האוניברסיטה העברית בירושלים
THE HEBREW UNIVERSITY OF JERUSALEM

AUTOMATIC LEARNING OF EVALUATION, WITH APPLICATIONS TO COMPUTER CHESS

By

AMIR BAN

Discussion Paper # 613 July 2012

מרכז לחקר הרציונליות

CENTER FOR THE STUDY OF RATIONALITY

Feldman Building, Givat-Ram, 91904 Jerusalem, Israel
PHONE: [972]-2-6584135 FAX: [972]-2-6513681
E-MAIL: ratio@math.huji.ac.il
URL: http://www.ratio.huji.ac.il/

Automatic Learning of Evaluation, with Applications to Computer Chess

Amir Ban

Abstract

A new and fast learning method is described in the context of teaching a program to play chess. A theory of the meaning of a position evaluation is developed, and is then confronted with a large collection of games played by masters or other programs. The program learns by fitting its evaluation to better predict the results of the games. The method has been employed by a top-rated program for the past 10 years, and has earned several world championships and successful matches against the world's best grandmasters for the program. The effectiveness of the method is demonstrated by showing its successful prediction of known playing strength of the programs.

1. Introduction

Teaching a computer to play chess is a classical problem in artificial intelligence research. Several of the founding fathers of computer science turned their attention to the problem at the dawn of the computer era, including Alan Turing [18] and Claude Shannon. In 1950, Shannon wrote a seminal article [13] on the subject.

Shannon proposed a *search and evaluate* procedure: The program would generate a tree of possible continuations from where it should select a move, first of positions arrived at by its own legal moves, then at lower depth of positions arrived at by possible opponent replies to its moves, and so on.

In principle this tree generation should end at leaves that are *terminal* positions (a win for either side, or a draw according to the rules of chess). Definite values may then be attached to the terminal positions, which may be back-propagated to the root position by the *minmax algorithm*, resulting in a root position value, as well as a best move, which may then be played. Had this been possible, the procedure would produce the game-theoretical result, and perfect play, for that particular root position. However the intractable myriad of possibilities in chess generally make such a conclusive determination unattainable.

Shannon proposed instead to cut off the tree expansion at some arbitrary point, so that the tree leaves are, in general, not terminal positions. A value must be attached to those for the back-propagating minmax algorithm to succeed. It is here that the need for an *evaluation function* arises. Shannon's suggestion was that the evaluation function would be based on weighing and tallying familiar

chess concepts such as the value of the pieces, the general structure of the position, pawn formation, mobility etc. He noted that any good chess player must be able to form such an evaluation, and that, while such an evaluation would not be perfect, **the stronger the player the better the evaluation**.

What *better* means in the context of evaluation was not spelled out. Shannon apparently was saying that better evaluation was evaluation that better fits the consensus of chess experts. Also implied was that better evaluation is one that would lead to better results in a program implemented as suggested in his paper.

All successful chess programs written in the past 50 years followed Shannon's framework. Great advances were made in search heuristics: the alpha-beta algorithm [9], transposition tables, the null-move pruning [4] and others. These, as well as our ever-improving computer hardware, have pushed the search capabilities of our best chess programs beyond the search capabilities of the strongest masters in almost all positions.

No such parallel advance has taken place in the domain of evaluation, for although program evaluations have greatly improved since the early days, no deeper meaning of evaluation has surfaced. Chess programmers still code expert chess knowledge into their evaluation functions, to the best of their understanding and capability, subject to its observed success in improving the results achieved by their programs (itself a time-consuming and statistically error-prone determination).

The field lacks a theory of evaluation which is able to suggest answers to any of the following questions: What is the meaning of the numerical value of the evaluation? What makes an evaluation right? In what sense can it be wrong? Between two evaluation functions, how to judge which is better?

It is the purpose of this article to suggest a theory of evaluation within which these questions and related ones can be answered. Furthermore I will show that once such a theory is formulated, it may immediately and economically be tested vis-a-vis the entire wealth of recorded chess game repertoire. As a result I will derive a novel automatic learning procedure that works by fitting its evaluation to recorded game results. While new to computer games, this method bears similarity to techniques of maximum-likelihood optimization and logistic regression widely used in medical and social sciences.

Some of the results I shall present in this paper are taken from real-world competition, rather than from models and controlled experiments: The methods presented here have been gradually developed within the chess program **Deep Junior** (a.k.a. **Junior**) since 2001, and all latter Deep Junior versions owe their strength and results in part to them. Deep Junior has competed at the top level of computer chess for the past 15 years, has won several computer chess world titles, and has played matches against the best human players.

The rest of this article is organized as follows: In Section 2 I will review learning methods used in computer chess and other computer games. Section 3 will be devoted to developing a theory of evaluation and an associated learning method. In Section 4 I will presents results and analyze performance. Section 5 contains concluding remarks.

2. Learning in Computer Games

Despite the popularity of the challenge posed by the game of chess to AI research, and notwithstanding the tremendous achievements of modern-day programs, attempts to equip chess programs with an automatic learning capability have had little application or success (In computer chess parlance, "learning" refers to the trivial action of demoting an opening line that has proved unsuccessful).

In other games, attempts at automatic learning were more fruitful. Samuel's checkers programs (e.g. [15]) was an early application of reinforcement learning. Temporal difference learning was successfully applied to checkers [11] and backgammon [16]. Temporal difference learning has also been applied to chess [3], but the playing strength of the resulting program was significantly lower than that of the best programs in the field. TD learning has also not had significant success with Go [12], a game comparable to chess in complexity. One problem has been that the credit assignment aspect has remained extremely difficult. Another is the unrealistically long training times necessary for games of high complexity. Wiering [19] and Thrun [17] each discussed the problems of applying TD learning to games as complex as chess, and Thrun proposed an EBNN (Explanation Based Neural Network) learning method instead.

As a rule, reinforcement learning methods work by having the program explore the problem space, using its own results to adjust its heuristics. A different class of methods relies on expert advice: Test suites are groups of positions, usually key positions taken from high-level master play, where the program must find the correct move, which is known. The program scores the number of correct moves it is able to calculate in a given time. Test suites have been constructed [10] that claim the existence of a formula calculating the solver's playing strength based on his results.

The programmer of the Falcon chess program, a participant in recent computer chess world championship, has used a "Mentor" method [5] in which the program learns evaluation from a stronger program by directly observing its evaluation and approximating its own to it.

3. The Method

3.1. Numerical Values of Evaluation

Developers and users of chess programs have adopted a more-or-less common parlance when discussing evaluation, which I adopt here:

A chess program's value of a position is commonly expressed in *centipawns*, i.e. a value of 100 denotes an advantage of a nominal pawn from white's point of view. A value of -100 denotes a disadvantage of a nominal pawn from white's point of view.

A win for white has the value of $+\infty$, a win for black, the value of $-\infty$, while a draw is valued at 0.

3.1.1. From Evaluation to Game Expectation

I shall interpret the evaluation result in any position in a game as a probabilistic forecast of that game's result. The game result expectation (with 1 counted for a white win, $\frac{1}{2}$ for a draw, and 0 for a black win) will be computed by the so-called *logistic function*:

$$E(v) = \frac{1}{1 + e^{-\frac{v}{b}}} \tag{1}$$

where v is the evaluation and b is a scaling constant.

Several arguments may be advanced for this choice of function:

1. It maps the evaluation space to the required interval of [0,1]. Furthermore, it satisfies expected identities:

$$E(+\infty) = 1 \tag{2}$$

$$E(-\infty) = 0 \tag{3}$$

$$E(0) = \frac{1}{2} \tag{4}$$

$$E(v) + E(-v) = 1 \tag{5}$$

2. Its quasi-exponential character is well suited to the additive nature of the evaluation function, mapping an additive element of the evaluation function approximately to a multiplicative factor in the game expectation.
3. It closely resembles the Elo formula used to calculate the rating of active chess players [6]. In fact, Elo's formula predicts that the expectation of a game between a player with rating A and a player with rating B is:

$$L_{A,B} = \frac{1}{1 + 10^{-\frac{A-B}{400}}} \tag{6}$$

which is equation (1) with $v = A - B$ and $b = \frac{400}{\ln 10}$. This similarity is natural and intuitive: Being stronger than one's opponent by a certain degree (in an equal position) is equivalent to having a certain advantage in a game (against an equally strong opponent), and vice versa.

3.1.2. From Evaluation to Result Probability Distribution

In carrying forward the interpretation of a game expectation to a probability distribution on the game results, a snag is encountered: The correspondence is not unique. For example, any probability distribution that attaches equal probability to a win and a loss has expectation of 1/2. Consequently, the position evaluation alone cannot provide a result probability distribution.

To overcome this, I introduce a *drawishness* factor α, which, like the evaluation, is a function of the position, and like it is to be calculated by the chess program's evaluation function.

Let Γ denote a position in a chess game. As we noted, the position has an ultimate game-theoretical outcome and value, which are however usually far

beyond computation reach. We therefore instead view the outcome as a random event[1]:

Define Ω to be the set of possible outcomes of a game: $\Omega \equiv \{Win, Draw, Loss\}$.

Define $Y(\Gamma, \omega)$ as the probability for outcome $\omega \in \Omega$ from position Γ. Furthermore define $Q(\Gamma)$ as the game expectation for white. Then the following necessarily hold:

$$Y(\Gamma, Win) + Y(\Gamma, Draw) + Y(\Gamma, Loss) = 1 \tag{7}$$

$$Q(\Gamma) = Y(\Gamma, Win) + \frac{Y(\Gamma, Draw)}{2} \tag{8}$$

Let $\alpha(\Gamma) \in [0, 1]$ be the *drawishness* of position Γ. For positive evaluations ($Q(\Gamma) \geq \frac{1}{2}$) set $\phi(\Gamma) = \frac{2(1-Q(\Gamma))}{2-\alpha(\Gamma)}$. Define Y as follows:

$$Y(\Gamma, Win) = 1 - \phi(\Gamma) \tag{9}$$

$$Y(\Gamma, Draw) = \alpha(\Gamma)\phi(\Gamma) \tag{10}$$

$$Y(\Gamma, Loss) = (1 - \alpha(\Gamma))\phi(\Gamma) \tag{11}$$

While for negative evaluations ($Q(\Gamma) < \frac{1}{2}$) set $\phi(\Gamma) = \frac{2Q(\Gamma)}{2-\alpha(\Gamma)}$, define Y as follows:

$$Y(\Gamma, Win) = (1 - \alpha(\Gamma))\phi(\Gamma) \tag{12}$$

$$Y(\Gamma, Draw) = \alpha(\Gamma)\phi(\Gamma) \tag{13}$$

$$Y(\Gamma, Loss) = 1 - \phi(\Gamma) \tag{14}$$

It is easily verified that these definitions satisfy (7) and (8). In conjunction with equation (1), equations (9)-(14) fully specify the result probability distribution given the position's evaluation and drawishness.

Note that a drawishness of 1 maximizes the expected probability of a draw. The percentage of draws decreases with the drawishness, becoming zero as it reaches 0.

The program calculates the drawishness based on the features of the position. Drawish positions, e.g. many opposite-color bishop endings, will have their drawishness set close to 1, while sharp positions, e.g. a king attack in the middlegame will set it lower.

I must emphasize that the drawishness, as defined in this section, is not a standard output of a standard chess-playing program (unlike the position evaluation, which is). Therefore the method being described here is not applicable to any off-the-shelf computer chess program: The program's evaluation function must be augmented by code and algorithms to compute a position's drawishness.

[1]The probability space for this random event may be defined, for example, as the set of game continuations from Γ by pairs of evenly-matched masters or computer programs.

3.1.3. Measuring the Evaluation Correctness

With an interpretation of the evaluation (together with the auxillary drawishness) as a probability distribution, it is possible to provide a numerical answer to whether any specific evaluation is "right" or "wrong", at least in the statistical sense:

Suppose a game has been played in the past between two opponents, whether human or computer. Select some position within that game and let the program calculate an evaluation for it. Is that evaluation right or wrong? The answer is the likelihood of the actual game result given result probability distribution derived from the evaluation. E.g. If the game ended in a white win and the program's evaluation predicts 0.7 probability of a white win, the evaluation is 70% right (and 30% wrong).

Definition 1. *Let the valuation of a position* Γ *be* $v(\Gamma)$ *and its drawishness* $\alpha(\Gamma)$.

Let $Q(\Gamma) \equiv E(v(\Gamma))$ *with* $E(\cdot)$ *defined in* (1).

Let the outcome of Γ*'s game (in some continuation) be* $\omega_0 \in \Omega$.

Let the result probability distribution for Γ *be computed from (9)-(14) resulting in* $Y(\Gamma, \omega)$ *with* $\omega \in \Omega$.

Then define the ***correctness*** *of* $(v(\Gamma), \alpha(\Gamma))$ *in view of* ω_0 *to be* $Y(\Gamma, \omega_0)$.

The correctness of the evaluation in view of the game result, as defined, is equal to the likelihood of the game result in view of the evaluation.

3.2. Method for Automatic Learning of Evaluation

Armed with the definition of evaluation correctness, based on the likelihood of game results as defined above, a method for the learning of evaluation suggests itself, based on logistic regression (e.g. [7]) and maximum likelihood optimization.

The major part of the method is an **algorithm for grading and comparing evaluation functions** based on their average correctness. It consists of the following steps:

Algorithm 2.

1. *Equip the learning program's evaluation function with code and algorithms that are able to compute the drawishness for any given position.*
   (a) *While it is possible to make only a* pro forma *effort in this regard, e.g. return a constant drawishness for all positions, the better the accuracy of the drawishness calculated by the program, the higher the grade (i.e. average correctness) scored by this algorithm.*
2. *Gather a large collection of chess games.*
   (a) *The games may be between any two players, computer or human, including games by the program itself.*
   (b) *Preferably, the games should be between strong players who are about evenly matched.*
   (c) *Let there be* n *games* $G_1, \ldots, G_n$ *with results* $\omega_1, \ldots, \omega_n \in \Omega$.

3. *From each game G_i in the collection, select a position Γ_i.*
   (a) *The position may be selected at random out of positions in the game. In general, the later in the game the move selected, the "easier" it is to predict the game's outcome (for a program that "understands" the game, in any case), and so the higher the evaluation correctness.*
4. *Have the program evaluate each of the n selected positions resulting in values for $v(\Gamma_i)$ and $\alpha(\Gamma_i)$.*
5. *Calculate the average correctness of the evaluation in view of the game collection as follows:*
   (a) *Select a scaling factor b for the evaluation and calculate white's expectation (see (1)) for each position as $Q(\Gamma_i) = \left(1 + e^{-\frac{v(\Gamma_i)}{b}}\right)^{-1}$*
   (b) *For each i use $Q(\Gamma_i), \alpha(\Gamma_i)$ to compute the result probability distribution $Y_i(\Gamma_i, \omega)$ with $\omega \in \Omega$ as given in (9)-(14). For each i derive the evaluation correctness $C_i = Y_i(\Gamma_i, \omega_i)$.*
   (c) *Calculate the average correctness as the geometric mean of correctnesses for all positions:*

$$\bar{C} = \sqrt[n]{\prod_{i=1}^{n} C_i} \tag{15}$$

Algorithm 2 enables the following learning procedure, based on incremental improvements verified by the grading algorithm:

Algorithm 3.
1. *Use algorithm 2 to calculate the average correctness of the baseline chess program.*
2. *Modify chess program with suggested improvement.*
3. *Use algorithm 2 to calculate the average correctness of the modified chess program.*
4. *Reject change if average correctness did not increase.*
5. *Accept change (i.e. adopt modified program as new baseline) if average correctness increased.*
6. *Repeat at step 2.*

This learning algorithm is not entirely possible to automate, as step 2 depends on the availability of successful improvements to the program, an availability that ultimately depends on the ingenuity and inventiveness of a programmer. Presumably the algorithm will stop when suggestions for improvement run out.

3.3. Discussion

The learning method presented harnesses a new resource to achieve its results: The history of other agents' games. In this it differs from various reinforcement learning methods in that it does not require the learning agent to explore the problem space. It is also different from expert-guided learning, in

that no vetting of the test data is needed, and indeed the source data is not assumed to be correct: The learning mechanism is agnostic regarding the correctness of play in the source games and does not even need to be aware of the moves played in the selected positions, let alone assume that they are correct.

An objection may be raised that the implicit assumption that the outcome of a game is determined by an intermediate position in the game is invalidated by a result-changing mistake by either player at a later stage of the game. While this is true, the possibility for errors is already encapsulated in the prediction of a probability distribution of game results. In addition, errors occur in both directions, so any first-order bias introduced will cancel, leaving at worst second-order biases. The requirement that the game be taken from high-class games is meant to minimize whatever effect mistakes have on the results.

Another possible objection is that the prediction of results from the position alone ignores the disparity in strengths between the players: The stronger player is more likely to win a level position, or save a bad one. Again this bias works in both directions, leaving only second-order effects. The requirement that opponents be about evenly matched is meant to minimize this effect.

The selection of a single position in each game is due to the need for independence between the individual correctness/likelihood values: Selecting two or more positions from the same game would factor in two or more predictions on the game result, introducing the tacit assumption that these may be different, but a game may only have a single result.

The described method is fast, and its running time is highly customizable: Modern chess programs are able to calculate and evaluate a large subtree in a fraction of a second. Even taking a game collection comprising tens of thousands of games, it is therefore possible to calculate the average evaluation correctness in several minutes.

4. Results

4.1. The Programs

The learning method as presented above, or forerunners of it, has been used in the development of Deep Junior since 2001. In that timeframe, 5 versions of the program have been released to the public (in the form of game CD's), and their performance has been rated by several voluntary organizations, e.g. [8]. The programs and their ratings are detailed in Table 1. In addition, the programs competed in the annual world computer chess championships of the International Computer Games Association (ICGA) and won several of them. The results of recent ICGA world championships are detailed in Table 2.

4.2. The Games

For the purpose of this study about 80,000 games played between May 2007 and August 2008 were downloaded from [2]. All games were played between various computer programs on state-of-the-art PC's at a time control of each 40 moves

Table 1: Ratings of Junior & Deep Junior versions

Version	Learning	Date	SSDF rating (K6-2 450 MHz)	SSDF rating (Athlon 1200 MHz)	SSDF rating (Q6600 2.4 GHz)
Junior 5	no	1998	2539	n/a	n/a
Junior 6	no	1999	2593	n/a	n/a
Deep Junior 7	partial	2001	2630	2693	n/a
Deep Junior 8	yes	2003	n/a	2764	2851
Deep Junior 9	yes	2004	n/a	2783	n/a
Deep Junior 10	yes	2006	n/a	2842	n/a
Deep Junior 10.1	yes	2006	n/a	2856	2980
Deep Junior 11	yes	2008	n/a	n/a	n/a

Table 2: World Computer Chess Championship Winners since 1996

Year	Place	Winner	Country
1996	Jakarta	Shredder	(Germany)
1997	Paris	**Junior**	(Israel)
1999	Paderborn	Shredder	(Germany)
2000	London	Shredder	(Germany)
2001	Maastricht	**Deep Junior**	(Israel)
2002	Maastricht	**Deep Junior**	(Israel)
2003	Graz	Shredder	(Germany)
2004	Ramat-Gan	**Deep Junior**	(Israel)
2005	Reykjavik	Zappa	(USA)
2006	Turin	**Deep Junior**	(Israel)
2007	Amsterdam	Rybka	(USA)

in 40 minutes. It should be noted that the games were played **later** than the release date of 4 out of 5 of the tested Deep Junior programs.

From each game a position was selected. The first 10 moves in the game for either side were excluded from the selection: These moves are often part of chess opening theory and played by chess programs from an opening database without calculation. Also excluded were the final 10% of the moves of each game: The game result is often apparent in the final moves, and the purpose was to exclude "easy" predictions. A position was selected at random from the remaining positions.

4.3. Measurements of Program Evaluation Correctness

The programs were set to evaluate each of the positions in the game collection. This was repeated for several time constraints, i.e. the subtree spanned and evaluated by the program was the largest that could be generated in several given computing times. To make the computing times comparable across different processors, the times were multiplied by the known performance of the local processor to form a processor-independent "processing power" index.

The results are tabulated in Figure 1. Some observations stand out:

- The average correctness increases with increasing processor power. Furthermore it is approximately linear in the logarithm of processing power. This is to be expected, and is in line with common observation in computer chess that a doubling of thinking time or of processor power leads to a roughly constant rating increase.

- The measurements of average correctness are in the range of 40%-48%, i.e. the predicted probability of the game result was on average 0.4-0.48. Note that a trivial evaluation function which predicts equal probabilities for the 3 possible results would score 0.33..., so the performances do not seem very impressive. It is worth remembering however, that only a fraction of a second of processor time was invested in each position, and that the average correctness is a geometric mean, which is lower than the arithmetic mean. Yet the results are a reminder that, as predictions go, there is great scope for improvement.

- Roughly, the results show a progressive improvement in the programs when arranged in chronological order.

Let us now comapre the measured evaluation performances with the actual performance of the programs, as rated by an independent association, SSDF [8], and tabulated in Table 1. To make the comparison, a common basis between an SSDF rating and an evaluation correctness is needed. I use the rough formula:

$$evaluation\ correctness = \frac{SSDF\ rating}{6000} \tag{16}$$

which then enables to combine the results in Figure 2, leading to additional observations:

- The program ratings are roughly linear extrapolations of the evaluation performance results.

- The measurements showing a similarity between the strength of versions 8 and 9, and of versions 10 and 10.1 are born out by the ratings, though SSDF rate version 10.1 slightly higher than version 10 while the evaluation performance measurements rate it slightly lower.

4.4. Effect on Style

Possibly the most intriguing result of the use of the current method has been on the playing style of the Deep Junior programs: It has become daring, attacking and disdainful of "material" in a way that is entertaining to human observers and often produces brilliancies. Commentators on Deep Junior have called [1] this style "speculative", "human", "understanding of the concept of compensation" and have likened it to the playing styles of former world champion Mikhail Tal and, by former world champion Garry Kasparov, to himself [14].

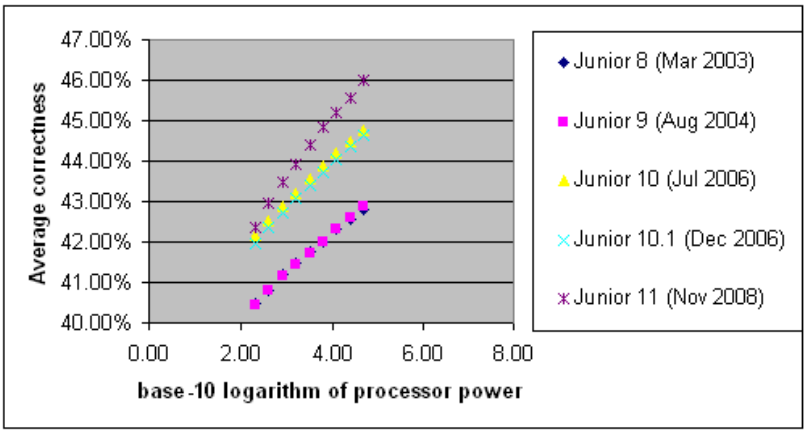

Figure 1: Evaluation performance of Deep Junior versions

4.5. What the Results Prove

At first glance the results prove that Deep Junior has managed to improve and to be at highest level in its field in the period that it used the method described. While this is true, the main point of the results lies elsewhere. It is in validating Algorithm 2: The average correctness is indeed strongly and positively correlated with playing strength.

Indeed, the results would be a strong validation of Algorithm 2 even had the programs tested *not* been developed with assistance from the described learning method.

In this context, it is worth repeating that the game dataset used to generate the results took no part in the development and learning of the programs: The games were mostly played *after* the programs were released. This rules out various forms of survivorship bias that could otherwise be argued against the results.

Note that once the effectiveness of Algorithm 2 is proven, the effectiveness of the learning algorithm itself (Algorithm 3) is self-evident.

5. Conclusion

In this paper I presented a novel learning method that has been successful at the highest levels of a problem of the highest complexity, as the game of chess is. The success of the method is demonstrated by the achievements of the programs that have employed it, by the experimental results presented in this paper, and, intangibly but very significantly, by the impression made by the playing style of the resulting programs on human observers.

In bare essence, the method relies on the fact that the game of chess is a single problem space, and that the score of a game played anytime and anywhere is sufficient to reconstruct the problem facing its players at any stage of the game.

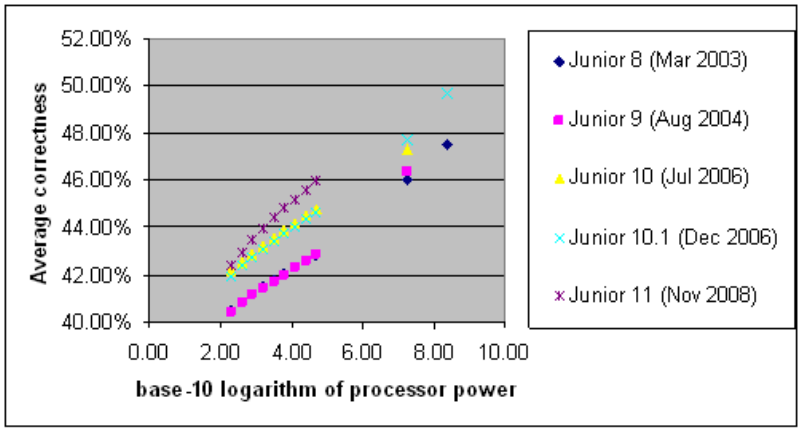

Figure 2: Correlation to ratings

In other words, a game may have been played in some tournament hall in Europe a century ago between already-forgotten masters, yet the game is relevant to the program's understanding of its own future games, and all the program needs to put itself in the shoes of the players at any stage of the game is the complete game score.

As such, the method may clearly be used to tackle other board games, such as checkers and backgammon. The game of Go, intensely difficult for computers, may be worth tackling with this method. The difficult incomplete-information games of Poker and Bridge may also profit from the method. Also applicable may be complex non-game scenarios such as securities trading, on the premise that some universal laws govern these fields, so that the past's experience is relevant to the present and future.

[1] A. Arganewal: The Most Aggressive Chess Program in the World, *http://www.chessbase.com/newsdetail.asp?newsid=2195*

[2] G. Banks et. al.: CCRL (Computer Chess Rating List), *http://www.computerchess.org.uk/ccrl*

[3] J. Baxter, A. Tridgell, L. and Weaver: Learning to play chess using temporal-differences. *Machine Learning*, **40(3)**, 243263, 2000.

[4] D.F. Beal: Experiments with the null move. *Advances in Computer Chess 5*, (Ed. D.F. Beal), pp. 65-79. Elsevier Science Publishers, Amsterdam, The Netherlands, 1989.

[5] O. David-Tabibi, M. Koppel, and N. S. Netanyahu: Genetic Algorithms for Mentor-Assisted Evaluation Function Optimization, *ACM Genetic and Evolutionary Computation Conference (GECCO '08)*, pp. 1469-1475, Atlanta, GA, July 2008.

[6] A.E. Elo: The rating of chessplayers, past and present. 1978, London: Batsford.

[7] D.W. Hosmer and S. Lemeshow: Applied Logistic Regression, 2nd edition, New York, Wiley, 2000

[8] T. Karlsson et. al. Svenska schackdatorföreningen (SSDF) *http://ssdf.bosjo.net/list.htm*

[9] D. E. Knuth and R. W. Moore: An analysis of alpha-beta pruning. *Artificial Intelligence*, **6** 293326, 1975.

[10] D. Kopec and I. Bratko: The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess *in Advances in Computer Chess 3, M.R.B. Clarke (ed.)*, Pergamon Press, 1982, 57-72.

[11] J. Schaeffer, M. Hlynka, and V. Jussila: Temporal difference learning applied to a high-performance game-playing program. *Proceedings of the 2001 International Joint Conference on Artificial Intelligence*, 2001, 529-534. Seattle, WA.

[12] N.N. Schraudolph, P. Dayan and T.J. Sejnowski: Temporal difference learning of position rate. evaluation in the game of Go. *Advances in Neural Information experts, Processing Systems* **6**, 1994, 817824. Morgan Kaufmann, San Mateo, Calif.

[13] C.E. Shannon: Programming a Computer to Play Chess, *Philosophical Mag.* **41**, (1950), 265-275.

[14] D. Sieberg: CNN interview: Kasparov: 'Intuition versus the brute force of calculation' *http://edition.cnn.com/2003/TECH/fun.games/02/08/cnna.kasparov*

[15] R.S. Sutton and A.G. Barto: Reinforcement Learning, An Introduction, MIT Press, Cambridge, Mass., 1998.

[16] G. Tesauro: Practical Issues in Temporal Difference Learning, *Machine Learning*, **8(3-4)** 257-277, 1992.

[17] S. Thrun Learning to play the game of chess. *In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems* **7**, Cambridge, Massachusetts, 1995. The MIT Press.

[18] A.M. Turing: Faster than Thought, B.V. Bowder (ed.), Putman, London (1953), 288-295.

[19] M.A. Wiering: TD learning of game evaluation functions with hierarchical neural architectures. Masters Thesis, University of Amsterdam, 1995.