.e. TD(λ) to TD-Leaf(λ), without having to modify the game engine. The communications between the modules is shown in Figure 3.1. It is done via files so the learning can be performed in either an iterative fashion, where the weights for the program are updated after each game, or in large batches where the learner goes though a number of games using the same weight vector and the new weight vector can be calculated from the batch.

3.1 Game-Playing Program

Chapter 2 already covered the workings of game-playing program, which can be summarized as a search engine that is heavily dependent on a reliable heuristic evaluation function. The main differences between game-playing programs for different kinds of games are their evaluation functions as many search engine techniques apply to different

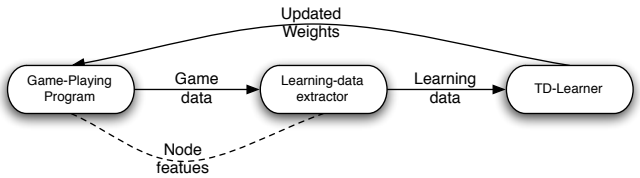

Figure 3.1: How the modules communicate via files: the game-playing program generates game logs in PGN format, which are converted to learning data for the TD-Learner, which creates better weights for the game-playing program.

```
[Date "2006.10.16"]
[Round "1755"]
[White "Learning"]
[Black "Fruit"]
[Result "1-0"]

1. Nf3 {(Ng1f3 Nb8c6 Nb1c3 Ng8f6 d2d4 d7d5 Nf3e5 Bc8f5 Ne5xc6
b7xc6 Bc1f4 Ra8b8 b2b3)} Nc6 {(Nb8c6 Nb1c3 Ng8f6 d2d4 d7d5
e2e3 Bc8g4 h2h3 Bg4f5 g2g4 Bf5e4 Bf1d3)} 2. d4 {(d2d4 d7d5
e2e3 Ng8f6 Bf1b5 e7e6 Nf3e5 Bc8d7 Bb5xc6 b7xc6 Ne5xd7 Qd8xd7
00 Bf8b4 Nb1d2)} d5 ...
```

Figure 3.2: This figure shows the format of PGN files used in the Framework, showing the first two and a half move of the game. Each move is numbered and followed by the Principal Variation inside comments.

board games. Because these programs follow the same basic principles, this framework can be utilized for weight learning in any of them.

There are a two of requirements that the programs must fulfill to enable the use of the framework:

First of all the game-playing program must be able to read in the learnt weights from files, to be able to utilize the updated weights from the TD-Learning module.

Secondly, they have to log their games in a standard format that can be read by the module that extracts the learning data. For example, chess playing programs typically use Portable Game Notation. Portable Game Notation (PGN) is a standardized notation for storing chess game data. It contains both game specific properties such as the players in the game, their ELO points, but more importantly it contains the moves that were made during the game. Each move is logged in algebraic chess notation, with an optional comment within braces after each move. The game specific information it needs, besides the moves during the game, are the names of the players to know if the outcome of the episode was a win or a loose for the learning agent, and also the result of the game which is used for sanity check during parsing. To enable TD-Leaf(λ) learning, the principle variation for each move must be logged as a comment by the learning player.

3.2 Learning Data Extraction

This module extracts the learning data from the game logs created by the game-playing program, to a file which contains each game as an learning episode for the TD-Learning module. The process of extracting data is dependent on the functionalities of the game-playing program, as game positions have to be recreated within the game-playing program to extract the features of nodes for that are used by its evaluation function.

The data extraction module replays the games from the logs, and polls the game-playing program for the features of the state that it is interested in. This is done by handing the game-playing program the state in some standard notation, like the Forsyth-Edwards Notation (FEN) notation in chess. It returns the features of the state as a numeric vector, where each element is the value of a particular feature. It does, however, mean that the data extraction module has to include an implementation of the game logic for the game which is being learnt in order to perform these replays from log files.

```
# Comment which will be discarded
# Values that are printed out;
# Mat - PawnStr - PiAct - KiSty - PP - Patt - PST
# w - weights that were used in the evaluation function
w -0.01 0.02 0.04 -0.02 0.05 0.03 0.1
# f - features of the nodes evaluated
#     'w' or 'b' tells us which player was evaluated
#     followed by the values of each feature
f w -967.5 0 -37.75 -69.75 0 -37.5 -15.25
f w -1000 0 0.328125 -42.4336 0 0 -29.5273
f w -2021.64 12.0898 94.625 0 -15.8203 0 31.0078
f w -2694.14 34.5703 -1.42969 -159.633 -22.8555 0 66.2813
f w -3186.72 -22.5391 81.1211 -97.7813 -24.7773 0 101.086
# r - result of the game
r -1
```

Figure 3.3: The format of the *Learning Data* output by the Log2LearningData module.

The module can both create data for ordinary Temporal Difference(λ) learning by writing out the features of the root nodes of the search, or it can create data for TD-Leaf(λ) learning by outputting the features of the terminal nodes of the principal variation for each move. The Principal Variation must then be included in the comments after each move in the PGN file.

One delicate part of creating the learning data is squashing the evaluation of states to a number in the range between $[-1..1]$. This has be done because the data generator uses -1 as penalty for loosing, 0 for a draw, and +1 in reward for winning. The squashing of the evaluation value has to be consistent with the outcome of the game, e.g. it can not always be close to 1 when the player has the upper hand but has to land somewhere on the interval in proportion to how good the state is. As the temporal differences are calculated using the squashed evaluation, then it is very important that this part is given some attention. This is best done in the game-playing program when it is polled for the features, as only

the developer of the evaluation function known the range of the values it can return and how the function really behaves.

3.3 TD-Learner

The TD-Learner is the Reinforcement Learning part of the framework, which takes the learning data that the Learning-data extractor produces as input and outputs a set of updated weights indicating the relative importance of each feature. The TD-Learning module has some adjustable parameters: the discount parameter γ that had to be included in testing the learner, the learning rate parameter α and the backup parameter λ, which can all be set in a configuration file that is read each time the learner starts. That enables a schedule for the parameters, allowing them to change over time, like lowering the learning rate parameter α after each episode so it converges to more stable values or starting off with a high λ backup parameter to generalize more when learning is starting off.

To ensure that the TD Learner was working as expected, it was given a set of simple standard testbeds to solve. These included *Random Walk*, a simple *Grid World* as a path finding domain and also the *Windy World* from Sutton's and Barto's book [17]. There were numerous refinements added to the learner during this process, such as enabling discounting, intermediate rewards and continuous tasks. The learner solved all these tasks and converged to the correct solution in all domains.

Chapter 4

Experiments

We used the framework for comparing the performance of two reinforcement learning methods used for learning heuristic evaluations functions in game-playing programs. The former, TD(λ), is a standard temporal-difference approach that learns from episodes generated from game positions occurring in played games. The latter method, TD-Leaf(λ), is based on an improvement proposed to the standard approach as especially suitable for learning in games. Essentially, instead of creating the learning episodes from the game positions occurring in the game, the episodes are instead generated from the position representing the leaf of the principal-variations from the game program's search process. Such episodes are considered to provide a more robust training data, because they are free from temporary imbalances that inevitably show up during a game. As an example, if one player sacrifices a knight seeing that it can be regained in a few moves, then that will not show up as a material imbalance in a principal variation episode, whereas it will in a game log episode. On the other hand, the drawback of TD-Leaf(λ) is that it can only learn from games where the principal variation information are available. Several experiments were performed, contrasting the two learning approaches.

4.1 Experimental setup

Use of the framework described in the previous chapter for conducting these experiments was straightforward, as the learning methods differ only in how the learning data is extracted by the Learning-data extraction module. Both methods use the same game-playing program and the same TD-Learner implementation. We test two different values for the generalization parameter λ. This parameter can take a value in the range $[0..1]$; the closer the value is to zero the more shortsighted the learning becomes, in the one extreme of zero the value of the current state is updated by looking only at the next state occurring in the learning episode. On the other hand, the closer the value is to one the more farsighted the learning becomes as more subsequent states one the episode are used for estimating the value of the current state.

As the framework is general, we had to find a game-playing program module suitable for these experiments. We decided on using Fruit, which is a high performance chess program

written by Fabien Letouzey [9]. The program finished second in the world championship of chess programs in 2005, which was held at Reykjavik University. Fruit was open source until version 2.2 when the author decided to close the source in order to prevent clones in tournaments and also to turn Fruit into a commercial product. Fruit 2.1, which was used in this project, is the last open source version released and works on both Unix and Windows. Its UCI compliant, which stands for Universal Chess Interface, so it can be used with many different chess GUIs.

In each experiment a learning version of Fruit, modified as described in Chapter 4.1.1, played against an unmodified Fruit 2.1. The chess engines were run within Arena which is a GUI for UCI compatible chess engines [6]. The programs were given 5 minutes of thinking time for each game, playing without any opening book nor endgame databases. The hardware used were Dell machines with 2.4GHz Pentium4 processors, 1GB of memory, and running Windows XP SP2.

Initial weights for the learning versions were set to zero, except for the weight value of the *Material Difference* feature which was fixed at the value 1 to ensure that all learnt values would be proportional to it. The learning parameter α was kept fixed at the value 0.01 while learning was performed in an incremental fashion, meaning that the weights were updated after each game to be used by the game-playing program in the next game.

To set a baseline for these experiments the modified version of Fruit, using the same weights as the original version, competed against the original version of Fruit. The modified version achieved 42% winning ratio in a series of couple hundred games which was an expected performance hit, as the modified evaluation function uses floating point numbers and has been somewhat simplified. Fruit 2.1 also competed against the modified version with all of its weights set to zero except the weight for material difference which was set to 1, where the modified version did not manage to win a single game. These results highlight that the performance of Fruit is affected by the modifications to the evaluation function that were necessary to enable learning. It also shows that its performance is greatly influenced by how the weights are tuned.

The searches are to a variable depth, where the computer is given 5 minutes of thinking time for each game. Similar projects using Reinforcement Learning for learning weights have been using a fixed depth and the results are reported to be only applicable to search to that fixed depth. Using variable search depth represents a more realistic scenario of learnt weights, as the search depths have to be variable in tournaments.

Endgame situations get a little cumbersome at times, where some situations that should have been a win for one side end up in a draw either because of the 50 move rule or the threefold repletion rule, which are both situations that the features in the evaluation function have no means of predicting. Because of this, learning is not performed when 50 move rule of threefold repetition arise as including those games as training data would only contribute noise to the learning experience. This can be solved using the Namilov endgame database, which is sadly not supported in Fruit 2.1.

4.1.1 Game-playing program: Fruit

There are several tunable parameters in Fruit which can be used to adjust the programs playing style. These are:

- **Material difference:** sum of the standard piece values; 1 for pawn, 3 for rook and bishop, 5 for castle, and 9 for the queen, subtracted with the piece values of the opponent, including a bonus for a bishop pairs.
- **King safety:** mixed features related to the king.
- **Pawn structure:** static pawn structure evaluation.
- **Passed pawns:** A pawn on a line with no opponents pawns in front of it, making it more likely to promoted.
- **Piece activity:** piece mobility and placement.
- **PST:**[1] piece on square table, an extension to material difference to rate pieces by their placement on the board.
- **Pattern:**[1] a feature trying to detect blocked or trapped rooks or bishops, to prevent that from happening.

Modifications to Fruit

First to fulfill its part as the game-playing program module in the framework, it was changed to use the user supplied weights which were read from file at startup. The evaluation function was also simplified slightly so that we could represent it as a linear function of the feature vector. The precision of Fruits evaluation function was also increased by using double values instead of integers. Fruits evaluation function calculates the features of the opening part of the game and the endgame separately, and then phases between those by calculating a phase factor which indicates how much has elapsed of the game. These features had different weights in original fruit, but now share the same learnt weights in the modified version. These modifications result in a modified version of Fruit that does not perform as well as the original version.

The second modifications was to enable the use of Fruit for extracting learning data in the *Log2LearningData* module. These modifications included replaying games from PGN logs, in order to reconstruct the exact chessboard configuration of the node that was evaluated by the heuristic evaluation function, so the features of the evaluation function of those states could be extracted from Fruit. The evaluation value of the state also had to be squashed to the range between $[-1..1]$, which was done by using $tanh(0.0025 * x)$ where x is the evaluation. The constant 0.0025 was estimated from the outcome of the evaluation function of some board positions where one player was surely gaining the upper hand. So

[1] These two parameters are not directly adjustable though the UCI interface, but are coefficients which can be found in its Evaluation Function

the evaluation function comes to:

$$tanh * (0.0025 \sum_{n}^{N} \vec{\phi}_n \vec{w}_n) \tag{4.1}$$

Where $N \in \{$Material Difference, King Safety, Passed Pawns, Piece Activity, Piece on Square Table, Pattern$\}$, and $\vec{W}$ are the learnt weight coefficients for each of the features.

Fruit seems to have a bug in its use of Forsyth-Edwards Notation (FEN) notation, as some of the states handed to Fruit in FEN were not consistent with the states reconstructed by using the PGN logs. This hindered the use of FEN notation in the implementation of the learning data extraction module.

4.2 Experiments with $\lambda = 0.70$

Baxter et al [3] used the λ value of 0.70 in the paper where they introduced the TD-Leaf(λ) method, giving no explanation other than it was a arbitrary chosen value. We performed similar experiments in our framework using TD(λ) and TD-Leaf(λ) methods with a λ value of 0.70. Our learning version of Fruit played a series of 2000 games against an unmodified Fruit 2.1.

The results for TD($\lambda = 0.70$) is shown in Figure 4.1. The figure shows that learning starts off slowly, but the winning ratio gradually improves as more games are played. The parameter weights are still increasing towards the end of the 2000 game series, indicating that the program is still learning although slowly. The agent achieves a 27% winning average for the 2000 games. The average winning ratio for the last 1000 games is 32%.

The results for TD-Leaf($\lambda = 0.70$) is shown in Figure 4.2. The figure shows that weights rise quickly during the first 300 games but then stabilize. The program achieves a 39% average for the 2000 games series. For the last 1000 games, these settings achieve a 42% winning ratio, which is equal to the baseline that was set for these experiments. Essentially, the program is able to learn just as effective parameters weights as the hand-tuned ones.

The TD-Leaf(λ) method is clearly a superior of the two, as it both learns faster and achieves considerable higher winning ratio. The parameter weights learned differ between the methods, which in part may be explained by how slowly TD(λ) is learning, not having converged yet. The order of the importance of each feature is however similar between the methods, with *passed pawns* rated the most important feature followed by *piece activity*. The conclusion here is that TD(λ) learns less effective weights in a much slower fashion, which is consistent with the results reported by TD-Leaf(λ) authors.

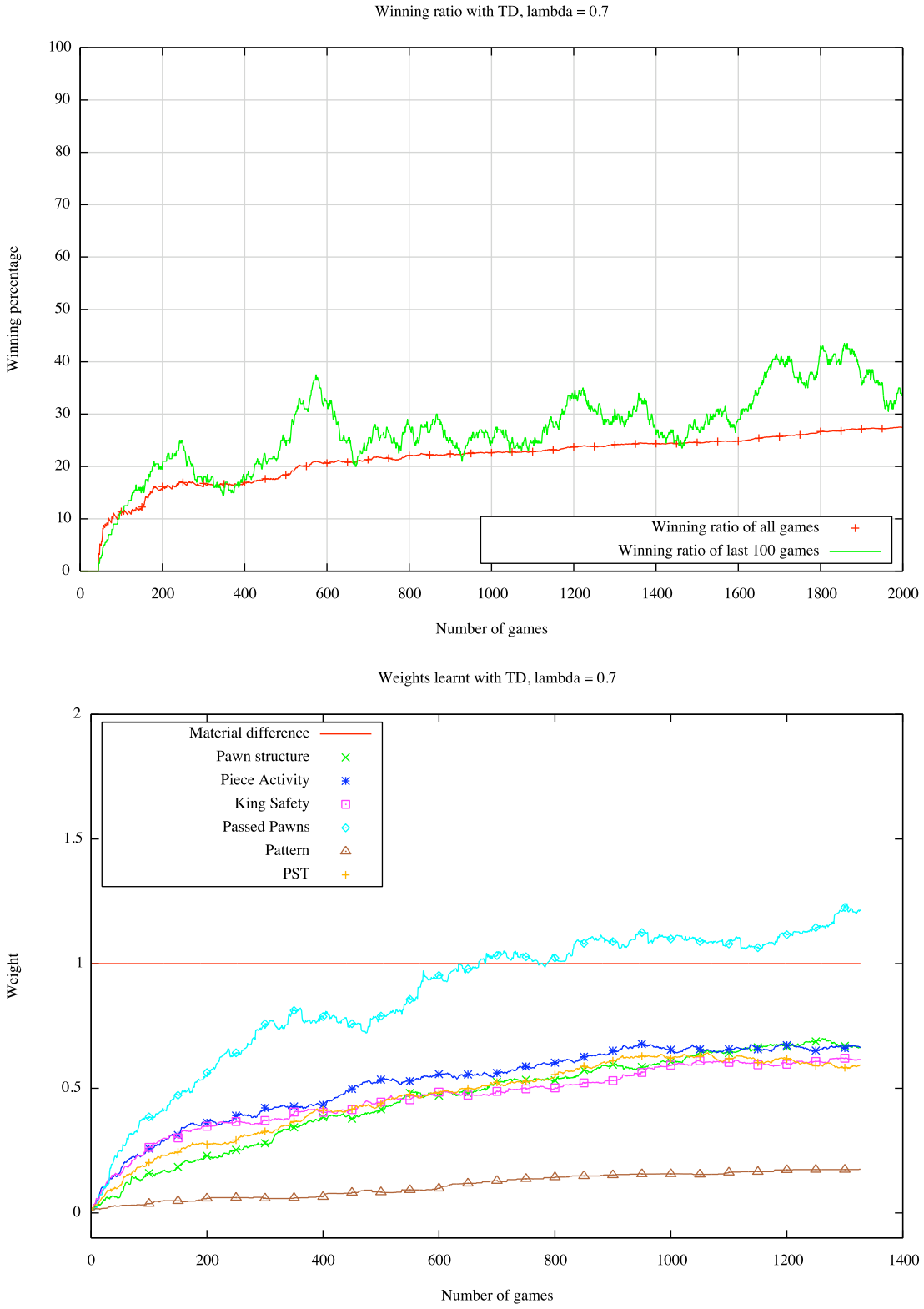

Figure 4.1: Winning ratio and weights learnt by TD($\lambda = 0.70$). Material difference is fixed at 1.

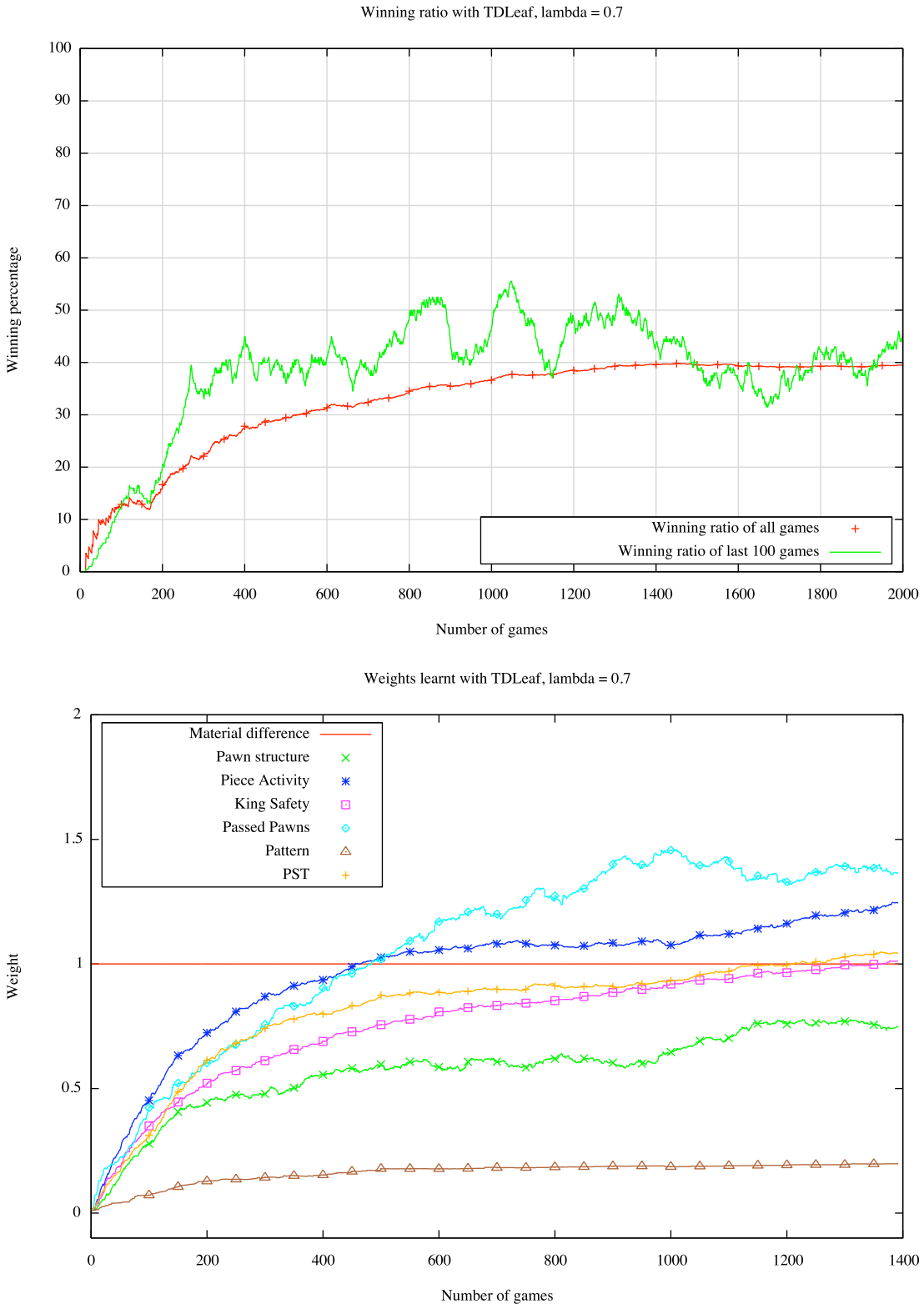

Figure 4.2: Winning ratio and weights learnt by TD-Leaf(λ = 0.70). Material difference is fixed at 1.

4.3 Experiments with $\lambda = 0.95$

When using TD-Leaf(λ) to learn weights automatically for the checkers world champion Chinook, Schaeffer et al [13] used the λ value of 0.95, but left the optimal setting of the parameter as an open question. To see how TD(λ) and TD-Leaf(λ) perform with this different λ value, we repeated the previous set of experiments with that setting of λ.

The results for the TD($\lambda = 0.95$) method are shown in Figure 4.3. The weights rise quickly for the first 400 games, but then stop improving. Each weight then fluctuates around the same value without quite converging. The reason for this is that higher λ values result in more course adjustment of the weights after each game. The winning ratio rises rapidly, reaching its overall winning ration 41% average in only 200 games, the same as for the last 1000 games.

The results for the TD-Leaf($\lambda = 0.95$) method are shown in Figure 4.4. The weights and the winning ratio rise quickly, achieving 37% average for the whole 2000 games (same for the last 1000 games). There is also high fluctuation of the weight values during learning.

With these settings there is no clear winning method as the learned weights, speed of learning, and the winning ratios are all quite similar. The weights learnt by the two methods have very similar relative weights and internal ordering of importance.

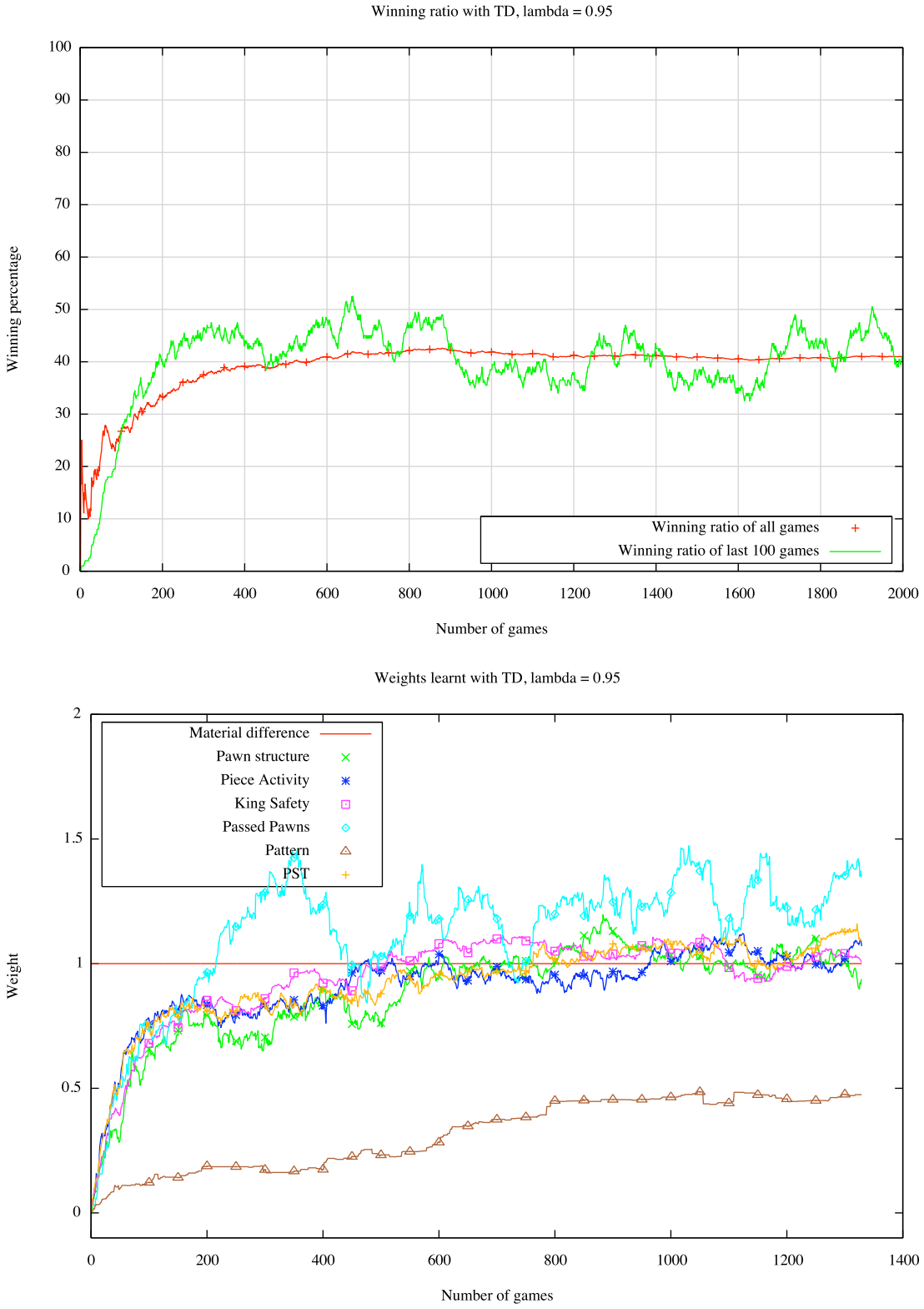

Figure 4.3: Winning ratio and weights learnt by TD(λ = 0.95). Material difference is fixed at 1.

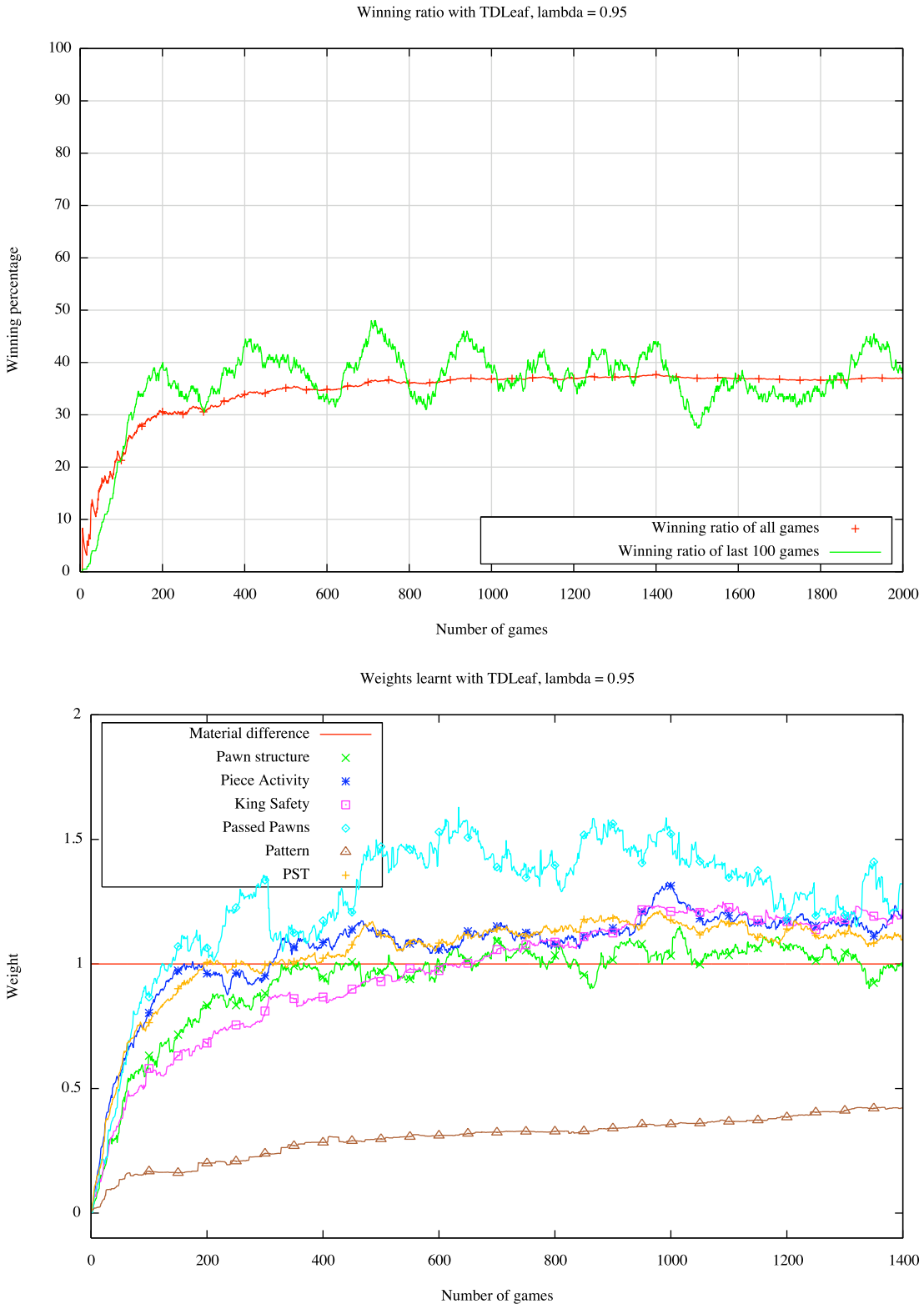

Figure 4.4: Winning ratio and weights learnt by TD-Leaf(λ = 0.95). Material difference is fixed at 1.

4.4 TD(λ) vs. TD-Leaf(λ)

The aforementioned experiments show that there is a substantial difference between the two learning methods and how well they perform with different values of the generalization parameter λ. TD-Leaf(λ) seems to be more robust with respect to that parameter setting, whereas the performance of the ordinary TD(λ) is more sensitive to the setting of the λ value. The difference in performance between the two TD(λ) learners, where TD($\lambda = 0.70$) performs considerably worse than TD($\lambda = 0.95$), is interesting and sheds light on how the two learning methods, TD(λ) and TD-Leaf(λ), differ in learning.

These different effects of higher λ values within the TD(λ) method is the result of how the method is applied to the root nodes of searches, using the temporal differences between each move that was made during the game. As the players can be in the midst of swapping queens or other kinds of piece scarifices yielding long term gains, some of these subsequent nodes can have different evaluation values that result in a high prediction error. TD-Leaf(λ) on the other hand is guaranteed be be using stable nodes in its learning process, as it is performed on the PV-Leafs which are guaranteed to be stable by the search engines which search deeper in case of instability of nodes that are to be evaluated.

Figure 4.5 brings more detail on the performance difference between TD(λ) with the two different λ values, as it shows just how differently the value of λ decreases. High values of λ cause the temporal differences of more states in the trajectory to be used in each update, while low values only utilizes the closest neighboring states and even to a lesser extent as λ drops off very quickly. The results of the experiments show that high generalization is vital for the TD(λ) method when it is applied to search trees, in order to minimize the effects of instability between subsequent nodes. In practice, greater generalization minimizes the effects of these inconsistencies in the update process. Therefore, high values of λ are necessary factor when using the roots of the search trees as the training data as ordinary TD learning does.

It is also interesting to see that even though TD($\lambda = 0.70$) performs considerably worse than its counterpart, the same λ value for TD-Leaf(λ) only starts off more slowly than $\lambda = 0.95$ but then achieves better overall winning ratio. This indicates that high levels of generalization is good to begin with, but should be lowered as learning progresses as it leads to more stable weights and stronger values given that the learning data is consistent. This is logical for an episodic task like chess, as some features may only be observed early in the game and can not be easily rated without propagating the prediction error for the endgame states back to the early stages of the game. This indicates that setting a schedule for λ where it decreases slowly over time may be good practice, as it allows the learning to attain decent values quickly and then minimize the fluctuation with lower λ values.

The winning percentage of the different learning variants against Fruit 2.1 are summarized in Table 4.1. We used the Students t-test to test how statistically significant the performance of the last 1000 games for the learning players differs from the 42% that our baseline version scored against Fruit 2.1. Essentially, we can state with over 95% statistical significance that TD(λ = 0.95) and TD-Leaf(λ = 0.70) perform as well as the baseline,

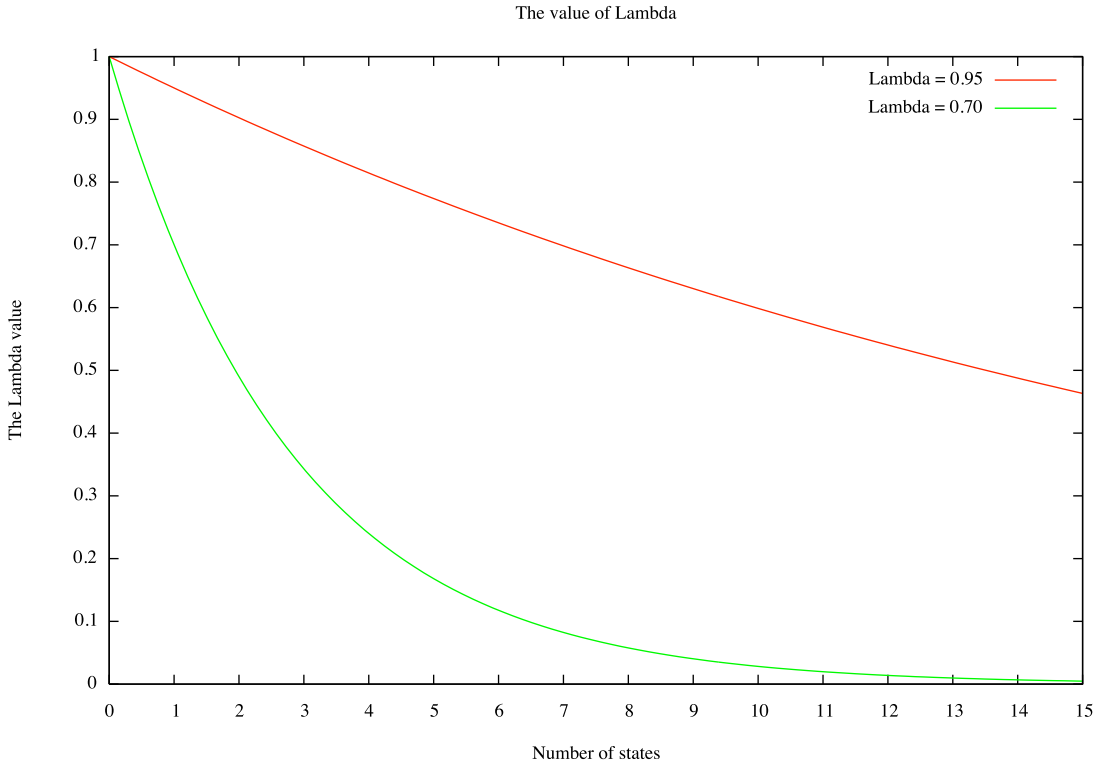

Figure 4.5: This figure shows how λ decreases for the two values: $\lambda = 0.95$ and $\lambda = 0.7$. Its value controls how much of the prediction error is propagated to the previous moves. With a value as high as 0.70, the speed of the drop off is still very fast so that neighboring states have almost no impact on the update value of states. However, with $\lambda = 0.95$ the prediction error of the neighboring states has a significantly more impact on the update value, minimizing the impact of unstable nodes in the learning experience.

Table 4.1: Average Winning Ratio for the whole 2000 episode of both learning methods with the different λ values.

λ / Method	TD(λ)	TD-Leaf(λ)	Difference
$\lambda = 0.70$	27%	39%	12%
$\lambda = 0.95$	41%	37%	4%
Difference	14%	2%	

indicating that equally effective weights were learned. We cannot state the same for TD(λ = 0.70) and TD-Leaf(λ = 0.95) with the same significance level.

Chapter 5

Conclusions

5.1 Conclusions

The experiments in the previous chapter contrasted two reinforcement learning variants, and showed that there is a real difference between the two learning methods and how they perform with different settings of the generalization parameter λ. TD-Leaf(λ) yields good results with both settings of λ, while the performance of ordinary TD(λ) is highly dependent on having a high value of λ to even out the effects of inconsistencies between unstable subsequent nodes.

These are interesting results in view of the fact that TD-Leaf(λ) is generally considered to be superior. However, we have showed that this is not necessarily the case. Given that TD(λ) can perform as well as TD-Leaf with careful parameter setting it may be advantageous to use it instead, because it can learn form game records that do not have principal variation information embedded.

This work leaves many questions unanswered. For example, the highlighted difference between TD-Leaf(λ) and TD(λ) shows that the learning is quite sensitive to the λ parameter. That raises the question if it is possible to detect how reliable the training data is and tune the λ parameter and/or the learning rate parameter α in proportion to the certainty, as an extension to the work introduced by Sutton and Singh [18].

It is desirable to be able to use all of the currently available chess games that have been recorded through the history and are available in archives on the Internet. To be able to do this, we would need do *off policy* learning using other players policies instead of performing *on policy* learning like done in this project. These games however do not usually have the PV recorded in the game data, especially in the case of human players, so we have to use ordinary TD(λ) learning. This project indicates that off policy learning should be possible with good results, as the winning ratio of the TD(λ) version of the learner is as good as the TD-Leaf(λ), and that using good training data is sufficient to achieve good results.

To be able to use external training data, we would have to modify an off policy learning algorithm such as Q-Learning to fit the Minimax search learning domain. This would

probably mean that we would only update the values when our search engine agrees with the data which moves should have been made in, similar to how KnightCap only updates when it agrees with the opponent moves unless it is competing against a stronger player. Another idea would be to set our λ and α parameters according to how certain we are that the move is correct based on results from the search engine, ELO points of the player and etc.

5.2 Discussion

Automatic tuning of heuristics evaluation functions is a difficult task. The generic framework developed for this task using reinforcement learning techniques was able to tune such functions to perform just as well as carefully hand tuned ones. This is by it self an achievement. At the time of this writing the chess playing program Deep Fritz has just won the reigning human chess world champion Vladimir Kramnik in a six game match [7], running on standard computer hardware. The success of the program can largely be credited to years of hard work by its developers, adding new chess knowledge and carefully tuning the program. It is interesting to see whether machine learning techniques can totally automate this process in the future, both with parameter tuning and maybe more importantly by automatic discovery of features.

Weight tuning has however not reach its full potentials, as automating the process opens up new possibilities. When weight tuning is done automatically, there is no reason to use just one or two weight vectors for the game when we can learn weights for much finer granularity. I experimented with learning a set of 128 vectors each for a different parts of the game, but after a week of training I decided that the results were not promising enough to keep on learning as the winning ratio was still under 10%. I still think that this is a good idea, that was probably crushed by the *curse of dimensionality* [5]. As this was done in the late part of the project there was not enough time to refine the idea, but I am however certain that this will become possible with better learning techniques.

References

[1] J. Baxter, A. Tridgell, and L. Weaver. Experiments in Parameter Learning Using Temporal Differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[2] J. Baxter, A. Tridgell, and L. Weaver. TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search. In *Proceedings of the Ninth Australian Conference on Neural Networks*, pages 168–172, 1998.

[3] J. Baxter, A. Trigdell, and L. Weaver. KnightCap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning*, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998.

[4] D. Beal and M. Smith. Learning piece values using Temporal Differences. *Journal of the International Chess Association, September*, 1997.

[5] R. E. Bellman. Adaptive Control Processes. *Princeton University Press, Princeton, NJ*, 1961.

[6] M. Blume. Arena Chess GUI. *http://www.playwitharena.com/*, 2006.

[7] Chessbase.com. Man vs machine shocker: Kramnik allows mate in one. *http://www.chessbase.com/newsdetail.asp?newsid=3509*, 2006.

[8] D. E. Knuth and R. W. Moore. An analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

[9] F. Letouzey. Fruit - pure playing strength. *http://www.fruitchess.com*, 2006.

[10] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. A New Paradigm for Minimax Search. Technical Report EUR-CS-95-03, Rotterdam, Netherlands, 1995.

[11] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

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

[13] J. Schaeffer, M. Hlynka, and V. Jussila. Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *IJCAI*, pages 529–534, 2001.

[14] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The World Man-Machine Checkers Champion. *AI Magazine*, 17(1):21–29, 1996.

[15] C. E. Shannon. Programming a Computer for Playing Chess. *Pholosophical Magazine*, 41(314), March 1950.

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

[17] R. S. Sutton and A. G. Barto. *Reinforcement Learning, An Introduction*. MIT Press, Cambridge, MA, 1998.

[18] R. S. Sutton and S. P. Singh. On step-size and bias in temporal difference learning. *Yale Workshop on Adaptive and Learning Systems*, pages 91–96, 1994.

[19] G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–277, 1992.

[20] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.

[21] A. Tridgell. Knightcap - Yet another free chess program. *http://samba.org/KnightCap/*, 2006.

[22] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press, Princeton, 1944. Second edition.

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

Department of Computer Science
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6201
http://www.ru.is