Using Reinforcement Learning in Chess Engines

Marco Block, Maro Bader, Ernesto Tapia, Marte Ramírez, Ketill Gunnarsson, Erik Cuevas[2], Daniel Zaldivar[2], and Raúl Rojas

[1] Free University of Berlin, Institut of Computer Science
Takustr. 9, 14195 Berlin, Germany
[block, bader, tapia, marte, ketill, rojas]@inf.fu-berlin.de
[2] Universidad de Guadalajara, Depto. Electrónica y Computación
Av. Revolución 1500, 44430 Guadalajara, Jalisco, México
[erik.cuevas, daniel.zaldivar]@cucei.udg.mx

Abstract. Up until recently, the use of reinforcement learning (RL) in chess programming has been problematic and failed to yield the expected results. The breakthrough was finally achieved through Gerald's Tesauros work on backgammon, which resulted in a program that could beat the world champion of backgammon in the majority of the matches they played. Our chess engine proved that reinforcement learning in combination with the classification of board state leads to a notable improvement, when compared with other engines that only use reinforcement learning, such as KnightCap. We extended KnightCap's learning algorithm by using a bigger and more complete board state database, and adjusting and optimizing the coefficients for each position class individually. A clear enhancement of our engine's learning and playing skills is reached after only a few trained games.

1 Introduction

The complexity of chess makes it impossible for computers to explore every possible move throughout the whole space of possible variants and pick the best one. Most chess engines therefore focus on a brute force strategy to search in the space of the next possible moves up to a certain depth only. Many pruning-processes are used, as well as linear position evaluation which incorporates knowledge based approaches in order to evaluate a special position. However, the main problem still lies in the correct tuning of the coefficients used in these functions. The method presented in this paper optimizes the evaluation functions and its coefficients by automating the use of temporal differences [2,3] and thereby increasing it's own understanding of chess after each game.

1.1 Related Work

Temporal difference was first tested in the program SAL by Michael Gherrity [5]. The structure of SAL allows the realisation of a move generation for different games and the determination of the best next move using a search-tree based algorithm. SAL learns good and bad moves from the played games. The evaluation of individual moves is performed using an artificial neuronal network. TD was used for the optimization of the networks parameters by comparing the evaluation values for the root nodes of the search tree. In a test against the prominent chess program GNUChess [18], where SAL was using 1031 position evaluation factors, 8 remis could be achieved in 4200 games (while the rest was lost).

The chess program NeuroChess, developed by Sebastian Thrun, also uses a neuronal network as position evaluation and a TD-method based on the root nodes to modify the coefficients. In contrast to SAL, NeuroChess only learns from itself. Games from a grand master database have mostly been used as entry points of the learning process (90%), while only 10% of the training games where played from the initial positioning. Later experiments with other programs showed that a learning strategy based on playing against oneself, does not yield satisfying results. In an experiment against GNUChess, where both programs where calculating a move depth of 2 and using the same evaluation, 316 out of 2400 games could be won by NeuroChess and the learned coefficients. Thrun, the main developer of NeuroChess admitted two fundamental problems of his approach: the large training time and the incompleteness of the evaluation coefficients. Thrun concludes that it is unclear whether TD-based solutions will ever find usage in chess programming.

Tesauro describes a better application of TD for the adjustment of the evaluation coefficients [8]. By using the open source chess program SCP he demonstrate that TD with a search depth of 1 will never yield good results. SCP works with an alpha-beta-algorithm with fixed search depth and the evaluation of 165 factors. The clear separation of the search and the evaluation procedure in SCP allows a stronger focus on the learning of evaluation factors by keeping a fixed search solution. A version of this algorithm has also been used at Deep Blue [12] to improve the king security in the game (1).

Tridgell implemented the chess program KnightCap [9,10], which has a parallel nature. The main idea was not to work on the root nodes, but rather applying the learning procedure on the best forced move-path of the search-tree. Small changes had to be made, e.g. storing all evaluation factors in a vector. Even the first experiment was a major success: in only three days and 308 games played on the Internet Chess Server(ICC) the rating of KnightCap increased from 1650 to 2150. The usage of ToPiecesBoard as an internal board representation allows an evaluation that can quickly identify complex patterns. The move selection algorithm MTD(f) [11] was chosen. A naive classification of the game situation into four different categories was made by the evaluation function: opening, midgame, endgame and mate positions. For each position class 1468 different evaluation factors were defined, so that KnightCap could evaluate a total of 5872 different coefficients. The version using a opening book reaches an ELO rating around $2400 - 2500$ and beats international masters ($2400 - 2600$ ELO points) on a regular basis.

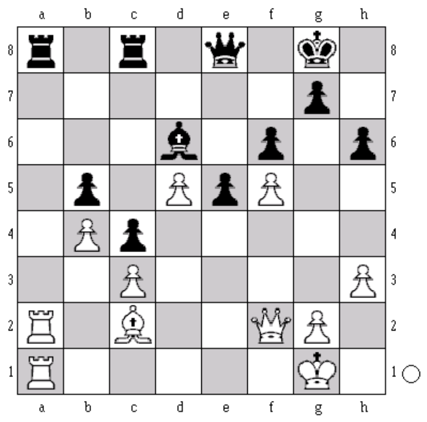


Fig. 1. Deep Blue versus Garry Kasparov, game 2 in 1997. At this position Deep Blue, using the normal coefficients, would have played the typical computer move Qb6 in order to win a pawn. Kasparov and the viewers were quit surprised by Deep Blues move Be4. This positionally strong move almost chokes every counter play and makes the threatening move Qb6 now even stronger. Kasparov eventually lost this game.

2 Reinforcement Learning and Temporal Difference

In this section we introduce a reinforcement learning method known as *Temporal Difference* (*TD*) and the learning algorithm TD(λ). This methodology was first introduced by Samuel in 1959 [13], but our discussion adopts notation and concepts first introduced by Sutton in 1988 [14]. Even though we explain the learning algorithm in the context of chess, it can easily be regarded as a general discussion about the methodology.

Let us denote with S the set of all possible states (chess positions) and with $x_t \in S$ the state at time t. The index t also means that x_t was obtained after t played "actions". For simplicity, we also assume that each game has a fix length of N moves. Each state x_t defines a set $A_{x_t}$of possible actions (legal moves). An *agent* selects one action $a \in A_{x_t}$ that produces a new state x_{t+1} from x_t to x_{t+1} with probability $p(x_t, x_{t+1}, a)$. The next state x_{t+1}denotes the position when both own and opponent moves have been performed. Hence only these positions will be examined, at which the agent can perform an action.

The agent will receive after each finished game a reward for the final move $r(x_N)$, which in chess takes the values 0 for remis, 1 for victory and -1 for a defeat. The expected value of the reward with an ideal evaluation function $J^*(x)$ is $J^* := E_{x_N|x} r(x_N)$. The main goal of the learning process is to approximate the unknown and (probably non-linear) evaluation function with a linear function $J' : S \times \mathbb{R}^k \to \mathbb{R}$:

$$J'(x, \omega) = \sum_{i=1}^{k} \omega_i J_i(x),$$

where $\omega = (\omega_1, ..., \omega_k)^T$is the parameter vector. This assumption reduces the problem to find ω the for the corresponding $J'(\cdot, \omega)$ that best approximates the function $J^*(\cdot)$.

The learning algorithm TD(λ) consist of an iterative update of the coefficients ω. Each iteration consist of playing a complete game with a fixed parameter ω to obtain a state sequence $x_1, \ldots, x_N$. Thus, the TD(λ)-algorithm computes the *temporal difference* d_t between the evaluations of consecutive positions x_t and x_{t+1} in the game:

$$d_t := J'(x_{t+1}, \omega) - J'(x_t, \omega)$$

Since the function $J'(x_N, \omega)$ for the last state x_N can be set to $r(x_N)$, we can compute the temporal difference for the predecessor state using $d_{N-1} = r(x_N) - J'(x_{N-1}, \omega)$. It is expected for any ideal evaluation function that if the evaluation of the state x_t at any time t is positive, then the outcome of the complete game will also be positive (that is, a victory). If the temporal difference is positive, then the actions of the agent improved. Since the player and the opponent played during the transition from x_t to x_{t+1}, it is possible that the opponent made an error. Therefore positive temporal differences are not taken into account, because they can degrade the approximation. Playing errors of opponent are not forced and should not be learned. If the temporal difference was negative, then the state x_t was not correctly evaluated. This evaluation needs to be degraded, since the chosen move turned out to be worse than expected. We therefore search the smallest change to the set of parameters holding the biggest effect towards this goal. The direction of the correction is obtained by computing the gradient $\nabla J'(\cdot, \omega)$. The strength of the correction Δt is defined as:

$$\Delta t := \sum_{j=t}^{N-1} \lambda^{j-t} d_j$$

The update of ω is performed is

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right].$$

The value Δt is the weighted sum of the differences in the rest of the game. A value $\Delta t > 0$ means that the position x_t was probably undervalued. Therefore a positive multiplier of the gradient is added to the vector ω, which will result in a better evaluation with the updated parameters. If $\Delta t < 0$ holds, then the position $x_t$was overestimated and therefore w will be updated with a negative multiplier of the gradient. The positive parameter α is the learning rate and will slowly converge stepwise after each learned game to 0. The parameter λ controls the contribution of the temporal difference from a position $x_t$until the end of a game. With $\lambda = 0$ no succeeding positions will be taken into account, and with $\lambda = 1$the complete state sequence influences the learning update. It has been shown that $\lambda = 0.7$ gives the most satisfying results [10].

In 1995 Tesauro published an article on the use of TD in his backgammon program. His solution trained the coefficients of the very complex evaluation function with the TD(λ)-algorithm. The performance of the program was astonishing. In its first appearance at the backgammon world championship in 1992 a version participated that had been trained with 800.000 games. From a total of 38 tournaments encounters, only seven were lost. The succeeding version which had been trained with 1.5 million games, lost only one of 40 games against one of the world strongest players, Bill Robertie.

2.1 Combining the Min-Max-Algorithm with Temporal Differences

Even though some learning approaches have successfully been applied to backgammon, they can not be applied to chess directly. There are fundamental differences between the two games. For example, small changes in the position of a backgammon game will result in only small changes of position evaluation. This represents a big difference in chess, where the main focus of the search lies in the chosen future tactic. More computational effort should therefore be spent in order to quickly analyze the search tree and make a prediction about the probable course of the game. Thus, a fast evaluation of the position is required, which makes the application of neuronal networks inadvisable.

A neuronal network seems suitable to correctly classify and cluster a current position into position types. Unfortunately, a small change in the position can result in a strong difference of the evaluation. This becomes obvious in chess programming, where two position differing in only one figures location, may need to be evaluated totally different. Hence a combination of forward computation in a search tree and the usage of a temporal difference algorithm TD(λ) seems to be logical solution.

The strategy used in the backgammon engine consists on the selection of the action a out of a position x, which minimizes the chances of the opponent to increase its evaluation:

$$a(x) := \arg\min_{a \in A_x} J'(x'_a, \omega),$$

where x_a denotes the position reached after the action α has been performed on x. Since the approach only to look forward a single move is not yet satisfying in chess, some modifications had to be made, in order to use a search procedure taking into note all future possible game situation until a specific depth. It seems reasonable that modified algorithm TD-Leaf(λ) will work on the best forced leaf nodes of the search tree rather then just working on the root node like TD(λ).

Let $J'_d(x, \omega)$ denote the evaluation value of the position x, when *forward looking* d steps starting at x. The modified temporal algorithm is now using a new temporal difference definition:

$$d_t := J'_d(x_{t+1}, \omega) - J'_d(x_t, \omega)$$

and a new the update step for ω

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'_d(x_t, \omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j\right]$$

In an experiment made by KnightCap, all evaluation coefficients but the material values (pawn 1, bishop and knight 4, rook 6, queen 12) were set to 0. Playing on the Internet chess server FICS the initial elo rating of 1650 of the program was computed in 25 non learning played games using this coefficient vector. In the fallowing three days and 308 played games with the usage of TD the rating could be raised to 2150.

3 State Classification

During a typical opening phase the figures bishops and knights should advance, the king castle, and the pawns assume control of the center. The midgame is most difficult to learn, since it can contain many different advantageous patterns such as the opening of lines, defense of open line with rooks, and attack possibilities on the opponent king. In other words, many typical constellations and short term goals are possible. In the endgame the king becomes a more important and active figure. It holds the opponent away from his own pawns, which are crucial due to their ability to transform into higher figures by reaching the endline. Most beginners are thought to correctly differentiate between these three position types, in order to avoid bringing the queen or king too early into the game. With further experience and game practice one quickly learns further criteria to distinguish between more position types in order to improve one's game evaluation. For example, opposing castle direction (large castle vs small castle) are most often followed by a tactical attack, which should result in a different evaluation of the position. Closed positions require a more strategical and long termed based plan. In chess programming these observations are often not taken into account. Chess-specific knowledge is not reflected enough in the implementation of a evaluation function. A main goal of the FUSc# development was to include the *Plan im Schach* (the chess plan) and give the chess engine the ability to search for its decision according to the current position type. Smaller, short termed plans exist, such as the exchange of a strong opponent bishop, the control of key fields, strong figure patterns and long termed plans such as a coordinated attack on the king side wind.

3.1 Chess Programming: Opening, Middle Game and End Game

Only a few open-source chess programs can be found that differ between position types beyond opening, mid, and end-game. A popular approach is to stress the importance of king security in the opening phase and evaluate it highly, while assigning a poor evaluation to the use of the center fields as target fields. This is done in order to prevent the king from going forward and instead bring him to safety on one side to allow the mid-game to start . In the endgame however, the king is a key figure which should be brought to the center fields. The same applies to the other figures. In the beginning development is important, and in the mid-game the control of the center and attacks on the opponent king.

In contrast, a method using TD-Leaf(λ) to optimize coefficients, succeeded in maintaining each position type represented in one game. On the other hand, shortcomings include the fact that on average the mid-game is over-represented, which results in very well learned mid-game coefficients, but others, like the endgame coefficients get left out.

The planed future approach of *FUSc#* is to use a grandmaster database to classify position types with the use of few important position properties, and evaluate these position types with their own coefficients. It is not possible to use TD-Leaf(λ)to achieve this, since some position types occur more often than others. This would mean, that for example, after 1000 games the position type x was optimized 400 times, but the position type y only 5 times.

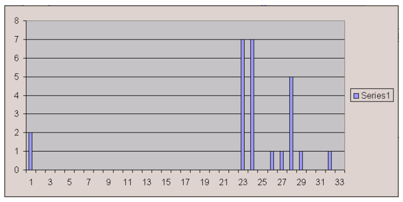
3.2 Classification Types of the Chess Engine FUSc#

FUSc# extends the basis set of position classes with the 32 combinations of these boolean rules: both queens on the board (yes/no), kings position (left, middle, right). Additionally a position vector for the endgame is added, which applies when no more queen is on the board and the sum of bishops, knights, and rooks is smaller then six or this sum is smaller then 3, when queens are still on the board.

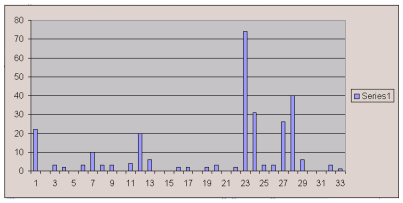
Only a few basic operations are required to decide which evaluation vector is chosen for its detected position type. For each of the 33 position types a respective vector containing 1706 coefficients is stored, to a total of 56298 adaptable position criteria. KnightCap used 4 different position classes throughout the training phase which each had1468 position criteria, differentiating 5872 coefficients.

4 TD-Leaf with Complex Evaluation

When using a very complex evaluation function, based on the differentiation of various position classes and therefore differentiating the evaluation coefficients of each class, a global learning rate α for the TD learning method would falsify the learning process. This is due to the fact, that the distribution of the position classes may not be normal and hence some classes will learn stronger than others, depending on their detection frequency in the best move search routine. Like many other chess programs, KnightCap only used three different position classes, so that the learning difference between these classes grew negligibly small, since most trained games had positions in each position class. The chess engine FUSc# deals with a wider set of position classes, so that it is likely to happen, that some classes will be detected significantly less often then others. A local learning rate solves this problem: for each position coefficient vector s_1, s_2, ..., s_ka local learning rate α_1, α_2, ... α_k is defined, that is individually adjusted depending on its detection accuracy.



(a)



(b)

Fig. 2. Distribution of the position classes. (a) The learning rate of the 33 position types after 7 played games. Initially the learning rate was set to 1.0. After each learning step the rate was decreased for each detected position type. The classes 23 and 24 have detected most often. (b) The distribution of the 33 position types detected in the 72 learned games. Position class 23 represents the opening phase.

The update function for the coefficient vector ω, containing all 56298 values of all partial position vectors $\omega_1, \omega_2, ..., \omega_{33}$ has to be adapted, so that the $\alpha_1, \alpha_2, ..., \alpha_{33}$ local learning rates are assigned to the position vectors:

$$\omega_k := \omega_k + \alpha_k \sum_{t=1}^{N-1} \nabla J'_d(x_t, \omega_k) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right],$$

where $k = 1, 2, ..., 33$ denotes the position class to the employed position x_t.

The difference to TD-Lear(λ)-algorithm used by KnightCap therefore lies in the usage of the data gained during a game and by the independent optimization of each position class detected in a trained game. Further improvements can be achieved by using information gathered during a game, for example the main variant and its computed position class values, and to use them in the learning process.

5 Conclusion and Experimental Results

To determine FUSc#'s performance, it was tested on human players in 50 games with different time settings ($1 - 5$ minutes time consideration). The evaluation on the chess server resulted in a strength around 1800 ± 50. A manually created vector was assigned to each position type. FUSc# thus did not. The initial values for the learning rates $\alpha_1, \alpha_2, ..., \alpha_{33}$ were 1.0 and λ set to 0.7. After a few games a small increase in performance was observable, since FUSc# was making first adaptions to the evaluation of the figure position and their effects on the game. One problem was that some position classes were not detected at all and therefore could not be learned, while others were found and learned regularly. The distribution of the 33 position classes after 7 played games is shown in Fig. 2.

At the end of the experiment FUSc# had played 119 games and increased its performance from 1800 to 2016 (see 3). The position classes 23 and 24 were found most often. The position type 23 represents an opening position, with both kings on their ground lines. Position type 24 is detected when the black player

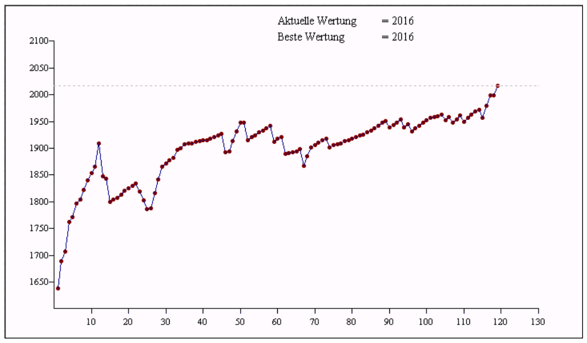


Fig. 3. Evaluation in the chess server after 119 games.

has made a short castle, 28 means that both players have made a short castle. From the 119 played games 72 have been used to adapt the coefficients and learn from the outcome of the game. The distribution of all detected position types is shown in Fig. 2b.

Improved performance was notably after playing only few games. It is crucial that all position types receive enough information for a correct adjustment of the coefficients. We estimate that FUSc# requires more then 50000 training games to correctly adapt all 56000 of the 33 position types and correctly learn them. This has not been verified yet. Looking at the results, it is clear that the optimization of the manually set values (all class types were set to the same initial values) increased the game quality remarkable. After 72 trained games the elo rating raised up to more then 2000. A main factor here was the king's security, which was correctly learned by FUSc#, activating the king in some of the position types and bringing him to the center of the board.

The successful usage of TD-Leaf(λ) in the chess engine KnighCap could be confirmed with FUSc#. Since the evaluation function of FUSc# takes into account many more position types, it needs a much greater sum of games to train with in order to achieve results comparable to KnightCap. Not even human players can become grand masters over night. It remains difficult to compare the learning progress of both programs, since KnightCap calculates its moves deeper then FUSc# and therefore has more success in finding tactical moves. Often, FUSc# appeared to be in a superior position and could have developed its strategy pretty good, but oversaw tactical implications and for this reason lost the respective games. To extend the automatization of the learning process the UCI-protocol needs to be changed. The chess engine should e.g. be informed about the outcome of a game and be allowed to perform a thinking phase, where learning parameters can be updated and allowing for learning to take place. Tests on the chess servers showed that the choice of the 33 different position types was a little unfortunate. e.g no differentiation was made whether both or just one queen was on the board or not. FUSc# classification does not take this information into account. Also some of the classifications occurred seldom or not at all. The opening phase needs some rethinking. At the moment only one vector is used to train it, but it seams that more diversified features are needed.

References

1. Zipproth S.: *"Suchet, so werdet ihr finden."*, CSS, p.15, 3/2003
2. Luger G.F.: *"Künstliche Intelligenz . Strategien zur Lösung komplexer Probleme"*, 2001
3. Zhang B.-T.: *"Lernen durch Genetisch-Neuronale Evolution: Aktive Anpassung an unbekannte Umgebungen mit selbstentwicklenden parallelen Netzwerken."*, Doktorarbeit, Infix 1992
4. Thong B.T., Anwer S.B., Nikhil D.: *"Temporal Difference Learning in Chinese Chess."*, IEA/AIE (Vol. 2), 612-618, 1998
5. Gherrity M.: *"A Game-Learning Machine"*, Dissertation, University of California, San Diego, 1993
6. Thrun S.: *"Learning To Play the Game of Chess"*, Advances in Neural Information Processing Systems (NIPS) 7, Cambridge, MIT Press, 1995
7. Baxter J., Triddgell A., Weaver L.: *"Experiments in Parameter Learning Using Temporal Differences"*, ICCA JOURNAL ,ISSN 0920-234X, Vol. 21 No. 2, pages 84-99, 1998
8. Tesauro G.: *"Comparison Training of Chess Evaluation Functions"*, Advances In Computation: Theory And Practice archive, Machines that learn to play games, pages: 117 - 130, 2001
9. Tridgell A.: *"KnightCap - a parallel chess program on the AP1000+"*, Technical Report, Australian National University, 1997
10. Baxter J., Tridgell A., Weaver L.: *"KnightCap:A chess program that learns by combining TD(lambda) with minimax search"*, In MACHINE LEARNING Proceedings of the Fifteenth International Conference (ICML '98), ISBN 1-55860-556-8, ISSN 1049-1910, Madison WISCONSIN, pages 28-36, 1998
11. Plaat A.: *"RESEARCH, RE:SEARCH & RE-SEARCH"*, Doktorarbeit, Tinbergen Institute, 1996
12. King D.: *"KASPAROW gegen DEEP BLUE"*, Beyer Verlag 1997
13. Samuel A.L.: *"Some Studies in Machine Learning Using the Game of Checkers"*. IBM Journal of Research and Development, 3:210-229, 1959
14. Sutton R.S.: *"Learning to Predict by the Methods of Temporal Differences"*. Machine Learning, 3:9-44, 1988
15. Tesauro G.: *"Temporal Difference Learning and TD-Gammon"*, Communications of the ACM, Vol. 38, No. 3, 1995
16. Rojas R.: *Theorie der Neuronalen Netze*, Springer-Verlag, 1993
17. T-Rex Webseite: *http://membres.lycos.fr/refigny63/nf3.htm*
18. GNUChess Webseite: *http://www.gnu.org/software/chess/*
19. NeuroChess Webseite: *http://satirist.org/learn-game/systems/neurochess.html*