Chess Neighborhoods, Function Combination, and Reinforcement Learning

Robert Levinson and Ryan Weber
University of California Santa Cruz
Santa Cruz, CA 95064 U.S.A
levinson@cse.ucsc.edu, weber@cse.ucsc.edu

Abstract. Over the years, various research projects have attempted to develop a chess program that learns to play well given little prior knowledge beyond the rules of the game. Early on it was recognized that the key would be to adequately represent the relationships between the pieces and to evaluate the strengths or weaknesses of such relationships. As such, representations have developed, including a graph-based model. In this paper we extend the work on graph representation to a precise type of graph that we call a piece or square neighborhood. Specifically, a chessboard is represented as 64 neighborhoods, one for each square. Each neighborhood has a center, and 16 satellites corresponding to the pieces that are immediately close on the 4 diagonals, 2 ranks, 2 files, and 8 knight moves related to the square.

Games are played and training values for boards are developed using temporal difference learning, as in other reinforcement learning systems. We then use a 2-layer regression network to learn. At the lower level the values (expected probability of winning) of the neighborhoods are learned and at the top they are combined based on their product and entropy.

We report on relevant experiments including a learning experience on the Internet Chess Club (ICC) from which we can estimate a rating for the new program. The level of chess play achieved in a few days of training is comparable to a few months of work on previous systems such as Morph which is described as "one of the best from-scratch game learning systems, perhaps the best" [22].

Keywords: linear regression, value function approximation, temporal difference learning, reinforcement learning, computer chess, exponentiated gradient, gradient descent, multi-layer neural nets.

INTRODUCTION

For years, researchers have sought a model of computer chess play that was similar to that used by humans. In particular, we are assuming a model of humans applying patterns learned from experience as opposed to employment of deep brute-force search as in the top chess systems. In addition, attempts have been made to create a chess program that is both autonomous and adaptive, while still being competitive. A wide range of machine learning techniques have also been tested, but most have met with limited success, especially when applied to playing complete games. Brute-force search and human supplied static knowledge bases still dominate the domain of chess as well as other more complex domains. Therefore we have attempted to create such a system, which conforms more to cognitive models of chess. Our research will not be completed until we can compete effectively against the best programs. We believe the research presented here represents a significant step in that direction compared to other previous attempts.

Our approach can be divided in three main parts that will be discussed in detail throughout this paper. First, since the representation is the foundation of the learning system, we have spent a lot of time developing a representation which can be processed efficiently and provide an even balance between detail and generalization. Secondly, we focus on assigning appropriate evaluations in the range [0,1] to board positions. Our system steps back through the temporal sequence of moves in a training episode to assign credit to each of the board states reached, using Temporal Difference learning [24]. This relies only on the given knowledge that a win is worth 1 and a loss is worth 0. This credit assignment can be very difficult, given little a priori knowledge. Finally, once the system has assigned evaluations to all the positions in a game, the next step is to update the internally represented weights of the global optimization function, to predict more accurately in the future based on the loss incurred between the prediction $\hat{Y}_t$ and the result Y_t. This is achieved with the use of internal interconnected nodes with associated weights in a 2-layer regression network. Multiple layers of representation are very important for representing complex domains, but they are often difficult to deal with in practice. In many of our experiments, a multiplicative update on the upper level and non-linear combinations of input vector products at the lower level perform best. However there are still many additional modifications that must be made to the standard feed forward neural network to achieve adequate performance within reasonable time constraints. For the first time, our system has achieved a level of play, which is competitive against skilled amateur opponents after only about 1000 on-line games. One of the most difficult hurdles confronting previous chess learning systems was getting an agent to avoid losing pieces without explicitly representing material. In this paper we have largely solved the material loss problem with this combination of modeling and learning techniques.

Section 1 gives some background on previous work. Section 2 describes the temporal difference learning algorithm in some detail, especially concerning its application to chess and game theory. Section 3 gives a general discussion of various linear and non-linear function approximation schemes. Section 4 ties this into the multi-layer network approach. Then Section 5 brings all of these elements together in describing our system's overall learning architecture. Section 6 presents the results of our experiments with off-line training on Grandmaster databases, on-line ICC, and bootstrap learning.

PREVIOUS EFFORTS

Since Samuel's checker playing program [21] there has been a desire to use TD methods for more sophisticated game-playing models. Tesauro's TD-Gammon represented a significant achievement in the fusion of neural networks and TD learning. Baxter and Tridgell created a program named KnightCap [2] for chess, which can improve the initial weights of a complex evaluation function that included other positional aspects besides simply material values. However, this approach was quite limited since it required good initial weights, human-supplied traditional chess features, and a knowledgeable opponent instead of relying on bootstrap learning. Morph, Korf and Christensen [9], and Beal and Smith [5] were also quite successful in applying temporal difference learning to learn piece values with no initial knowledge of chess and without the aid of a skilled adversary. Beal and Smith also report similar findings for Shogi [6] where piece values are less traditionally agreed upon. Another type of chess learning is rote learning in which exact positions that have been improperly evaluated are stored for future reference [23].

"At the time Deep Blue defeated the human World Champion Garry Kasparov, many laymen thought that computer-chess research had collapsed," until the publication of an extended report by "Ian Frank and Reijer Grimbergen [showed] that the world of games is still thrilling and sparkling, and that the game of chess is considered as the math reference point for many scientific and organizational problems, such as how to perform research, how to improve a search technique, how to handle knowledge representation, how to deal with grandmaster notions (cognitive science), etc" [13]. The Morph system uses a more complex model than previous approaches, which makes its goals "the central goals of artificial intelligence: efficient autonomous domain-independent machine learning for high performance" [22]. It uses graph representations of chess positions and pattern-oriented databases, in conjunction with minimax search to evaluate board states [5]. This approach can be limited in large domains since the number of patterns can become too large. Morph III and Morph IV [15,17] used nearest neighbor and decision trees to divide positions into equivalence classes and query them on-line in logarithmic time. However these approaches require a large amount of training data to achieve reasonable levels of play. "Morph is arguably the most advanced (not necessarily the strongest) temporal difference learning system in the chess domain…However, a major problem of Morph is that although it is able to delete useless patterns, it will still be swamped by too many patterns, a problem that is common to all pattern-learning systems” [10] (Fürnkranz, pg. 10). Compared to Morph I, we have shifted away from a bag of patterns towards a representation that stores fixed numbers of internal weights. This is more efficient in both space and time, and can lead to faster global optimization. Currently, to our knowledge, there is no completely adaptive autonomous program that is actually competitive on a tournament level.

2.TEMPORAL DIFFERENCE LEARNING

For adaptive computer chess, the problem is learning the values of board states or at least being able to accurately approximate them. In practice, there is seldom an exact actual value for a given model state but instead there may be an episodic task where an entire sequence of predictions receives only a final value. The actual values are typically approximated using some form of discounted rewards/penalties of future returns. This is the basic idea behind TD(λ), which has been successfully applied to many applications in game theory and others in machine learning. Temporal difference learning was originally applied to checkers by Samuel [21] but it has been less successful in more complex domains. KnightCap’s variant of TD(λ) called TDleaf(λ) is used to evaluate positions at the leaves of a minimax search tree. The value of λ in TD(λ), provides a trade-off between bias and variance. Tesauro also successfully incorporated TD learning for backgammon, with a set of real valued features at the lowest level of representation. This differs from our model in the sense that there are no obvious numerical representations at the base level, so we learn a set of weights for each base feature. Our lowest level of representation is discussed further in Section 5.2.

The general structure of the TD learning paradigm, as described by Sutton [24], is based on learning the policy value function $V^{\pi}(s)$ given a sequence of predictions s, and their associated reinforcement signals r. Our system uses TD(0), which simplifies the update rule to

$$V(s_t) = V(s_t) + \alpha\ [r_t + \gamma V(s_{t+1}) - V(s_t)], \qquad (2.1)$$

where α is the learning rate parameter and γ is the discount rate.

3. LINEAR AND NON-LINEAR OPTIMIZATION

Given a sequence of trials, the learner tries to accurately predict y_t, the value of the function f(x) given a vector x of attributes. The actual value y_t is returned to the learner, which attempts to minimize the loss L($\hat{y}_t$,y_t) between its prediction and the actual outcome. If the actual value is unknown it can be estimated using temporal difference learning methods described in the previous section, but obviously this makes convergence much more difficult.

The complexity of the learner, determined by its number of hidden nodes, should depend on the complexity of the target class. For example, we have considered the case where x is expanded into its power set with 2^n elements where n is the length of x. Therefore the learner stores 2^n internal weights—one for each of the non-linear combinations of x vector products—which are combined to predict with the equation

$$\hat{y}_t = \sum_{i=1}^{2^n} w_i C(x_i) + w_0, \tag{3.1}$$

where n is the dimension of x and w_0 is a constant weight which corresponds to the empty set in the power set expansion of x. C(x_i) represents the *ith* combination of x vector products.

The linear case is very similar except that the number of weights corresponds directly to the dimensionality of x and the prediction is of the form:

$$\hat{y}_t = \sum_{i=1}^{n} w_i x_i + w_0. \tag{3.2}$$

Here w_0 is a constant factor used for scaling as in the non-linear case. This method performs much better for learning simple linear target classes of functions. Tesauro suggested the use of these linear combinations for learning rules at the bottom-level of a multi-layer network for TD-Gammon [26]. However chess may require the non-linear method described above or at least more complexity than the linear model. This is explored further in Section 5.

There are two other intermediate cases we consider, which allow us to work in higher dimensional spaces without relying upon a linear solution. These involve the use of only the paired or 3-tuple terms. This reduces the number of weights to n(n+1)/2 or $n(n^2 + 5)/6$ respectively, where n is the dimensionality of x. Therefore the prediction is

$$\hat{y}_t = \sum_{i=1}^{n} \sum_{j=1}^{i} w_{i,j} x_i x_j + w_0. \tag{3.3}$$

or the pairs and similarly

$$\hat{y}_t = \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} w_{i,j,k} x_i x_j x_k + w_0 . \qquad (3.4)$$

for the 3-tuples prediction. The practical difference between these two methods for chess is illustrated in Section 6.1.

3.1. Weight Update Policies

After each training example, all of the weights must be updated to move towards the objective function and minimize the loss of the learner on future predictions. In the words of Widrow: “The problem is to develop systematic procedures or algorithms capable of searching the performance surface and finding the optimal weight vector when only measured or estimated data are available” [29].

There are numerous methods for this regression problem, which make use of different loss functions. The 2 main algorithms considered here are gradient descent (GD), which is sometimes called the Widrow-Hoff algorithm [29], and exponentiated gradient (EG±) with positive and negative weight vectors. Kivinen and Warmuth showed the superiority of EG± for sparse target instances. In the non-linear case or even for the pairs, some of the additional expanded hidden terms will act like irrelevant attributes and therefore EG± has some advantages in that case. “For the EG± algorithm, the dependence on the number of irrelevant variables is only logarithmic, so doubling the number of irrelevant variables results in only a constant increase in the total loss” [14]. Complex games like chess can greatly benefit from ignoring many irrelevant attributes and focusing on only the relevant piece interactions, but situations may arise where GD is superior since “as one might expect, neither of the algorithms is uniformly better than the other” [12].

Both GD and EG± are presented briefly here since they are crucial to the proper convergence of any learning algorithm or predictor. Although the performance of the two methods is quite different, they both rely on a common framework that updates each of their weights from w_{old} to w_{new} in order to minimize

$$d(w_{new}, w_{old}) + \eta L(y_t, w \cdot x_t), \qquad (3.5)$$

where L is the square loss function $L(y,x) = (y - x)^2$, η is a positive learning rate, and $d(w_{new},w_{old})$ is a particular distance measure between the two weight vectors. The only difference between them lies in the choice of this distance function, where the GD algorithm uses the squared Euclidean distance and the EG± algorithm uses the relative entropy, also called the Kullback-Leibler divergence [14].

3.2. Multiplicative Updates

"An alternative approach to account for the uncertainty in evaluating the strength of game positions is to translate these evaluations into estimates of the probability of winning the games from these positions, then propagate with estimates by the **product rule** as dictated by probability theory. In chess this translation can be based on statistical records. For example, a standard scale for scoring a chess position is the equivalent number-of-pawns that this position represents" [19]

- (Pearl, pg. 359)

In addition to the update rules considered above, we consider a second class of update rules that use a simple non-weighted product of the input vector components. This method offers some important advantages over other mappings from an arbitrary dimensional space to a real-valued prediction since each vector dimension has a greater effect on the overall prediction. It makes those inputs with the most extreme minimum values clearly stand out from other inputs, and therefore the learner is much more sensitive to small values and in the case of chess makes the learner more risk adverse. The prediction rule is of the form:

$$\hat{Y}_t = \prod_{i=1}^{N} X_{t,i}, \qquad (3.6)$$

where the prediction $\hat{Y}_t$ is based on the weighted product of input vector X_t in N dimensions. This product is proportional to the sum of the logs, which is oftentimes the preferred form since it is easier to calculate and its range is easier to bound, hence avoiding overflow or underflow. Therefore the prediction becomes

$$\hat{Y}_t = \sum_{i=1}^{N} \log_2(X_{t,i}). \qquad (3.7)$$

This method has the additional advantage/restriction of zero weights to be learned, which can add to the simplicity and stability of the learner. We also divide by the entropy to favor states with less variance or uncertainty giving the final prediction of

$$\hat{Y}_t = \frac{\sum_{i=1}^{N} \log_2(X_{t,i})}{\sum_{i=1}^{N} X_{t,i} \log_2\left(\frac{1}{X_{t,i}}\right)}. \qquad (3.8)$$

During the course of our study we began by multiplying the prediction by the minimum of all the Chess Neighborhood vector products in order to induce a more conservative strategy. This naturally leads to the representation presented here, where the smallest values are valued the most.

3.2 Gaussian Normal Distribution

Since a winning position has an evaluation of 1 and a losing position has an evaluation of 0, TD will always return values in the range (0,1). This requires that the predictions of the on-line agent also be scaled into this range. Although this may seem trivial it has been a persistent problem in many of our complex learning models. Our solution lies in using the Gaussian probability density function or normal distribution to compress the predictions into the proper range before computing the loss for the regression network. This refers to approximating the integral,

$$\phi(x) = \int_{-\infty}^{x} \frac{e^{-\frac{1}{2}d^2(x,m,\sigma)}}{\sqrt{2\pi\sigma}}, \qquad (3.9)$$

where d^2 is the distance metric

$$d^2(x,m,\sigma) = \frac{(x-m)^2}{\sigma} \qquad (3.10)$$

for input x, mean m, and standard deviation σ [3]. This can be approximated efficiently without any difficulty using a lookup table and interpolation. Beal and Smith [6] also suggest the use of this type of sigmoid squashing function to convert the prediction into a probability of winning. In particular they suggest that the function

$$S(v) = \frac{1}{1+e^{-v}} \qquad (3.11)$$

can also be used effectively for game playing programs with a prediction v(x) that is a weighted combination of input x, but they only consider the linear case. Also, we have observed that by taking the variance into account, the system can adapt appropriately to tight ranges or wide oscillations that sometimes occur in the learning process.

4. REGRESSION NETWORKS

Neural networks and regression have been the focus of many studies in machine learning and statistics [7]. The traditional neural network has internal connections between each node, which creates a total of 2^n connections for n dimension input vectors. These paths are defined as hidden nodes since they directly correspond to the amount of weights that must be stored and heuristically they represent the relationships between all the different combinations of input spaces in a chess neighborhood. However, for large n this many connections can be impractical both for storage and for time-efficient calculation. Therefore we make use of the 2-tuples and 3-tuples of inputs as shown in (3.3) and (3.4). This improved the performance enormously from using linear terms at the lowest level, a method that already proved sufficient for backgammon. Tesauro motivated his linear approach at the base-level since “the neural network first extracts the linear component of the evaluation function, while non-linear concepts emerge later in learning” [26]. Experiment 6.1 shows the relative performance change after including all the triples, which increases the number of internal nodes to 833 weights for our 17-length input. In general, the number of internal nodes for a k-tuple representation of an n-dimensional input vector is

$$\sum_{i=1}^{k} \binom{n}{i}, \qquad (4.1)$$

but since time is a crucial consideration in chess, we must restrict the learner to a maximum number of connections. The Chess Neighborhoods described below are of length 17 and each board contains 64 squares, so the entire power set of Chess Neighborhood terms would be 2^{17}, which is quite expensive. There is undoubtedly a trade-off between search depth and more internal network weights or connections. Our experiments show promising preliminary findings about the importance of non-linear terms for chess evaluation.

4.1 Multi-layer Networks

Single layer regression networks are the easiest to train, but they are limited to a smaller class of problems. The multi-layer network has proven highly effective for small problems but it too can be very costly in higher dimensional spaces. A simple nested function learner example is given here where the learner tries to predict a function F(X), where F(X) is a non-linear weighted sum of g(x) terms, after receiving the vector x as input. The functions F(X) and g(x) are of dimension N and M respectively, which makes the number of internal weights on each level equal 2^N and 2^M where the internal nodes for g(x) must be calculated for each of N’s components. For the case considered, N was set to 4 and M was set to 3. The functions F(X) and g(x) are set to

$$F(\overline{X}) = 0.2X_1X_2 + 0.6X_1X_2X_3X_4 + 0.1X_1X_3X_4 + 0.9$$
$$s.t. \quad \forall_{0<i\leq N}\ X_i = g(\bar{x}_i) \ \ where$$
$$g(\bar{x}_i) = 0.1x_{i,1} + 0.2x_{i,1}x_{i,2} + 0.3x_{i,2} + 0.4x_{i,3} + 0.5x_{i,1}x_{i,2}x_{i,3}.$$

Using the on-line learning model described for the single-layer case presented in (3.1), the learner is able to effectively minimize its loss on the training data with weights that converge quite rapidly when applied to a 2-layer network with hidden nodes. Both of the update rules GD and EG± are compared using the same randomly drawn examples with the exact same target function. Figure 5.1 shows how both networks were able to accurately predict with a loss of 180 and 38 for GD and EG± respectively on 10,000 training examples.

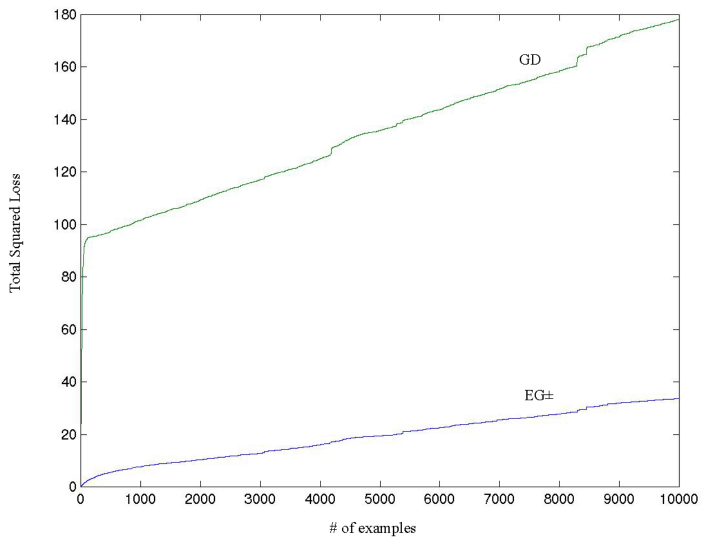

Fig. 5.1 – Cumulative loss of EG± and GD for multi-layer function evaluation

Clearly the EG± algorithm outperforms the GD algorithm in this particular case. The value of η must be carefully tuned for both levels to ensure good performance. This problem is eliminated for our chess agent, with the use of a variable learning rate as shown in Section 5.3.

5. OUR REPRESENTATION

"In more complex games such as chess and Go, one would guess that an ability to learn a linear function of the raw board variables would be less useful than in backgammon. In those games, the value of a particular piece at a particular board location is more dependent on its relation to other pieces on the board. A linear evaluation function based on the raw board variables might not give very good play at all – it could be substantially worse than beginner-level play. In the absence of a suitable recording of the raw board information, this might provide an important limitation in such cases to the success of a TD learning system similar to the one studied here."[26]

-Tesauro (TD-Gammon, pg. 10)

5.1. Chess Neighborhoods

The core of any learning system is the representation, since it provides the basis for anything it could possibly learn. Therefore for chess, we must choose a representation that accurately represents the geometry of the board. The naïve method of simply storing each of the 64 squares would suffer from lack of generalization and would therefore require too much training to be feasible in practice. In fact the geometry is counterintuitive for most humans due to the different movements of the pieces. Snyder and Levinson provided an abstraction in terms of "safe shortest path distances" [16] between the pieces but didn't indicate the best way to evaluate such paths.

We have discovered that the 64 chess neighborhoods that exist in each position can accurately show the complex piece relationships without being too specific to be worthwhile for on-line learning. A *chess neighborhood* is defined as a vector of length 17, where each dimension represents one of the: 2 ranks, 2 files, 4 diagonals, and 8 knight squares around one central square. Blank squares are also included in this representation so there will always be exactly 64 in any position. An example of such a neighborhood is shown in Fig 5.2.

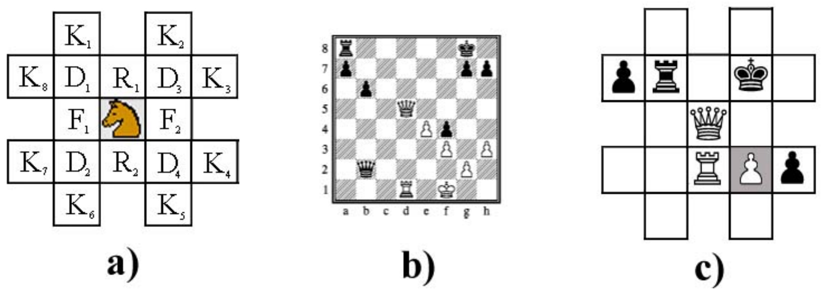

Fig. 5.2 – a) This is a general example of a chess neighborhood with the 2 Files, 2 Ranks, 4 Diagonals, and 8 Knight distances away from one particular center square, which in this case happens to be a Knight. b) Shows a real game position our agent encountered. c) shows one of the 64 chess neighborhoods in the position in b). The darkened lower diagonal signifies its adjacency to the queen.

In a chess neighborhood the values for $D_{1\text{-}4}$, $R_{1\text{-}2}$, $F_{1\text{-}2}$, $K_{1\text{-}8}$, and the value of the center square from Figure 5.2a come from the lowest-level piece weights, which also take into account the chess neighborhood position, like upper left diagonal or left file. We represent the difference between adjacent and non-adjacent pieces in each neighborhood. This is illustrated for an actual game position in Figure 5.2b, in which our agent forked the opponent while simultaneously threatening mate.

5.2. Lowest Level Representation

On the lowest level, the weights are stored for each possible position where a piece could reside in a chess neighborhood, with respect to what piece is in the center of the position. For example, the value of having a black queen adjacent to the upper diagonal of a chess neighborhood is highly dependent on the piece in the center of that particular neighborhood. A white king in the center is in check, whereas an opponent's piece is supported by the black queen. Therefore our learner stores weights for each of these situations, which are updated proportionally to loss on each training example. In particular, the learned values of a piece being in the center square can be thought of as an approximation of the program's material piece values. The values obtained after approximately 2500 games of training, starting with randomly initialized values, are listed in Table 5.1 and they make a great deal of intuitive sense as material evaluation terms. This provides an adequate base onto which we can begin to develop a strong learning system, where there are no human-supplied bottom-level numerical features.

	King	*Queen*	*Rook*	*Bishop*	*Knight*	*Pawn*
Our Agent	0.51354	0.7158	0.63394	0.60757	0.61083	0.54027
Opponent	0.51354	0.29919	0.38528	0.41552	0.41471	0.47008

Table 5.1 – The learned weight values after a mixture of on-line and off-line training with the 2-tuple agent using GD. These values are from white's perspective.

5.3.Bringing it All Together

The overall design of our system incorporates all of the features discussed in this paper. The weights on the lowest level are used as inputs to the 64 chess neighborhoods in each position. Each of the 17-vector chess neighborhoods uses a global set of weights for non-linear evaluation terms to make 64 predictions $\hat{y}_t$. These are then combined on the top-level multiplicatively to create an overall prediction for the board state. Figure 5.3 shows a graph of this learning hierarchy without including the table of base-level feature values presented in the previous section. The Chess Neighborhood weights are updated with the GD algorithm but we also consider the

EG± algorithm. For gradient descent, we use a learning rate η where $\eta = (Y_t - 0.5) / 200$ for actual value Y_t. Dividing by 200 decreases the magnitude that the weights move after each update, which therefore increases the stability of our entire model since the weights change less erratically over time. This puts a higher value on past experience.

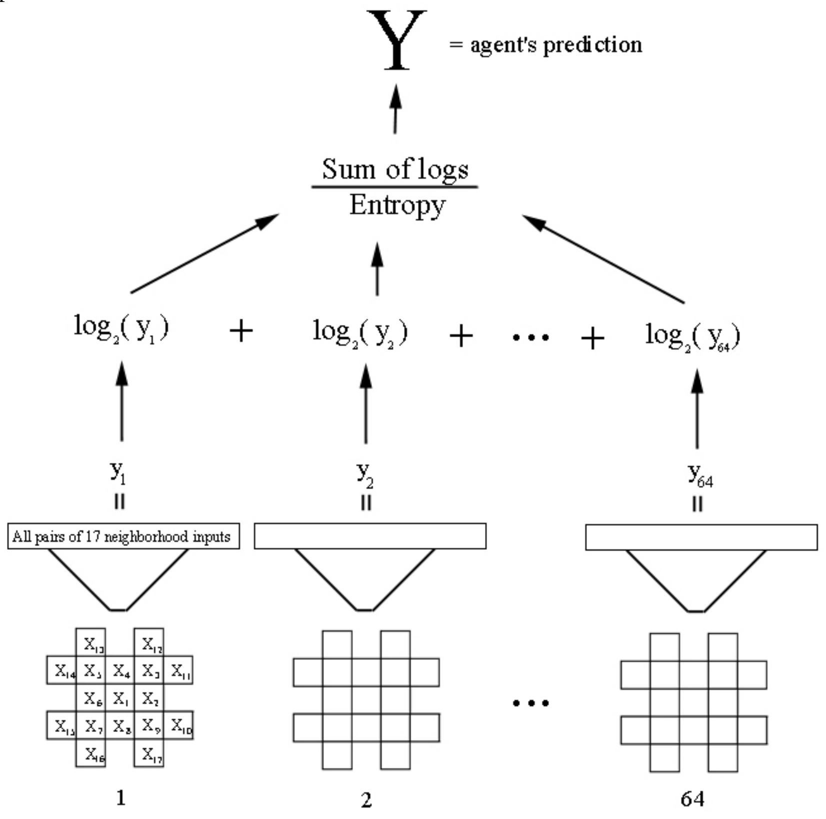

Fig. 5.3 - This shows an overall model of our learner. The Chess Neighborhoods *(bottom)* of are expanded to create intermediate predictions $\hat{y}_t$. The sum of the logs of these predictions is then divided by the entropy to give the overall evaluation Y_t *(top)*.

5.4. Symmetry

One of the problems with building a learner based on, for example, a standard 64 square input representation is that the large number of symmetries on that board would not be exploited, leading to tremendous learning deficiencies. For example, a rook attacking a bishop should have similar significance regardless of where it occurs on the board. The Morph graph representation exploited some of these symmetries but was not able to exploit redundancies across graphs as we can with non-linear regression on the lower-layer [15]. After each training episode, the lowest level weights are averaged based on rank, file, diagonal, and knight movement symmetry. Pawns are a special case where only file symmetry applies.

6. EXPERIMENTAL RESULTS

6.1 Varying Number of Internal nodes

In the first experiment we studied the relative performance improvement with greater numbers of internal weights or nodes. Two versions of our system are trained against one another, where both versions have a different number of internal nodes. Games are conducted in an on-line fashion, where each agent begins with randomly initialized weights and doesn't learn after winning. Both agents use a 2-ply search. The results of Figure 6.1 show the improvement achieved by going from storing and updating all pairs of inputs to storing and updating all triples of the input vector. Tesauro showed for his TD net that performance increases monotonically with the number of hidden nodes [27]. We expect the performance to steadily increase in this fashion until the complexity of the internal representation is greater than or equal to the complexity of the problem. For example, a linear version of the regression model outperforms the non-linear version, when learning simple learner target classes, which is expected since the knowledge of the correct target starts the linear learner off with a much smaller universe of possible classifications. However, using a representation greater than the 2-tuples or 3-tuples will take a great deal of time, especially when combined with minimax search. Ideally search can be eliminated entirely and replaced with positional knowledge in the form of internal weights and improved generalization among similar positions but minimal search may still be necessary to achieve competitive play and keep the program from throwing away material.

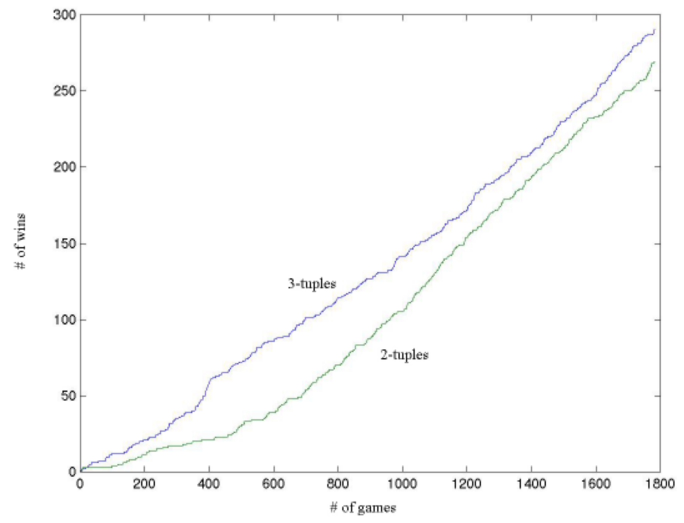

Fig. 6.1 – GD learning agents with 2-tuples (pairs) and 3-tuples of input vector products

6.2 Replacing Minimax Search with Knowledge

A very important question is whether the representation and learning scheme we have illustrated in this paper can be used to effectively replace minimax tree search. Therefore we have tested the 3-tuple agent against a random agent with a greater search depth. Despite its random evaluation at the leaf nodes, the random agent is quite strong due to the importance of mobility in chess and the uniform distribution of the random numbers, causing those positions with the greatest number of leaf nodes to tend to have the most extreme evaluations [4]. Games are conducted on-line as in the previous experiment where the winner doesn't learn from the previous game. The actual learning agent is set to 2-ply of search, and the random agent is set to 5-ply. As the graph in Figure 6.2 expresses, the 3-tuple agent effectively learns to beat the 5-ply random agent at an increasing rate with 3-ply less search. This conclusion illustrates the importance of a good model with relevant features to accelerate learning and decrease the importance of search.

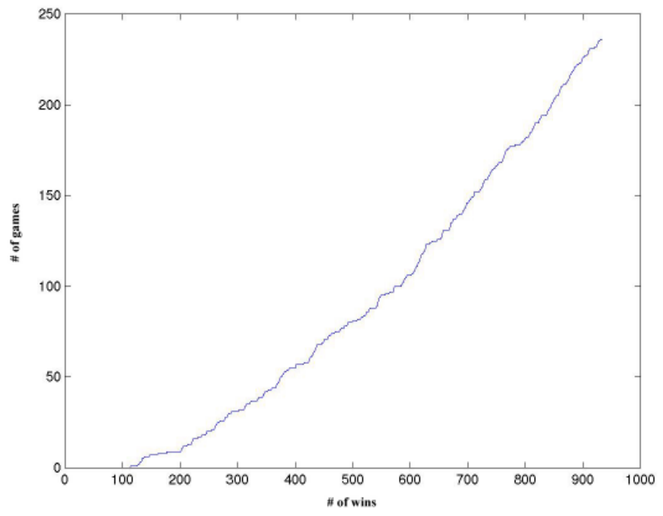

Fig 6.2 – The wins of a 3-tuple 2-ply GD agent over time playing a random agent with a 5-ply search. Initially the learner's weights are randomly chosen

6.3. Exponentiated Gradient vs. Gradient Descent

We previously showed that EG± can outperform GD in some instances, especially for sparse input vectors and non-linear target functions but neither method has been shown to be superior in every case. For this reason, it is important to compare the performance of both methods in this application. This might give us a greater understanding of the underlying target function, which represents the interactions between pieces in our model. As in the previous experiment both agents GD and EG± start with an empty database and play successive games against each other where only the loser learns from the previous trial. In this case our experiments up to now have proven inconclusive, since gradient descent is the clear winner initially but EG± is ahead at the conclusion of the experiment and appears to have a steeper learning curve. Gradient descent is better when "the weight vectors with low empirical loss have many nonzero components, but the instances contain many zero components," [12] which may be the case in our chess model since many of the 64 squares are blank squares, especially in the endgame positions which have the most extreme evaluations and the non-blank pairs of input vectors may stand-out. However, the nature of the multi-layer learning system with non-linear terms seems to favor the EG± algorithm, especially as the number of internal nodes increases. We intend to continue this experiment to compare their performance on a larger time scale and with different numbers of internal nodes.

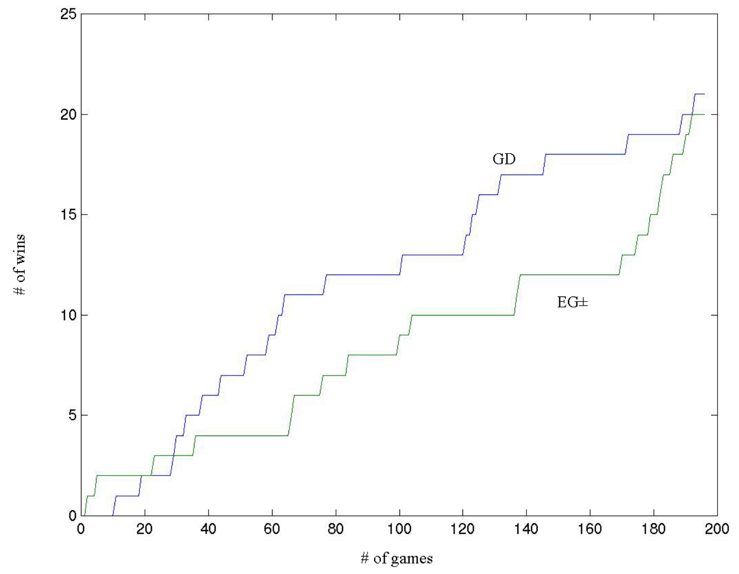

Fig 6.3 – The sum of wins for each of GD and EG± with a 2-ply search and initially random weights

6.4. Internet Chess Club and Grandmaster Datasets

Our new agent has competed for a few days on the Internet Chess Cub (ICC) and trained from several hundred Grandmaster games from online datasets. The former is by far the most useful training technique since the agent can actively explore the consequence of its own conclusions. As mentioned previously, the 2 initial random moves add enough variability to the games to make them interesting and productive for the learning process. Its rating continues to climb while playing on the internet chess server (ICC). So far the 2-tuple agent with a 4-ply search has achieved a rating of 1042 on ICC, which is a significant improvement over other learning programs such as Morph IV [17], which required months of training to reach the same level. Its learning and/or playing performance appears to be superior to that reported for other learning chess systems, such as NeuroChess [28], SAL [11], and Octavius [20]. We have also found bootstrap learning to be extremely effective for training, which contradicts the findings Baxter and Tridgell had for their agent KnightCap, which had difficulty acquiring knowledge autonomously [2]. Computers have certainly gotten faster in recent years, but our agent's improved learning ability is due to better representations rather than processing speed.

CONCLUSION

One of the enjoyable things about encountering a new opponent is to find out what “chess theory" they are consciously or unconsciously using. One difference between human and machine chess players is that the human's “theory" about chess evolves from game to game and sometimes move-to-move. However, for traditional chess computers, their “theory" is static, being built into their evaluation function and search heuristics - if any flexibility is retained it is style knobs that are alterable by the user. The program discussed in this paper represents a departure from the norm: the computer develops its own theory of chess evaluation and tests and evolves the theory through over-the-board experience. Another aspect of a theory is that its individual primitive components are either few (as in $E=MC^2$) or uniform. In this case we give the computer the framework of a theory based on uniform “chess neighborhoods" but leave it up to the system with its experience to fill in the details.

In order to participate properly in chess combat, the program must learn to evaluate various aspects of its position and then combine the values of these aspects into a single number that it attempts to maximize with the assistance of look-ahead search. To date, these two types of valuations: 1) Values of parts or features of a position 2) Whole positions based on the values of individual features - have been too combinatorally complex for machine learning programs to employ without making serious blunders such as needless tossing away of material. Unlike backgammon, where a probabilistic approach is highly appropriate, in chess, one bad move can be fatal. We believe that in this paper we have made significant steps in the resolution of the learned evaluation problem.

The contributions in this paper, by themselves, may not be original or novel, but when put together they represent a significantly new approach to chess evaluation. Contributions of this paper include:

1. The definition and use of chess neighborhoods to encapsulate local knowledge about a chess position.
2. The use of a regression network to learn non-linear combinations of the individual values of pieces that make up a piece neighborhood to arrive at a single value for the entire neighborhood. The use of the regression network dramatically reduces the cost of learning the value of patterns over pattern systems such as Morph, that are unable to exploit the redundancy across the patterns.
3. The use of ``exponential gradient descent" as opposed to traditional gradient descent in the learning of non-linear functions in which many sub-terms may be irrelevant.
4. The use of a symmetry updating phase to improve the speed of learning in a network by making nodes that ``should be equal" be equal by taking an average of their values.

5. The use of a maximum product rule and minimum entropy rule to combine the 64 neighborhood evaluations in a position in a conservative risk-adverse way appropriate to good chess evaluation.
6. By starting the game by playing two random moves, increasing the exploration of the chess learned space.

In ongoing work, among other things, we are working on assessing the trade-offs between number of hidden (non-linear) nodes in the regression network, search depth and performance.

REFERENCES

1. Allen, J., Hamilton, E., and Levinson, R. New Advances in Adaptive Pattern-Oriented Chess (1997). In H.J. van den Herik and J .W.H., Uiterwijk. Advances in Computer Chess 8, pp. 312-233., Universiteit Maastricht, The Netherlands.

2. Baxter, J., Tridgell, A., and Weaver, L. A chess program that learns by combining TD(λ) with game tree search. In *Proceedings of the 15th International Conference on Machine Learning (ICML-98)*, pages 28-36. Madision, WI. 1998. Morgan Kaufmann.

3. Ballard, D. H. An Introduction to Natural Computation. Cambridge: MIT Press.

4. Beal, D. F., & Smith, M.C. (1994). Random Evaluation in Chess. ICCA Journal, Vol. 17, No. 1, pp. 3-9 (A).

5. Beal, D. F., & Smith, M.C. Learning Piece Values Using Temporal Differences. *Journal of The International Computer Chess Association*, September 1997.

6. Beal, D. F., & Smith, M.C. First results from using temporal difference learning in Shogi. In H. J. van den Herik and H. Iida, editors, Proceedings of the First International Conference on Computers and Games (CG-98), volume 1558 of Lecture Notes in Computer Science, page 114, Tsukuba, Japan, 1998. Springer-Verlag.

7. Bishop, Chirstopher M. *Neural Networks for Pattern Recognition*, Oxford Univ. Press, 1998. ISBN 0-19-853864-2.

8. Bradtke, S. J., and Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22, 33-57.

9. Christensen, J. and Korf, R. (1986). A unified theory of heuristic evaluation functions and its applications to learning. Proceedings of AAAI-86 (pp. 148-152).

10. Fürnkranz, J., Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147-160, September (1996).

11. Gherrity, M. A Game-Learning Machine. Ph.D thesis. University of California, San Diego. San Diego, CA. 1993.

12. Helmbold, D. P., Kivinen, J., and Warmuth, M. K. (1996a), Worst-case loss bounds for signmoided linear neurons, in "Advances in Neural Information Processing Systems 8," MIT Press, Cambridge, MA.

13. Herik, H.J. van den. A New Research Scope. International Computer Chess Association Journal 21(4), 1998.

14. Kivinen, J. and Warmuth, M. K. Additive versus exponentiated gradient updates for linear prediction. Information and Computation. Vol. 2, pp. 285-318, 1998.

15. Levinson, R. A., and Snyder, R. (1991). Adaptive pattern-oriented chess. In L. Birnbaum and G. Collins (Eds.), Proceedings of the 8th International Workshop on Machine Learning, pp. 85-89, Morgan Kaufmann.

16. Levinson, R. A., and Snyder, R., "Distance: Towards the Unification of Chess Knowledge", International Computer Chess Association Journal 16(3): 123-136, September 1993.

17. Levinson, R. A., and Weber, R. J. (2000). "Pattern-level Temporal Difference Learning, Data Fusion, and Chess". In SPIE's 14th Annual Conference on Aerospace/Defense Sensing and Controls: Sensor Fusion: Architectures, Algorithms, and Applications IV.

18. Littlestone, N., Long, P.M., and Warmuth, M. K. (1995), On-line learning of linear functions, *Journal of Computational Complexity* 5, 1-23.

19. Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, Massachusetts.

20. Pellen, Luke. Nerual net chess program Octavius: http://home.seol.net.au/luke/Octavius (1999).

21. Samuel, A. (1959. Some studies in machine learning using the game of checkers. *IBM J. of Research and Development*, 3, 210-229.

22. Scott, J. Machine Learning in Games: the Morph Project, Swarthmore College, Swarthmore, PA. http://forum.swarthmore.edu/~jay/learn-game/projects/morph.html.

23. Slate, D.J., A chess program that uses its transposition table to learn from experience. *International Computer Chess Association Journal* 10(2):59-71, 1987.

24. Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9-44.

25. Sutton, R. S., & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. Cambridge: MIT Press.

26. Tesauro, G. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol 38, No 3, March 1995.

27. Tesauro, G. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257-278, 1992.

28. Thrun, S., 1995. Learning to Play the Game of Chess. In Advances in Neural Information Processing Systems (NIPS) 7, G. Tesauro, D. Touretzky, and T. Leen (eds.), MIT Press.

29. Widrow, B., and Stearns, S. (1985), "Adaptive Signal Processing," Prentice Hall, Engelwood Cliffs, NJ.