Tuning Evaluation Functions for Search

Chris McConnell, ccm@cs.cmu.edu
CMU School of Computer Science
Pittsburgh, PA 15213-3891

1 Introduction

This paper examines the problem of applying machine learning techniques to improving the performance of actual game playing programs in complex domains like chess. This is a challenging problem because chess is a domain where a great deal of human effort has been spent and the performance level of programs is already very high.

Current programs play chess by focusing on accumulating a material advantage that eventually becomes so crushing that a mate can be found. This advantage is accumulated through the application of tactics–a specific sequence of moves leading to a material gain. One of the current weaknesses in computer programs is that they have a weak sense of what to do when there are no tactical plans that can be found through search. What humans do in these positions is to play moves that provide strategic opportunities that eventually accumulate to the point that they can be converted to a tactical plan. By the time the computer sees the tactical plan it is too late.

2 Learning Types of Knowledge

How can machine learning techniques be applied to improving the play of an existing program? Current chess programs typically have four different forms of knowledge embodied in them: 1) an opening book, 2) static evaluation functions, 3) a search algorithm, and 4) endgame databases. The opening book and endgame databases are used in a straightforward fashion and applying ML techniques to them would not gain much. Discovering new algorithms is a challenging task beyond current ML techniques. The best opportunity is found in evaluation functions. They typically have the form: $E(p) = \sum w_i f_i(p)$ where p is a position, w_i is the weight assigned to feature i and $f_i(p)$ is the amount of feature i present. Evaluation functions estimate the quality of a position. In some programs there are several evaluation functions that are used in different ways by the search algorithm. A good example is B* Hitech where there is both a realistic and an optimistic evaluation function.[2]

There are several ways to improve an existing evaluation function. One approach is to find new features. This approach is difficult in a domain like chess. The dynamic aspects of a position are hard to capture statically. Being ahead a queen does not guarantee that you will not be mated in the next couple of moves!

A second way to improve an existing evaluation function is to tune the weights w_i such that $E(p)$ is a better estimator. This is historically the most common goal of ML research applied to games. A procedure for automatically tuning weights would be useful for several reasons. Introducing new features would be easier since the tuning could adjust the weights to reflect the interactions between the old and new features. Automatic tuning also allows multiple sets of weights appropriate to different situations to be learned. This is important in a complex domain where the relationships between features change depending on the overall context. For example, in chess middle game positions, the overall strategic situation is determined by the pawn structure.

3 Training Signals

Whatever learning technique is used for tuning weights, a training signal is necessary to indicate the kind of changes that need to be made. The training signal can come either from the difference between a static evaluation and an evaluation backed up by search or from the differences between the evaluations of a known good move and its alternatives.[7] Historically the first type of signal has been used by reinforcement learning schemes and the second type by schemes based on pattern recognition techniques like linear regression, linear discriminants or back propagation.

Reinforcement learning attempts to learn an exact value for a position. In a game like chess, a perfect evaluation function would assign the game theoretic value of win, lose or draw to every position. Finding the game theoretic value is not in general possible because of the size of the state space. Given the static features typically used in evaluation functions, there is no weight assignment that could lead to an accurate value for every position. The best an evaluation function can do is to use static features to estimate the dynamic outcome of playing from a position with bounded information for both sides.

Some static features are much more reliable indicators of the quality of a position than others. Combining the evaluation function with a search procedure improves the decision quality at the root by making reliable evaluations more likely. In chess, if a possible move from the initial position leads via best play to a position where the moving side is significantly ahead in material without any immediate chance of losing that advantage, then that would be a good move to make. Search also limits your fallibility, if you can't see calamity in n ply, your opponent must be able to see at least one move further or you are safe for the time being.

Pattern recognition approaches focus on the relative values of the evaluation of a desired state and of its alternatives. In theory it should be easier to learn an evaluation function that optimizes relative values rather than absolute ones since there are more functions that meet that standard. Pattern recognition approaches can also be applied to leaves found at search frontiers as well be described later.

4 Sources of Training Data

Where does the data for a training signal come from? Either kind of training signal can be generated by self play or from a database of good moves. One advantage of self play is that the evaluation function is likely to converge on a strategy that can be represented in terms of the evaluation function's features. Another advantage of self play is that it does not require the effort of acquiring a database. Unfortunately in a domain with a large state space like chess where search is required to get stable evaluations, self play is prohibitively expensive. Most of the effort is spent exploring parts of the state space that are unlikely to be reached in actual games. A better approach is to use a database of moves by human experts. The main advantage is that the effort of generating training signals is spent on positions that are likely to be seen. The main disadvantages are acquiring the database, the quality of the moves in the database and that the evaluation function may not have the features required to reliably make choices in the database.

A single position in the database contains a lot of implicit information. The search tree that is rooted at a database position contains many instances of the relationship between static evaluations and search evaluations. These instances can be used to generate training signals that are the difference between a static evaluation and the evaluation returned from search as used by reinforcement learning techniques. [5, 6]

The simplest pattern recognition approaches use only the static evaluations of the moves from the root.[4, 3] Slightly more sophisticated approaches consider an n-ply search, but only use the static evaluation of the leaf at the end of each move's principal variation. [1, 8] After changing the weights, search is done again possibly generating new principal variations and the procedure iterates. This procedure runs the risk of cycling and misses better possible weights by not considering the search values found on lines off of the principal variation. Ideally a tuning procedure could be found that can use the information found in the search tree below each position without getting overwhelmed.

Using search trees also makes weight changes more likely to be useful to a playing program. The changes in the weights actually change top level decisions only when combined with search. The weight changes very well may not predict the best move at depths shallower than the training data. The hope is that the weight changes will predict the choices made with searches that are deeper than were used in training since they inherit the progress already made towards the patterns implicit in the weight changes.

5 A New Tuning Procedure

At CMU, I have developed a new technique for automatically tuning evaluation function weights. It uses a database of human games broken up into strategic situations. A search tree is generated for each position from which critical lines are saved. The weights are then tuned with respect to these critical lines in all positions with a discrete version of subgradient optimization. The modified weights are then used for new searches which can add additional critical lines that can be thought of as a critique of the particular weight changes that are proposed. The procedure iterates until the expectations of the tuning procedure and the critical lines match. Some early results of applying this procedure to computer chess will also be described.

References

[1] Thomas S. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess.* PhD thesis, Carnegie-Mellon University, May 1990.

[2] Hans J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[3] T.A. Marsland. Evaluation function factors. *ICCA Journal*, 8(2):47–57, June 1985.

[4] T. Nitsche. A learning chess program. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*. Pergamon Press, 1982.

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

[6] G. Tesauro. Temporal difference learning of backgammon strategy. In *Proc. Ninth International Conf. on Machine Learning*, 1992.

[7] Paul E. Utgoff and Jeffery A. Clouse. Two kinds o ftriniang information for evalution function learning. In *Proc. AAAI*, pages 596–600, 1991.

[8] M. v.d. Meulen. Weight assessment in evaluation functions. In D.F. Beal, editor, *Advances in Computer Chess 5*, pages 81–89. North-Holland, 1989.