Machine Learning in Games: A Survey

Johannes Fürnkranz
Austrian Research Institute for Artificial Intelligence
Schottengasse 3, A-1010 Wien, Austria
E-mail: juffi@ai.univie.ac.at

Technical Report OEFAI-TR-2000-31*

Abstract

This paper provides a survey of previously published work on machine learning in game playing. The material is organized around a variety of problems that typically arise in game playing and that can be solved with machine learning methods. This approach, we believe, allows both, researchers in game playing to find appropriate learning techniques for helping to solve their problems as well as machine learning researchers to identify rewarding topics for further research in game-playing domains. The paper covers learning techniques that range from neural networks to decision tree learning in games that range from poker to chess. However, space constraints prevent us from giving detailed introductions to the used learning techniques or games. Overall, we aimed at striking a fair balance between being exhaustive and being exhausting.

* *To appear in:* J. Fürnkranz & M. Kubat: Machines that Learn to Play Games, Nova Scientific Publishers, Chapter 2, pp. 11–59, Huntington, NY, 2001.

1 Samuel's Legacy

In 1947, Arthur L. Samuel, at the time Professor of Electrical Engineering at the University of Illinois, came up with the idea of building a checkers program. Checkers, generally being regarded a simpler game than chess, seemed to be a perfect domain for demonstrating the power of symbolic computing with a quick programming project. The plan was straight-forward: "*write a program to play checkers [...] challenge the world champion and beat him*". With this tiny project, he hoped to generate enough interest for raising funds for a university computer.[1]

Little did Samuel know that he would work on this program for the next two decades, thereby producing not only a master-level checkers player—it beat one of America's best players of the time in a game that is published in (Feigenbaum and Feldman 1963)—but also introducing many important ideas in game playing and machine learning. The two main papers describing his research (Samuel 1959; Samuel 1967) became landmark papers in Artificial Intelligence. In these works, Samuel not only pioneered many popular enhancements of modern search-based game-playing programs (like alpha-beta cutoffs and quiescence search), but also invented a wide variety of learning techniques for automatically improving the program's performance over time. In fact, he considered checkers to be a perfect domain for the study of machine learning because in games, many of the complications that come with real-world problems are simplified allowing the researchers to focus on the learning problems themselves (Samuel 1959). As a result, many techniques that contributed to the success of machine learning as a science can be directly traced back to Samuel, and most of Samuel's ideas for learning are still in use in one form or another.

First, his checkers player remembered positions that it frequently encountered during play. This simple form of *rote learning* allowed it to save time, and to search deeper in subsequent games whenever a stored position was encountered on the board or in some line of calculation. Next, it featured the first successful application of what is now known as *reinforcement learning* for tuning the weights of its evaluation function. The program trained itself by playing against a stable copy of itself. After each move, the weights of the evaluation function were adjusted in a way that moved the evaluation of the root position after a quiescence search closer to the evaluation of the root position after searching several moves deep. This technique is a variant of what is nowadays known as *temporal-difference learning* and commonly used in successful game-playing programs. Samuel's program not only tuned the weights of the evaluation but it also employed on-line *feature subset selection* for constructing the evaluation function with the terms which seem to be the most significant for evaluating the current board situation. Later, he changed his evaluation function from a linear combination of terms into a structure that closely resembled a 3-layer neural network. This structure was trained with *comparison training* from several thousand positions from master games.

Since the days of Samuel's checkers program the fields of machine learning and game playing have gone a long way. Many of the new and powerful techniques that have been developed in these areas can be directly traced back to some of Samuel's

[1] The quotation and the facts in this paragraph are taken from (McCorduck 1979).

ideas. His checkers player is still considered to be among the most influential works in both research areas, and a perfect example for a fruitful symbiosis of the two fields.

1.1 Machine Learning

Machine learning has since developed into one of the main research areas in Artificial Intelligence with several journals and conferences devoted to the publication of its main results. Recently, the spin-off field *Knowledge Discovery in Databases* aka *Data Mining* (Fayyad et al. 1996; Fayyad et al. 1995; Witten and Frank 2000) has attracted the interest of the industry and is considered by many to be one of the fastest-growing commercial application areas for Artificial Intelligence techniques. Machine learning and data mining systems are used for analyzing of telecommunications network alarms (Hätönen et al. 1996), supporting medical applications (Lavrač 1999), detecting cellular phone fraud (Fawcett and Provost 1997), assisting basketball trainers (Bhandari et al. 1997), learning to play music expressively (Widmer 1996), efficiently controlling elevators (Crites and Barto 1998), automatically classifying celestial bodies (Fayyad et al. 1996), mining documents on the World-Wide Web (Chakrabarti 2000), and, last but not least, tuning the evaluation function of one of the strongest backgammon players on this planet (Tesauro 1995). Many more applications are described in (Michalski, Bratko, and Kubat 1998). For introductions into Machine Learning see (Mitchell 1997a) or (Mitchell 1997b), for an overview of some interesting research directions cf. (Dietterich 1997).

1.2 Game Playing

Research in game playing has fulfilled one of AI's first dreams, a program that beat the chess world champion in a tournament game (Newborn 1996) and, one year later, in a match (Schaeffer and Plaat 1997; Kasparov 1998). The checkers program CHINOOK was the first to win a human world championship in any game (Schaeffer et al. 1996; Schaeffer 1997). Some of the simpler popular games, like connect-4 (Allis 1988), Gomoku (Allis 1994), or nine men's morris aka mill (Gasser 1995) have been solved. Programs are currently strong competitors for the best human players in games like backgammon (Tesauro 1994), Othello (Buro 1997), and Scrabble (Sheppard 1999). Research is flourishing in many other games, like poker (Billings et al. 1998a), bridge (Ginsberg 1999), shogi (Matsubara 1998) or Go (Müller 1999), although humans still have the competitive edge in these games. A brief overview of the state-of-the-art in computer game playing can be found in (Ginsberg 1998), a detailed discussion in (Schaeffer 2000), which was written for the occasion of the 40th anniversary of the publication of (Samuel 1960).

1.3 Introduction

In this paper, we will attempt to survey the large amount of literature that deals with machine-learning approaches to game playing. Unfortunately, space and time do not permit us to provide introductory knowledge in either machine learning or game playing (but see the references given above). The main goal of this paper is to enable

the interested reader to quickly find previous results that are relevant for her research project, so that she may start her investigations from there.

There are several possible ways for organizing the material in this paper. We could, for example, have grouped it by the different games (chess, Go, backgammon, shogi, Othello, bridge, poker to name a few more popular ones) or by the learning techniques used (as we have previously done for the domain of chess (Fürnkranz 1996)). Instead, we decided to take a problem-oriented approach and grouped them by the challenges that are posed in different aspects of the game. This, we believe, allows both, researchers in game playing to find appropriate learning techniques for helping to solve their problems as well as machine learning researchers to identify rewarding topics for further research in game-playing domains.

We will start with a discussion of book learning, i.e., for techniques that store precalculated moves in a so-called book for rapid access in tournament play (Section 2). Next, we will address the problem of using learning techniques for controlling the search procedures that are commonly used in game playing programs (Section 3). In Section 4, we will review the most popular learning task, namely the automatic tuning of an evaluation function. We will consider supervised learning, comparison training, reinforcement and temporal-difference learning. In a separate subsection, we will discuss several important issues that are common to these approaches. Thereafter (Section 5), we will survey various approaches for automatically discovering patterns and plans, moving from simple advice-taking over cognitive modeling approaches to the induction of patterns and playing strategies from game databases. Finally, in Section 6, we will briefly discuss *opponent modeling*, i.e., the task of improving the program's play by learning to exploit the weaknesses of particular opponents.

2 Book Learning

Human game players not only rely on their ability to estimate the value of moves and positions but are often also able to play certain positions "by heart", i.e., without having to think about their next move. This is the result of home preparation, opening study, and *rote learning* of important lines and variations. As computers do not forget, the use of an opening book provides an easy way for increasing their playing strength. However, the construction of such opening books can be quite laborious, and the task of keeping it up-to-date is even more challenging.

In this section, we will discuss some approaches that support this process with machine learning techniques. We will not only concern ourselves with opening books in the strict sense, but summarize several techniques that aim at improving play by preparing the correct move for important positions through off-line analysis.

2.1 Learning to choose opening variations

The idea of using opening books to improve machine-play has been present since the early days of computer game-playing. Samuel (1959) already used an opening book in his checkers playing program, as did Greenblatt, Eastlake III, and Crocker (1967) in their chess program. Opening books, i.e., pre-computed to replies for a set of positions,

can be easily programmed and are a simple way for making human knowledge, which can be found in game-playing books, accessible to the machine. However, the question which of the many book moves a program should choose is far from trivial.

Hyatt (1999) tackles the problem of learning which opening his chess program CRAFTY should play and which it should avoid. He proposes a reinforcement learning technique (cf. Section 4.3) to solve this problem, using the computer's position evaluation after leaving the book as an evaluation of the playability of the chosen line. In order to avoid the problem that some openings (gambits) are typically underestimated by programs, CRAFTY uses the maximum or minimum (depending on the trend) of the evaluations of the ten positions encountered immediately after leaving book. Special provisions are made to take into account that values from deeper searches as well as results against stronger opponents are more trust-worthy.

2.2 Learning to extend the opening book

Hyatt's work relies on the presence of a (manually constructed) book. However—although there is a lot of book information readily available for games popular games like chess—for many games no such information is available and suitable opening books have to built from scratch. But even for games like chess, mechanisms for automatically extending or correcting existing opening books are desirable. In the simplest case, a program could extend its opening library off-line, by identifying lines that are frequently played and computing the best move for these positions (Frey 1986).

In its matches against Kasparov, DEEP BLUE employed a technique for using the enormous game databases that are available for chess to construct an *extended opening book* (Campbell 1999). The extended opening book is used in addition to the regular, manually maintained opening book. It is constructed from all positions up to the 30th move from a database of 700,000 grandmaster games. At each of these positions, one could compute an estimate of the goodness of each subsequent move by using the proportion of games won (draws counting as half a win) with this move. However, such a technique has some principle flaws, the most obvious being that a single game or analysis may refute a variation that has been successfully played for years. Hence, DEEP BLUE does not solely rely on this information, but performs a regular search where it awards small bonuses for moves that are in the extended opening book. The bonuses are not only based on the win percentages, but also on the number of times a move has been played, the strength of the players that played the move, annotations that were attributed to the move, the recency of the move, and several others. In total, these criteria can add up to increase (or decrease) a move's evaluation by about half a pawn. This amount is small enough to still allow the search to detect and avoid (or exploit) faulty moves in the database, but large enough to bias the move selection towards variations that have been successfully employed in practice. The procedure seems to work quite well, which is exemplified by the second game of DEEP BLUE's first match with Kasparov, where, due to an configuration error, its regular book was disabled and DEEP BLUE relied exclusively on its combination of regular search and extended book to find a reasonable way through the opening jungle (Campbell 1999).

Buro (1999b) describes a technique that is used in several of the strongest Othello programs. It incrementally builds an opening book by adding every position of every

game encountered by the program (or from external sources) to a move tree. The program evaluates all positions as it goes, and enters not only the move played but also the next-best move according to its own evaluation. Thus, at each node, at least two options are available, the move played and an alternative suggestion. Each leaf node of the book tree is either evaluated by the outcome (in case it was the terminal position of a game) or a heuristic evaluation (in case it was an alternative entered by the program). During play, the program finds the current position in the book and evaluates it with a negamax search through its subtree. The use of negamax search over these game trees with added computer evaluations is much more informative than straight-forward frequency-based methods. As long as the move tree can be kept in memory, this evaluation can be performed quite efficiently because all nodes in the book tree are already evaluated and thus there is no need for keeping track of the position on the board. It is also quite straight-forward to adjust the behavior of the book by, for example, specifying that draws are considered losses, so that the program will avoid drawing book lines.[2] Buro's article is reprinted as Chapter 4 of (Fürnkranz and Kubat 2001).

2.3 Learning from mistakes

A straight-forward approach for learning to avoid to repeat mistakes is to remember each position in which the program made a mistake so that it is alert when this position is encountered the next time. The game-playing system HOYLE (Epstein 2001) implements such an approach. After each decisive game, HOYLE looks for the last position in which the loser could have made an alternative move and tries to determine the value of this position through exhaustive search. If the search succeeds, the state is marked as "significant" and the optimal move is recorded for future encounters with this position. If the search does not succeed (and hence the optimal move could not be determined), the state is marked as "dangerous". More details on this technique can be found in (Epstein 2001).

In conventional, search-based game-playing programs such techniques can easily be implemented via their *transposition tables* (Greenblatt et al. 1967; Slate and Atkin 1983). Originally, transposition tables were only used locally with the aim of avoiding repetitive search efforts (e.g., by avoiding to repeatedly search for the evaluation of a position that can be reached with different move orders). However, the potential of using global transposition tables, which are initialized with a set of permanently stored positions, to improve play over a series of games was soon recognized.

Once more, it was Samuel (1959) who made the first contribution in this direction. His checkers player featured a rote learning procedure that simply stored every position encountered together with its evaluation so that it could be reused in subsequent searches. With such techniques, deeper searches are possible because on the one hand the program is able to save valuable time because positions encountered in memory do not have to be re-searched. On the other hand, if the search encounters a stored

[2] Actually, Buro suggests to discern between *public* draws and *private* draws. The latter—being a result of the program's own analysis or experience and thus, with some chance, not part of the opponent's book knowledge—could be tried in the hope that the opponents makes a mistake, while the former may lead to boring draws when both programs play their book moves (as is known from many chess grandmaster draws).

position during the look-ahead, it is effectively able to search the original position to an increased depth because by re-using the stored evaluation, the search depth at which this evaluation has been obtained is in effect added to the search depth at which the stored position is encountered. A very similar technique was used in the BEBE chess program, where the transposition table was initialized with positions from previous games. It has been experimentally confirmed that this simple technique learning in fact improves its score considerably when playing 100-200 games against the same opponent (Scherzer et al. 1990).

In game like chess, such rote learning techniques help in opening or endgame play. In complicated middlegame positions, where most pieces are still on the board, chances are considerably lower that the same position will be encountered in another game. Thus, in order to avoid an explosion of memory costs by saving unnecessary positions, Samuel (1959) also devised a scheme for *forgetting* positions that are not or only infrequently used. Other authors tried to cope with these problems by being selective in which positions are added to the table. For example, Hsu (1985) tried to identify the faulty moves in lost games by looking for positions in which the value of the evaluation function suddenly drops. Positions near that point were re-investigated with a deeper search. If the program detected that it had made a mistake, the position and the correct move were added to the program's global transposition table. If no single move could be blamed for the loss, a re-investigation of the game moves with a deeper search was started with the first position that was searched after leaving the opening book. Frey (1986) describes two cases where an Othello program (Hsu 1985) successfully learned to avoid a previous mistake. Similar techniques were refined later (Slate 1987) and were incorporated into state-of-the-art game playing programs, such as the the chess program CRAFTY (Hyatt 1999).

Baxter, Tridgell, and Weaver (1998b) also adopt this technology but discuss some inconsistencies and propose a few modifications to avoid them. In particular, they propose to insert not only the losing position but also its two successors. Every time a position is inserted, a consistency check is performed to determine whether the position leads to another book position with a contradictory evaluation, in which case both positions are re-evaluated. This technique has the advantage that only moves that have been evaluated by the computer are entered into the book, so that it never stumbles "blindly" into a bad book variation.

2.4 Learning from simulation

The previous techniques were developed for deterministic, perfect information games where evaluating a position is usually synonymous for searching all possible continuations to a fixed depth. Some of them may be hard to adapt for games with imperfect information (e.g., card games like bridge) or a random component (e.g., dice games like backgammon) where deep searches are infeasible and techniques like storing precomputed evaluations in a transposition table do not necessarily lead to significant changes in playing strengths. In these cases, however, conventional search can be replaced by *simulation search* (Schaeffer 2000), a search technique which evaluates positions by playing a multitude of games with this starting position against itself. In each of these games, the indeterministic parameters are assigned different, concrete

values (e.g., by different dice rolls or by dealing the opponents a different set of cards or tiles). Statistics are kept over all these games which are then used for evaluating the quality of the moves in the current state.

Tesauro (1995) notes that such *roll-outs* can produce quite reliable comparisons between moves, even if the used program is not of master strength. In the case of backgammon, such analyses have subsequently led to changes in opening theory (Robertie 1992, 1993). Similar techniques can be (and indeed are) used for position evaluation in games like bridge (Ginsberg 1999), Scrabble (Sheppard 1999), or poker (Billings et al. 1999), and were even tried as an alternative for conventional search in the game of Go (Brügmann 1993). It would also be interesting to explore the respective advantages of such Monte-Carlo search techniques and reinforcement learning (see (Sutton and Barto 1998) for a discussion of this issue in other domains).

3 Learning Search Control

A typical implementation of a search-based game-playing program has a multitude of parameters that can be used to make the search more efficient. Among them are search depth, search extensions, quiescence search parameters, move ordering heuristics and more.[3] Donninger (1996, personal communication) considers an automatic optimization of search parameters in his world-class chess program NIMZO as more rewarding than tuning the parameters of NIMZO's evaluation function. Similarly, Baxter, Tridgell, and Weaver (2000) consider the problem of "learning to search selectively" as a rewarding topic for further research. Surprisingly, this area is more or less still open research in game-playing. While there have been various attempts on the use of learning to increase the efficiency of search algorithms in other areas of Artificial Intelligence (Mitchell et al. 1983; Laird et al. 1987), this approach has so far been neglected in game playing.

Moriarty and Miikkulainen (1994) train a neural network with a genetic algorithm to prune unpromising branches of a search tree in Othello. Experiments show that their selective search maintains the same playing level as a full-width search. Similarly, Dahl (Dahl 2001) uses shape-evaluating neural networks to avoid searching unpromising moves in Go.

Buro (1995a) introduces the PROBCUT selective search extension, whose basic idea is to use the evaluations obtained by a shallow search to estimate the values that will be obtained by a deeper search. The relation between these estimates is computed by means of linear regression from the results of a large number of deep searches. During play, branches that seem unlikely to produce a result within the current alpha-beta search window are pruned. In subsequent work, Buro (1999a) generalized these results to allow pruning at multiple levels and with multiple significance thresholds. Again, the technique proves to be useful in Othello, where the correlation between shallow and deep evaluations seems to be particularly strong. However, this evaluation stability might also be observed in other games where a quiescence search is used to evaluate only stable positions (Buro 2000, personal communication). It is an open question how

[3] For more information about alpha-beta techniques and related search algorithms see (Schaeffer 2000) for a quick introduction and (Schaeffer 1989) and (Plaat 1996) for details.

well these algorithms generalize to other games in comparison to competing successful selective search techniques (like, e.g., null-move pruning).

Other researchers took a less radical approach by trying to learn move ordering heuristics for increasing the efficiency of alpha-beta search. More efficient move orderings will not directly change the outcome of a fixed-depth alpha-beta search but may result in significantly faster searches. The saved time can then be used for deeper searches, which may lead to better play. Greer, Ojha, and Bell (1999) train a neural network to learn to identify the important regions of a chess position and order moves according to the influence they exhibit upon these regions. Inuzuka et al. (1999) suggest the use of *inductive logic programming*[4] for learning a binary preference relation between moves and present some preliminary results with an Othello player. A similar approach was previously suggested by Utgoff and Heitman (1988) who used multivariate decision trees for representing the learned predicates. However, they used the learned preference predicate not for move ordering in a search algorithm, but to directly select the best move in a given position (as in (Nakano et al. 1998)). As the learned predicate need not be transitive or even symmetric (cf. Section 4.2), some heuristic procedure has to be used to turn the pair-wise comparisons of all successor states into a total order. One could, for example, treat them as results between players in a round-robin tournament and compute the winner, thereby relying on some typical tie-breaking methods for tournaments (like the Sonneborn-Berger system that is frequently used in chess).

However, for the purposes of move ordering in search, conventional techniques such as the history heuristic (Schaeffer 1983) or the killer heuristic (Akl and Newborn 1977), are already quite efficient and hard to beat (Schaeffer 2000), so that it is doubtful that above-mentioned learning techniques will find their way into competitive game-playing programs. Nevertheless, they are a first step into the right direction and will hopefully spawn new work in this area. For example, the tuning of various parameters that control search extension techniques would be a worth-while goal (Donninger 1996, personal communication). It may turn out that this is harder than evaluation function tuning—which we will discuss in the next session—because it is hard to separate these parameters from the search algorithm that is controlled by them.

4 Evaluation Function Tuning

The most extensively studied learning problem in game playing is the automatic adjustment of the weights of an evaluation function. Typically, the situation is as follows: the game programmer has provided the program with a library of routines that compute important properties of the current board position (e.g., the number of pieces of each

[4]Inductive logic programming (ILP) refers to a family of learning algorithms that are able to induce PROLOG programs and can thus rely on a more expressive concept language than conventional learning algorithms, which operate in propositional logic (Muggleton 1992; Lavrač and Džeroski 1993; De Raedt 1995; Muggleton and De Raedt 1994). Its strengths become particularly important in domains where a structural description of the training objects is of importance, like, e.g., in describing molecular structures (Bratko and King 1994; Bratko and Muggleton 1995). They also seem to be appropriate for many game playing domains, in which a description of the spatial relation between the pieces is often more important than their actual location.

kind on the board, the size of the territory controlled, etc.). What is not known is how to combine these pieces of knowledge and how to quantify their relative importance.

The known approaches to solving this problem can be categorized along several dimensions. In what follows, we will discriminate them by the type of training information they receive. In *supervised learning* the evaluation function is trained on information about its correct values, i.e., the learner receives examples of positions or moves along with their correct evaluation values. In *comparison training*, it is provided with a collection of move pairs and the information which of the two is preferable. Alternatively, it is given a collection of training positions and the moves that have been played in these positions. In *reinforcement learning*, the learner does not receive any direct information about the absolute or relative value of the training positions or moves. Instead, it receives feedback from the environment whether its moves were good or bad. In the simplest case, this feedback simply consists of the information whether it has won or lost the game. *Temporal-difference learning* is a special case of reinforcement learning which can use evaluation function values of later positions to reinforce or correct decisions earlier in the game. This type of algorithm, however, has become so fashionable for evaluation function tuning that it deserves its own subsection. Finally, in Section 4.5, we will discuss a few important issues for evaluation function training.

4.1 Supervised learning

A straight-forward approach for learning the weights of an evaluation function is to provide the program with example positions for which the exact value of the evaluation function is known. The program then tries to adjust the weights in a way that minimizes the error of the evaluation function on these positions. The resulting function, learned by linear optimization or some non-linear optimization technique like back-propagation training for neural networks, can then be used to evaluate new, previously unseen positions.

Mitchell (1984) applied such a technique to learning an evaluation function for the game of Othello (see also (Frey 1986)). He selected 180 positions that occurred in tournament play after the 44th move and computed the exact minimax value for these positions with exhaustive search. These values were then used for computing appropriate weights of the 28 features of a linear evaluation function by means of regression.

In the game of Othello, Lee and Mahajan (1988) relied on BILL, a—for the time—very strong program[5] that used hand-crafted features, to provide training examples by playing a series of games against itself. Variety was ensured by playing the first 20 plies randomly. Each position was labeled as won or lost, depending on the actual outcome of the game, and represented with four different numerical feature scores (Lee and Mahajan 1990). The covariance matrix of these features was computed from the training data and this information was used for learning several Bayesian discriminant functions (one for each ply from 24 to 49), which estimated the probability of winning in a given position. The results showed a great performance improvement over the

[5]In 1997, BILL was tested against Buro's LOGISTELLO and appeared comparably weak: running on equal hardware and using 20 minutes per game BILL was roughly on par with 4-ply LOGISTELLO, which only used a couple of seconds per game (Buro 2000, personal communication).

original program. A similar procedure was used by Buro (1995b). He further improved classification accuracy by building a complete position tree of all games. Interior nodes were labelled with the results of a fast negamax search, which he also used for his approach to opening book learning (see Section 2.2 and (Buro 2001)).

Tesauro and Sejnowski (1989) trained the first neural-network evaluation function of the program that has later developed into TD-GAMMON by providing it with several thousand expert-rated training positions. The training positions were from several sources, textbook games, Tesauro's games, and games of the network. In each position, Tesauro rated several moves, in particular the best and the worst moves in the position. He underlined the importance of having examples for bad moves in addition to good moves (otherwise the program would tend to rate each move as good) as well as the importance of having manually constructed training examples that are suitable for guiding the learner into specific directions. In particular, important but rarely occurring concepts should be trained by providing a number of training positions that illustrate the point. Interesting is also Tesauro's technique of handling unlabelled examples (i.e., legal moves in a training position that have not been rated by the expert). He claims that they are necessary for exposing the network to a wider variety of training positions (thus escaping local minima). Hence, he randomly included some of these positions, even though he could only label them with a random value.

The main problem of supervised approaches is to obtain appropriate labels for the training examples. Automated approaches—computing the game-theoretic value or training on the values of a strong program—are in general only of limited utility.[6] The manual approach on the other hand is very time-intensive and costly. Besides, experts are not used to evaluate game states in a quantitative way. While one can get a rough, qualitative estimate for the value of a game state (e.g., the qualitative annotations $=$, $\pm$, $+-$ and the like that are frequently used for evaluating chess positions), it is hard for an expert to evaluate a position in a quantitative way (e.g., "White's advantage is worth 0.236 pawns.").

In some works, providing exact values for training the evaluation function was not considered as important, but instead the training examples were simply regarded as means for signaling the learner into which direction it should adjust its weights. For example, in the above-mentioned work, Tesauro and Sejnowski (1989) did not expect their network to exactly reproduce the given training information. In fact, overfitting the training data did hurt the performance on independent test data, a common phenomenon in machine learning. Likewise, in (Dahl 2001), a neural network is trained to evaluate parts of Go positions, so-called receptive fields. Its training input consists of a number of positive examples, receptive fields in which the expert played into its center, and for each of them a negative example, another receptive field from the same position, which was chosen randomly from the legal moves that the expert did not play.

Obviously, it is quite convenient to be able to use expert moves a training information. While Dahl's above-mentioned approach uses information about expert moves for explicitly selecting positive and negative training examples, *comparison training*

[6] An exception is the game of poker where the computation of the correct action in *hindsight*—if the opponents' cards are known—is fairly trivial. This approach was used in Waterman's (1970) poker player (cf. Section 5.3).

refers to approaches in which this information can be directly used for minimizing the error of the evaluation function.

4.2 Comparison training

Tesauro (1989a) introduced a new framework for training evaluation functions, which he called *comparison training*. Like in (Utgoff and Heitman 1988, see Section 3), the learner is not given exact evaluations for the possible moves (or resulting positions) but is only informed about their relative order. Typically, it receives examples in the form of move pairs along with a training signal as to which of the two moves is preferable. However, the learner does not learn an explicit preference relation between moves as in (Utgoff and Heitman 1988), but tries to use this kind of training information for tuning the parameters of an evaluation function (Utgoff and Clouse 1991). Thus, the learner receives less training information than in the supervised setting, but more information than in the reinforcement learning setting, which will be discussed in the next two sections.

In the general case, there are two problems with such an architecture: efficiency and consistency. For computing the best among n moves, the program has to compare $n^2 - 1$ move pairs. Furthermore, the predicted preferences need not be transitive: the network might prefer move a over b, b over c, and c over a. In fact, it may not even be symmetric: the network might prefer a over b and b over a, depending on the order in which the moves are provided as inputs to the network.

Tesauro (1989a) solved these problems elegantly by enforcing that the network architecture is symmetric (the weights used for processing the first move must be equivalent to the weights used for processing the second move) and separable (the two sub-networks only share the same output unit). In fact, this means that the same sub-network is used in both halves of the entire network, with the difference that the weights that connect the hidden layer of one sub-network to the output unit are multiplied by -1 in the other sub-network. In practice, this means that only one sub-network has to be stored and that this network actually predicts an absolute evaluation for the move that is encoded at its input. Consequently, only n evaluations have to be performed for determining the best move. An experimental comparison between this training procedure and the supervised procedure described in the previous section produced a substantial performance gain, even in cases where a simpler network architecture was used (e.g., in some experiments, the comparison-trained networks did not use a hidden layer). Eventually, this training procedure yielded the NEUROGAMMON program, the first game-playing program based primarily on machine learning technology that won in tournament competition (at the First Computer Olympiad, see Tesauro 1989b, 1990; Levy and Beal 1989)

More recently, a comparison training procedure was used for training a chess evaluation function, where it made a crucial difference in the famous 1997 DEEP BLUE vs. Kasparov match. This work is described in detail in (Tesauro 2001).

Conceptually, the training information in Tesauro's comparison-trained networks consists of a move pair. However, in effect, the learner is only given the move played in a collection of training positions. This information is then used to minimize an error function over all legal moves in that position (see (Tesauro 2001) for details). Likewise,

Utgoff and Clouse (1991) show how to encode state (or move) preference predicates as a system of linear inequalities that is no more complex than learning from single states.

Similar techniques for reducing the prediction error on a set of moves have already been used earlier. Once more, it was Samuel (1967) who first realized such a technique in the second version of his checkers player, where a structure of stacked linear evaluation functions was trained by computing a correlation measure based on the number of times the feature rated an alternative move higher than the expert move.

Nitsche (1982) made this approach clearer by explicitly defining the so-called *selection function*, which is set to 1 for the move that was actually played in the training position, and to 0 for all other moves. He then computed the least-squares differences between the evaluation functions estimates of all move values and this selection function. Marsland (1985) extends this work by suggesting that the ordering of alternative moves should be included into the learning process as well. He proposes cost functions that give a higher penalty if the expert move has a low ranking than if it is considered as one of the best moves. He also favors a cost function that makes sure that changing the rank of a move near the top has a bigger effect than changing the rank of a move near the bottom.

Similarly, van der Meulen (1989) criticized Nitsche's approach because of the use of a discrete selection function. Instead he proposes to calculate the weights for the evaluation functions using a set of inequalities that define a hyper-rectangle in parameter space, in which the evaluation function would choose the same move that is specified in the training positions. The intersection of several such hyper-rectangles defines a region where the evaluation function finds the correct move for several test positions. Van der Meulen's algorithm greedily identifies several such regions and constructs a suitable evaluation function for each of them. New positions are then evaluated by first identifying an appropriate region and then using the associated evaluation function. However, this algorithm has not been tested, and it is an open question whether the number of optimal regions can be kept low enough for an efficient application in practice.

Hsu et al. (1990a) used an automatic tuning approach for the chess program DEEP THOUGHT, because they were convinced that it is infeasible to correctly reflect the interactions between the more than 100 weights of the evaluation functions. Like Nitsche (1982) they chose to compute the weights by minimizing the squared error between the program's and a grandmaster's choice of moves. Whenever an alternative move received a higher evaluation than the grandmaster's move (after a 5 to 6 ply search) the weights of the dominant position (see Section 4.5) were adjusted to decrease this difference. The authors were convinced that their automatically tuned evaluation function is no worse than the hand-tuned functions of their academic competitors. However, they also remarked that the evaluation functions of top commercial chess programs are still beyond them, but hoped to close the gap soon. Subsequent work on DEEP THOUGHT's successor DEEP BLUE is described in (Tesauro 2001)

Schaeffer et al. (1992) report that the use of similar techniques for tuning a checkers evaluation function on 800 Grandmaster games yielded reasonable results, but was not competitive with their hand-tuned weight settings. The GOLEM Go program (Enderton 1991) used a very similar technique for training a neural network to forward-prune in

a one-ply search. However, instead of comparing the master's move to all alternatives (which can be prohibitively many in Go), they required that the master's move has to be better than a random alternative by a certain minimum margin.

Tunstall-Pedoe (1991) tried to optimize the weights of an evaluation function with *genetic algorithms*,[7] where the fitness of a set of weights was evaluated with the percentage of test positions for which the evaluation function chose the move in the training set. A similar technique was also suggested for training a neural network (Tiggelen 1991).

Comparison training worked well in many of the above-mentioned approaches but it also has some fundamental shortcomings. For example, Schaeffer et al. (1992) observed that a program trained on expert games will imitate a human playing style, which makes it harder for the program to surprise a human being. This point is of particular importance in a game like checkers, where a relatively high percentage of the games end in book draws. On the other hand, this fact could also be used for adapting one's play towards a particular opponent, as has been suggested in (Carmel and Markovitch 1993, see Section 6). Buro (1998) pointed out that the training positions should not only be representative for the positions encountered during *play*, but representative for the positions encountered during *search*. This means that a lot of lop-sided and counter-intuitive positions should nevertheless be included in the training set because they will be encountered by the program during an exhaustive search. Last but not least, using expert games for training makes the often unjustified assumption that the move of the master is actually the best and it also does not give you additional information about the ranking of the alternative moves, some of which might be at least as good as the move actually played. It would be much more convenient (and more elegant) if a player could learn from its own experience, simply from playing numerous games and using the received feedback (won or lost games) for adjusting its evaluation function. This is what *reinforcement learning* is about.

4.3 Reinforcement learning

Reinforcement learning (Sutton and Barto 1998) is best described by imagining an *agent* that is able to take several actions whose task is to learn which actions are most preferable in which states. However, contrary to the supervised learning setting, the agent does not receive training information from a domain expert. Instead, it may explore the different actions and, while doing so, will receive feedback from the environment—the so-called *reinforcement* or *reward*—which it can use to rate the success of its own actions. In a game-playing setting, the actions are typically the legal moves in the current state of the game, and the feedback is whether the learner wins or loses the game or by which margin it does so. We will describe this setting in more

[7] A genetic algorithm (Goldberg 1989) is a randomized search algorithm. It maintains a population of individuals that are typically encoded as strings of 0's and 1's. All individuals of a so-called generation are evaluated according to their fitness, and the fittest individuals have the highest chance of surviving into the next generation and of spawning new individuals through the genetic operators cross-over and mutation. For more details, see also (Kojima and Yoshikawa 2001), which discusses the use of genetic algorithms for learning to solve tsume-go problems.

detail using MENACE, the *Matchbox Educable Noughts And Crosses Engine* (Michie 1961, 1963), which learned to play the game of *tic-tac-toe* by reinforcement.

MENACE has one weight associated with each of the 287 different positions with the first player to move (rotated or mirrored variants of identical positions were mapped to a unique position). In each state, all possible actions (all yet unoccupied squares) are assigned a weight. The next action is selected at random, with probabilities corresponding to the weights of the different choices. Depending on the outcome of the game, the moves played by the machine are rewarded or penalized by increasing or decreasing their weight. Drawing the game was considered a success and was also reinforced (albeit by a smaller amount).

MENACE's first implementation consists of 287 matchboxes, one for each state. The weights for the different actions are represented with a number of beads in different colors, one color for each action. The operator chooses a move by selecting the matchbox that corresponds to the current move, and drawing a single bead from it. Moves that have a higher weight in this position have more beads of their color in the box and therefore a higher chance of being selected. After completing a game, MENACE receives reinforcement through the information whether it has won or lost the game. However, the reward is not received after each move, but at the end of the game. If the program makes a good move, this is not immediately pointed out, but it will receive a *delayed reward* by winning the game.[8] In MENACE's case, positions are rewarded or penalized by increasing/decreasing their associated weights by adding/removing beads to/from each matchbox that was used in the game.

The main problem that has to be solved by the learner is the so-called *credit assignment problem* (Minsky 1963), i.e., the problem of distributing the received reward to the actions that were responsible for it. This problem and some domain-specific approaches towards its solution already arose in Section 2.3: In a lost game, only one move might be the culprit that should receive all of the negative reward, while all other moves might have been good moves. However, it is usually not known to the program which move was the mistake. Michie had two proposals for solving the credit assignment problem. The first technique simply gives all moves in the game equal credit, i.e., the same amount of reinforcement (one bead) is added (removed) from all boxes that have been used in the won (lost) game. The second technique assumes that positions later in the game have a bigger impact on the outcome than positions earlier in the game. This was realized by initializing positions earlier in the game with a higher number of beads, so that adding or removing a single bead results in a smaller change in the weights for the actions of that state. This simple technique does not prevent good positions from receiving negative feedback (if a mistake was made later in the game) or bad positions from positive reward (if the game was won because the opponent did not

[8] In card games like Hearts, feedback is often available after each individual move in the form of whether the player made the trick or how many points he made in the trick Kuvayev (1997) made use of this information for training an evaluation function that predicts the number of points the program will make by playing each card. In this setting, the distinction between supervised learning and reinforcement learning becomes unclear: on the one hand, the learner learns from the reinforcement it receives after each trick, while on the other hand it can use straight-forward supervised learning techniques to do so. However, if we consider the possibility of "shooting the moon", where one player makes all the negative points which are then counted as positive, the reward has to be delayed because it may not be clear immediately after the trick, whether the points in it are positive or negative.

exploit the mistake). However, the idea is that after many games, good positions will have received more positive than negative reward and vice versa, so that the evaluation function eventually converges to a reasonable value.

In the mean-time, convergence theorems for reinforcement learning have confirmed this proposition (Sutton and Barto 1998). Michie, however, had to rely on experimental evidence, first with the physical machine playing against a human tutor, then with a simulation program playing tournaments against a random player and against an independently learning copy of itself. The results showed that MENACE made continuous progress. After a few hundred games, the program produced near-expert play (Michie 1963).

However, MENACE has several obvious shortcomings. The first is the use of a look-up table. Every state has to be encoded as a separate table entry. While this is feasible for tic-tac-toe, more complex games need some sort of generalization over different states of the games because in chess, for example, it is quite unlikely that the same middlegame position will appear twice in a program's life-time. More importantly, however, training on the outcome of the game is very slow and a large number of games are needed before position evaluations converge to reasonable values. *Temporal-difference learning* introduced a significant improvement in that regard.

4.4 Temporal-difference learning

A small revolution happened in the field of reinforcement learning when Gerald Tesauro presented his first results on training a backgammon evaluation function by temporal-difference learning (Tesauro 1992a, 1992b). The basic idea of temporal-difference learning is perhaps best understood if one considers the supervised learning approach taken in (Lee and Mahajan 1988) where each encountered position is labelled with the final outcome of the game. Naturally, this procedure is quite error-prone, as a game in which the player is better most of the time can be lost by a single mistake in the endgame. All positions of the game would then be labelled as bad, most of them unjustly.[9] If, on the other hand, each position is trained on the position evaluation several moves deeper in the game, the weight adjustments are performed more intelligently: positions close to the final position will be moved towards the evaluation of the final position. However, in the case of a mistake late in the game, the effect of the final outcome will not be visible in the early phases of the game.

Sutton's (1988) TD(λ) learning framework, provides an elegant formalization of these idea. He introduces the parameter λ which is used to weight the influence of the current evaluation function value for weight updates of previous moves. In the simple case of a linear evaluation function

$$F(x) = \sum_i w_i \times f_i(x)$$

[9] In the experiments of Lee and Mahajan (1988), the initial phase of random play often yielded lop-sided positions which could easily be evaluated by their training program. However, in other games, one has to rely on human training information, which is naturally error-prone.

the weights are updated after each move according to the formula

$$w_{i,t+1} = w_{i,t} + \alpha(F(x_{t+1}) - F(x_t)) \sum_{k=1}^{t} \lambda^{t-k} f_i(x_k)$$

where α is a learning rate between 0 and 1 and $F(x_t)$ is the evaluation function value for the position after the t-th move.

To understand this, consider for a moment the case $\lambda = 0$. In this case, the sum at the end reduces to the single value $f_i(x_t)$. The entire formula then specifies that each feature weight is updated by an amount that is proportional to the product of the evaluation term $f_i(x_t)$ and the difference between the successive predictions $F(x_t)$ and $F(x_{t+1})$. The result of this computation is that the weights are adjusted in the direction of the temporal difference according to the degree to which they contributed to the final evaluation $F(x_t)$. At a setting of $\lambda \neq 0$, the degree to which the feature contributed to the final evaluation of previous positions is also taken into consideration, weighted by λ^{t-k} if the respective position occurred k moves ago. A value of $\lambda = 1$ assigns the same amount of credit or blame to all previous moves. It can be shown that in this case the sum of these updates over an entire games corresponds to a supervised learning approach where each position is labelled with the final outcome of the game (which is assigned to x_{n+1} if the game lasted for n moves). For more details, in particular for a discussion of TD(λ)'s sound formal basis and its various convergence properties see (Sutton and Barto 1998).

Tesauro has applied this procedure successfully to train a neural-net evaluation function for his backgammon player. After solving several of the practical issues that are involved in training multi-layer neural networks with TD(λ) on a complex task (Tesauro 1992a), the resulting program—TD-GAMMON—clearly surpassed its predecessors, in particular the Computer Olympiad champion NEUROGAMMON, which was trained with comparison training (Tesauro 1989a, 1990), and it did so entirely from playing against itself. In fact, early versions of TD-GAMMON, which only used the raw board information as features, already learned to play as well as NEUROGAMMON, which used a sophisticated set of features. Adding these features to the input representation further improved TD-GAMMON's playing strength (Tesauro 1992b). Over time (Tesauro 1994, 1995), TD-GAMMON not only increased the number of training games that it played against itself, but Tesauro also increased the search depth and changed the network architecture, so that TD-GAMMON reached world-championship strength.

However, despite TD-GAMMON's overwhelming success, its superiority to its predecessors that were trained with supervised learning or comparison training does not allow the conclusion that temporal-difference learning is the best solution for all games. For example, Samuel (1967), in the follow-up paper on his famous checkers player, had arrived at a different result: comparison training from about 150,000 expert moves appeared to be faster and more reliable than temporal-difference learning from self-play. Similarly, Utgoff and Clouse (1991) demonstrated that a variant of comparison training performed better than temporal-difference learning on a simple problem-solving task. They proposed an integration of both methods that uses comparison training only when the temporal-difference error exceeds a user-defined threshold, and showed that

this approach outperformed its predecessors. They also showed that the number of queries to the expert (needed for comparison training) decreases over time.

Many authors have tried to copy TD-GAMMON's learning methodology to other games, such as tic-tac-toe (Gherrity 1993), Othello (Isabelle 1993; Leouski and Utgoff 1996), chess (Gherrity 1993; Schmidt 1994), Go (Schraudolph et al. 1994), connect-4 (Gherrity 1993; Sommerlund 1996), checkers (Lynch 1997), or Hearts (Kuvayev 1997) with mixed success. An application using TD-learning to learn to evaluate the safety of groups in Go from self-play is discussed in (Dahl 2001).

None of these successors, however, achieved a performance that was as impressive as TD-GAMMON's. The reason for this seems to be that backgammon has various characteristics that make it perfectly suited for learning from self-play. Foremost among these are the fact that it only requires a very limited amount of search and that the dice rolls guarantee sufficient variability in the games so that all regions of the feature space are explored.

Chess, for example, does not share these properties. Although this problem has been known early on and, in fact, several solution attempts have been known since the days of Samuel, it took several years and many unsuccessful attempts before a successful chess program could be trained by self-play. For example, Gherrity (1993) tried to train a neural network with Q-learning (a temporal-difference learning procedure that is quite similar to TD(0); Watkins and Dayan 1992) against GNUCHESS, but, after several thousand games, his program was only able to achieve 8 draws by perpetual check. The NEUROCHESS program (Thrun 1995) also used TD(0) for training a neural network evaluation function by playing against GNUCHESS. Its innovation was the use of a second neural network that was trained to predict the board position two plies after the current position (using 120,000 expert games for training). The evaluation function was then trained with its own estimates for the predicted positions using a TD(0) algorithm. NEUROCHESS steadily increased its winning percentage from 3% to about 25% after about 2000 training games.

Finally, Baxter, Tridgell, and Weaver (1998a) were able to demonstrate a significant increase in playing strength using TD-learning in a chess program. Initially, their program, KNIGHTCAP, knew only about the value of the pieces and was rated at around 1600 on an Internet chess server. After about 1000 games on the server, its rating had improved to over 2150, which is an improvement from an average amateur to a strong expert. The crucial ingredients to KNIGHTCAP's success were the availability of a wide variety of training partners on the chess server and a sound integration of TD-learning into the program's search procedures. We will briefly discuss these aspects in the next section but more details can be found in (Baxter, Tridgell, and Weaver 2001).

4.5 Issues for evaluation function learning

There are many interesting issues in learning to automatically tune evaluation functions. We will discuss some of them in a little more detail, namely the choice of linear vs. non-linear evaluation functions, the issue of how to train the learner optimally, the integration of evaluation function learning and search, and automated ways for constructing the features used in an evaluation function.

4.5.1 Linear vs. non-linear evaluation functions

Most conventional game-playing programs depend on fast search algorithms and thus require an evaluation function that can be quickly evaluated. A linear combination of a few features that characterize the current board situation is an obvious choice here. Manual tuning of the weights of a linear evaluation function is comparably simple, but already very cumbersome. Not only the individual evaluation terms may depend on each other, so that small changes in one weight may affect the correctness of the settings of other weights, but also all weights depend on the characteristics of the program in which they are used. For example, the importance of being able to recognize tactical patterns such as fork threats may decrease with the program's search depth or depend on the efficiency of the program's quiescence search.

However, advances in automated tuning techniques have even made the use of non-linear function approximators feasible. Samuel (1967) already suggested the use of *signature tables*, a non-linear, layered structure of look-up tables. Clearly, non-linear techniques have the advantage that they can approximate a much larger class of functions. However, they are also slower in training and evaluation. Hence the question is whether they are necessary for a good performance. In many aspects, this problem is reminiscent of the well-known *search/knowledge trade-off* (Berliner 1984; Schaeffer 1986; Berliner et al. 1990; Junghanns and Schaeffer 1997).

Lee and Mahajan (1988) interpreted the big improvement that they achieved with Bayesian learning over a hand-crafted, linear evaluation function as evidence that a non-linear evaluation function is better than a linear evaluation function. In particular, they showed that the covariance matrices upon which their Bayesian evaluation function is based, exhibit positive correlations for all terms in their evaluation function, which refutes the independence assumptions that some training procedures have to make.

On the other hand, Buro (1995b) has pointed out that a quadratic discriminant function may reduce to a linear form if the covariance matrices for winning and losing are (almost) equal. In fact, in his experiments in Othello, he showed that Fisher's linear discriminant obtains even better results than a quadratic discriminant function because both are of similar quality but the former is faster to evaluate (which allows the program to search deeper).[10] Consequently, Buro (1995b) argues that the presence of feature correlations, which motivated the use of non-linear functions in (Lee and Mahajan 1988), is only a necessary but not a sufficient condition for switching to a non-linear function. It only pays off if the covariance matrices are different.

Likewise, Tesauro (1995) remarked that his networks tend to learn elementary concepts first, and that these can typically be expressed with a linear function. In fact, in his previous work (Tesauro 1989a), he noticed that a single-layer perceptron trained with comparison training can outperform a non-linear multi-layer perceptron that learned in a supervised way. Nevertheless, Tesauro (1998) is convinced that eventually non-linear structure is crucial to a successful performance. He claims that the networks of Pollack and Blair (1998) that were trained by a stochastic hill-climbing procedure (cf. next sec-

[10] Buro (1995b) also showed that a logistic regression obtains even better results because it does not make any assumptions on the probability distributions of the features.

tion) are inferior to those trained with TD-learning because they are unable to capture non-linearity.

A popular and comparably simple technique for achieving non-linearity, also originating with Samuel, is to use different evaluation functions for different game phases. Tesauro's early version of TD-GAMMON, for example, ignored important aspects like doubling or the running game (when the two opponents' tiles are already separated) because sufficiently powerful algorithms existed for these game parts. Boyan (1992) took this further and showed that the use of different networks for different subtasks of the game can improve over learning with a single neural network for the entire game. For example, he used 8 networks in tic-tac-toe and trained each of them to evaluate positions with N pieces on the board ($N = 1..8$). In backgammon, he used 12 networks for different classes of pipcounts.[11] Similarly, Lee and Mahajan (1988) used 26 evaluation functions, one for each ply from 24 to 49, for training their Othello player BILL. Training information was smoothed by sharing training examples between the positions that were within a distance of 2 plies.

There are several practical problems that have to be solved with such approaches. One is the so-called *blemish effect* (Berliner 1979), which refers to the problem of keeping the values returned by the different evaluation functions consistent.

4.5.2 Training strategies

Another important issue is how to provide the learner with training information that is on the one hand focussed enough to guarantee fast convergence towards a good evaluation function, but on the other hand provides enough variation to allow the learning of generally applicable functions. In particular in learning from self-play it is important to ensure that the evaluation function is subject to enough variety during training that prevents it from settling at a local minimum. In reinforcement learning, this problem is known as the trade-off between *exploration* of new options and *exploitation* of already acquired knowledge.

Interestingly, Tesauro's TD-GAMMON did not have this problem. The reason is that the roll of the dice before each move ensured a sufficient variety in the training positions so that learning from self-play worked very well. In fact, Pollack and Blair (1998) claimed that TD-GAMMON's self-play training methodology was the only reason for its formidable performance. They back up their claim with experiments in which they self-train a competitive backgammon playing network with a rather simple, straight-forward stochastic hill-climbing search instead of Tesauro's sophisticated combination of temporal-difference search and back-propagation. The methodology simply performs random mutations on the current weights of the network and lets the new version play against the old version. If the former wins in a series of games, the weight changes are kept, otherwise they are undone. Although Tesauro (1998) does not entirely agree with the conclusions that Pollack and Blair (1998) derive from this experiment, the fact that such a training procedure works at all is remarkable.

On the other hand, Epstein (1994c) presents some experimental evidence that self-training indeed does not perform well in the case of deterministic games such as tic-

[11] Pipcounts are a commonly used metric for evaluating the progress in a backgammon game.

tac-toe or achi. Her game-learning system HOYLE (Epstein 1992), which is based on the FORR architecture for learning and problem solving (Epstein 1994a), was trained with self-play, a perfect opponent, a randomly playing opponent, and various fallible opponents that made random moves $n\%$ of the time and played the other moves perfectly. Her result showed that neither the perfect or random trainers, nor learning by self-play produced optimal results. Increasing the percentage of random moves of the fallible trainers, which encourages exploration, usually increased the number of games they lost without increasing the number of games won against an expert benchmark player. As a consequence, Epstein (1994c) proposed *lesson-and-practice* training, in which phases of training with an expert player are intertwined with phases of self-play, and demonstrated its superiority. More details on HOYLE can be found in Section 5.2 and, in particular, in (Epstein 2001).

Potential problems with self-play were already noted earlier, and a variety of alternative strategies were proposed. Samuel (1959) suggested a simple strategy: his checkers player learned against a copy of itself that had its weights frozen. If, after a certain number of games, the learning copy made considerable progress, the current state of the weights was transferred to the frozen copy, and the procedure continued. Occasionally, when no progress could be detected, the weights of the evaluation function were changed radically by setting the largest weight to 0. This procedure reduced the chance of getting stuck in a local optimum. A similar technique was used by Lynch (1997).

Lee and Mahajan (1988) and Fogel (1993) ensured exploration of different regions of the parameter space by having their Othello and tic-tac-toe players learn from games in which the first moves were played randomly. Similarly, Schraudolph et al. (1994) used Gibbs sampling for introducing randomness into the move selection. Boyan (1992) inserted randomness into the play of his program's tic-tac-toe training partner. Angeline and Pollack (1994) avoided this problem by evolving several tic-tac-toe players in parallel with a genetic algorithm. The fitness of the players of each generation was determined by having them play a tournament.

Finally, the success of the chess program KNIGHTCAP (Baxter et al. 1998b) that employed TD-learning was not only due to the integration of TD-learning into search (cf. next section) but also to the fact that the program learned by playing on an Internet chess server where it could learn from a wide variety of opponents of all playing strengths. As human players tend to match with opponents of approximately their own strength (i.e., players with a similar rating), the strength of KNIGHTCAP's opposition increased with its own playing strength, which intuitively seems to be a good training procedure (Baxter, Tridgell, and Weaver 2001).

4.5.3 Evaluation function learning and search

In backgammon, deep searches are practically infeasible because of the large branching factor that is due to the chance element introduced by the use of dice. However, deep searches are also beyond the capabilities of human players whose strength lies in estimating the positional value of the current state of the board. Contrary to the successful chess programs, who can easily out-search their human opponent but still trail her ability of estimating the positional merits of the current board configuration,

TD-GAMMON was able to excel in backgammon for the same reasons that humans play well: its grasp of the positional strengths and weaknesses was excellent.

However, in games like chess or checkers, deep searches are necessary for expert performance. A problem that has to be solved for these games is how to integrate learning into the search techniques. In particular in chess, one has the problem that the position at the root of the node often has completely different characteristics than the evaluation of the node. Consider the situation where one is in the middle of a queen trade. The current board situation will evaluate as "being one queen behind", while a little bit of search will show that the position is actually even because the queen can easily be recaptured within the next few moves. Straight-forward application of an evaluation function tuning algorithm would then simply try to adjust the evaluation of the current position towards being even. Clearly, this is not the right thing to do because simple tactical patterns like piece trades are typically handled by the search and need not be recognized by the evaluation function.

The solution for this problem is to base the evaluation on the *dominant position* of the search. The dominant position is the leaf position in the search tree whose evaluation has been propagated back to the root of the search tree. Most conventional search-based programs employ some form of quiescence search to ensure that this evaluation is fairly stable. Using the dominant position instead of the root position makes sure that the estimation of the weight adjustments is based on the position that was responsible for the evaluation of the current board position. Not surprisingly, this problem has already been recognized and solved by Samuel (1959) but seemed to have been forgotten later on. For example, Gherrity (1993) published a thesis on a system architecture that integrates temporal-difference learning and search for a variety of games (tic-tac-toe, Connect-4, and chess), but this problem does not seem to be mentioned.

Hsu et al. (1990b) implemented the above solution into their comparison training framework. They compared the dominant position of the subtree starting with the grandmaster's move to the dominant position of any alternative move at a shallow 5 to 6 ply search. If an alternative move's dominant position gets a higher evaluation an appropriate adjustment direction is computed in the parameter space and the weight vector is adjusted a little in that direction. For a more recent technique integrating comparison training with deeper searches see (Tesauro 2001).

As far as temporal-difference learning is concerned, it was only recently that Beal and Smith (1997) and Baxter, Tridgell, and Weaver (1998b) independently re-discovered this technique for a proper integration of learning and search. Beal and Smith (1998) subsequently applied this technique for learning piece-values in shogi. A proper formalization of the algorithm can be found in (Baxter et al. 1998a) and in (Baxter, Tridgell, and Weaver 2001)

4.5.4 Feature construction

The crucial point for all approaches that tune evaluation functions is the presence of carefully selected features that capture important information about the current state of the game which goes beyond the location of the pieces. In chess, concepts like king safety, center control or mobility are commonly used for evaluating positions, and similar abstractions are used in other games as well (Lee and Mahajan 1988; Ender-

ton 1991). Tesauro and Sejnowski (1989) report an increase in playing strength of 15 to 20% when adding hand-crafted features that capture important concepts typically used by backgammon experts (e.g., pipcounts) to their neural network backgammon evaluation function. Although Tesauro later demonstrated that his TD(λ)-trained network could surpass this playing level without these features, re-inserting them brought yet another significant increase in playing strength (Tesauro 1992b). Samuel (1959) already concluded his famous study by making the point that the most promising road towards further improvements of his approach might be "*... to get the program to generate its own parameters for the evaluation polynomial*" instead of learning only weights for manually constructed features. However, in the follow-up paper, he had to concede that the goal of "*... getting the program to generate its own parameters remains as far in the future as it seemed to be in 1959*" (Samuel 1967).

Since then, the most important new step into that direction has been the development of automated training procedures for multi-layer neural networks. While the classical single-layer neural network—the so-called *perceptron* (Rosenblatt 1958; Minsky and Papert 1969)—combines the input features in a linear fashion to a single output unit, multi-layer perceptrons contain at least one so-called *hidden layer*, in which some or all of the input features are combined to intermediate concepts (Rumelhart and McClelland 1986; Bishop 1995). In the most common case, the three-layer network, the outputs of these hidden layer units are directly connected to the output unit. The weights to and from the hidden layer are typically initialized with random values. During training of the network, the weights are pushed into directions that facilitate the learning of the training signal and useful concepts emerge in the hidden layer. Tesauro (1992a) shows examples for two hidden units of TD-GAMMON that he interpreted as a race-oriented feature detector and an attack-oriented feature detector.

Typically, the learning of the hidden layers is completely unconstrained, and it is unclear which concepts will evolve. In fact, different runs on the same data may produce different concepts in the hidden layers (depending on the random initialization). Sometimes, however, it can be advantageous to impose some constraints on the network. For example, Schraudolph, Dayan, and Sejnowski (1994) report that they managed to increase the performance of their Go-playing network by choosing an appropriate network architecture that reflects various symmetries and translation-invariant properties of the Go board. Similarly, Leouski and Utgoff (1996) tried to exploit symmetries in the game of Othello by sharing the weights between eight sectors of the board.

The disadvantage of the features constructed in the hidden layers of neural networks is that they are not immediately interpretable. Several authors have worked on alternative approaches that attempt to create symbolic descriptions of new features. Fawcett and Utgoff (1992) discuss the ZENITH system, which automatically constructs features for a linear evaluation function for Othello. Each feature is represented as a formula in first-order predicate calculus. New features can be derived by decomposition, abstraction, regression, and specialization of old features. Originally, the program has only knowledge of the goal concept and the definitions of the move operators, but it is able to derive several interesting features from them, including a feature for predicting future piece mobility, which seems to be new in the literature (Fawcett and Utgoff 1992).

ELF (Utgoff and Precup 1998) constructs new features as conjunctions of a set of basic, Boolean evaluation terms. ELF starts with the most general feature that has *don't care*-values for all basic evaluation terms and therefore covers all game situations. It continuously adjusts the weight for each feature so that it minimizes its error on the incoming labelled training examples. It also keeps track which of the evaluation terms with *don't care*-values are most frequently present when such adjustments happen. ELF can remove these sources of error by spawning a new feature that is identical to its parent but also specifies that the evaluation term that contributed most to the error must not be present. Newly generated features start with a weight of 0, and features whose weight stays near 0 for a certain amount of time will be removed. The approach was used to learn to play checkers from self-play with limited success.

A similar system, the *generalized linear evaluation model* GLEM (Buro 1998), constructs a large number of simple Boolean combinations of basic evaluation terms. GLEM simply generates all feature combinations that occur with a user-specified minimum frequency, which is used to avoid overfitting of the training positions. For this purpose, GLEM uses an efficient algorithm that is quite similar to the well-known APRIORI data mining algorithm for generating frequent item sets in basket analysis problems (Agrawal et al. 1995). Contrary to ELF, where both the weights and the feature set are updated on-line and simultaneously, the features are constructed in batch and their weights are determined in a second pass by linear regression. The approach worked quite well for the Othello program LOGISTELLO, but Buro (1998) points out that it is particularly suited for Othello where important concepts can be expressed as Boolean combinations of the location of the pieces on the board. For other games, where one has to use more complicated atomic functions, this approach has not yet been tested.

Utgoff (2001) provides an in-depth description of the problem of automatic feature construction. Some of the techniques for pattern acquisition, which are discussed in more detail in the next section, can also be viewed in this context.

5 Learning Patterns and Plans

It has often been pointed out that game-playing systems can be classified along two dimensions, search and knowledge, which form a trade-off: perfect knowledge needs no search while exhaustive search needs no knowledge. For example, one can build a perfect tic-tac-toe player with a look-up table for all positions (no search) or with a fast alpha-beta search that searches each position until the outcome is determined (no knowledge). Clearly, both extremes are infeasible for complex games, and practical solutions have to use a little bit of both.

However, the region on this continuum which represents the knowledge-search combination that humans use, the so-called *human window*, is different from the region in which game playing programs excel (Clarke 1977; Michie 1982; Michie 1983; van den Herik 1988; Winkler and Fürnkranz 1998). Strong human players rely on deep knowledge, while game-playing programs typically use a lot of search. Even TD-GAMMON, whose strength is mostly due to the (learned) positional knowledge en-

coded in its evaluation function, investigates more moves than a human backgammon player.

The main problem seems to be the appropriate definition of background knowledge, i.e., the definition of the set of basic concepts or patterns that the program can use to formulate new concepts. Although many games have a rich vocabulary that describes important aspects of the game, these are often hard to define and even harder to formalize. In chess, for example, concepts like *passed pawn*, *skewer*, or *kings' opposition* are more or less common knowledge. However, even simple patterns like a *knight fork* are non-trivial to formalize.[12] Other games share similar problems. In Go, for example, it is very hard to describe long-distance influences between different groups of stones that are quite obvious to human players (e.g., a *ladder*). Obviously, if you cannot describe a concept in a given hypothesis language you cannot learn it. Nevertheless, there have been several attempts to build game-playing programs that rely mainly on pattern knowledge. We will review a few of them in this section.

5.1 Advice-taking

The learning technique that requires the least initiative by the learner is *learning by taking advice*. In this framework, the user is able to communicate abstract concepts and goals to the program. In the simplest case, the provided advice can be directly mapped on to the program's internal concept representation formalism. One such example is Waterman's poker player (Waterman 1970). Among other learning techniques, it provides the user the facility to directly add production rules to the game-playing program. Another classic example of such an approach is the work by Zobrist and Carlson (1973), in which a chess tutor could provide the program with a library of useful patterns using a chess programming language that looked a lot like assembly language. Many formalisms have since been developed in the same spirit (Bratko and Michie 1980; George and Schaeffer 1990; Michie and Bratko 1991), most of them limited to endgames, but some also addressing the full game (Levinson and Snyder 1993; Donninger 1996). Other authors have investigated similar techniques for incorporating long-term planning goals into search procedures (Kaindl 1982; Opdahl and Tessem 1994) or for specifying tactical patterns to focus search (Wilkins 1982).

While in the above-mentioned approaches the tutoring process is often more or less equivalent to programming in a high-level game programming language, typically advice-taking programs have to devote considerable effort into compiling the provided advice into their own pattern language. Thus they enable the user to communicate with the program in a very intuitive way that does not require any knowledge about the implementation of the program nor about programming in general.

The most prominent example for such an approach is the work by Mostow (1981). He has developed a system that is able to translate abstract pieces of advice in the card-game Hearts into operational knowledge that can be understood and directly accessed

[12] The basic pattern for a knight fork is a knight threatening two pieces, thereby winning one of them. In the endgame, these might simply be two unprotected pawns (unless one of them protects the other). In the middlegame, these are typically higher-valued pieces (protected or not). However, this definition might not work if the forking knight is attacked but not protected or even pinned. But then again, perhaps the attacking piece is pinned as well. Or the pinned knight can give a discovered check ...

by the machine. For example, the user can specify the hint "avoid taking points" and the program is able to translate this piece of advice into a simple, heuristic search procedure that determines the card that is likely to take the least number of points (Mostow 1983). However, his system is not actually able to play a game of Hearts. In particular, his architecture lacks a technique for evaluating and combining the different pieces of advice that might be applicable to a given game situation. In recent work, Fürnkranz, Pfahringer, Kaindl, and Kramer (2000) made a first step into that direction by identifying two different types of advice: *state abstraction advice* (advice that points the system to important features of the game, e.g., determine whether the queen is already out) and *move selection advice* (advice that suggests potentially good moves, e.g., flush the queen). They also proposed a straight-forward reinforcement learning approach for learning a mapping of state abstractions to move selections. Their architecture is quite similar to Waterman's poker player, which learns to prioritize action rules that operate on interpretation rules (Section 5.3). The use of weights for combining overlapping and possibly contradictory pieces of advice is also similar to HOYLE (Epstein 1992), where several Advisors are able to comment positively or negatively on the available set of moves. Initially, all of the Advisors are domain-independent and hand-coded, but HOYLE is able to autonomously acquire appropriate weights for the Advisors for the current games and to learn certain types of novel pattern-based Advisors (cf. Section 5.2 and, in particular, (Epstein 2001).

Similar in spirit to Mostow's work described above, Donninger (1996) introduces a very efficient interpreter of an extensible language for expressing chess patterns. Its goal is to allow the user to formulate simple pieces of advice for NIMZO, one of the most competitive chess programs available for the PC. The major difference from Mostow's work is the fact that people are not using language to express their advice, but can rely on an intuitive graphical user interface. The patterns and associated advice are then compiled into a byte code that can be interpreted at run-time (once for every root position to award bonuses for moves that follow some advice). One of the shortcomings of the system is that it is mostly limited to expressing patterns in propositional language, which does not allow to express relationships between pieces on the board (e.g., "put the rook behind the passed pawn").

A third approach that has similar goals is the use of so-called *structured induction* (Shapiro 1987). The basic idea is to induce patterns that are not only reliable but also simple and comprehensible by enabling the interaction of a domain expert with a learning algorithm, allowing it to focus the learner on relevant portions of the search space by defining new patterns that break up the problem into smaller, understandable sub-problems and use automated induction for each of them. We will discuss this approach in more detail in Section 5.5.

5.2 Cognitive models

Psychological studies have shown that the differences in playing strengths between chess experts and novices are not so much due to differences in the ability to calculate long move sequences, but to which moves they start to calculate (de Groot 1965; Chase and Simon 1973; Holding 1985; de Groot and Gobet 1996; Gobet and Simon 2001) For this pre-selection of moves chess players make use of patterns and accompanying

promising moves and plans. Simon and Gilmartin (1973) estimate the number of a chess expert's patterns to be of the order of 10,000 to 100,000. Similar results have been found for other games (Reitman 1976; Engle and Bukstel 1978; Wolff et al. 1984). On the other hand, most strong computer chess playing systems rely on a few hundred hand-coded patterns, typically the evaluation function terms, and attempt to counterbalance this deficiency by searching orders of magnitudes more positions than a human chess player does. In games like Go, where the search space is too big for such an approach, computer players make only slow progress.

Naturally, several researchers have tried to base their game-playing programs on findings in cognitive science. In particular early AI research has concentrated on the simulation of aspects of the problem-solving process that are closely related to memory, like perception (Simon and Barenfeld 1969) or retrieval (Simon and Gilmartin 1973). Several authors have more or less explicitly relied on the chunking hypothesis to design game-playing programs by providing it with access to a manually constructed database of patterns and chunks (Wilkins 1980; Wilkins 1982; Berliner and Campbell 1984; George and Schaeffer 1990). Naturally, attention has also focussed on automatically acquiring such chunk libraries.

Some of the early ideas on perception and chunk retrieval were expanded and integrated into CHREST (Gobet 1993), arguably the most advanced computational model of a chess player's memory organization. While CHREST mainly served the purpose of providing a computational model that is able to explain and reproduce the psychological findings, CHUMP (Gobet and Jansen 1994) is a variant of this program that is actually able to play a game. CHUMP plays by retrieving moves that it has previously associated with certain chunks in the program's pattern memory. It uses an eye-movement simulator similar to PERCEIVER (Simon and Barenfeld 1969) to scan the board into a fixed number (20) of meaningful chunks. New chunks will be added to a discrimination net (Simon and Gilmartin 1973) which is based on the EPAM model of memory and perception (Feigenbaum 1961). During the learning phase the move that has been played in the position is added into a different discrimination net and is linked to the chunk that contains the moved piece at its original location. In the playing phase the retrieved patterns are checked for associated moves (only about 10% of the chunks have associated moves). The move that is suggested by the most chunks will be played. Experimental evaluation (on a collection of games by Mikhail Tal and in the KQKR ending) was able to demonstrate that the correct move was often in the set of moves that were retrieved by CHUMP, although only rarely with the highest weight. One problem seems to be that the number of learned chunks increases almost linearly with the number of seen positions, while the percentage of chunks that have an associated move remains constant over time. This seems to indicate that CHUMP re-uses only few patterns, while it continuously generates new patterns.

TAL (Flinter and Keane 1995) is a similar system which also uses a library of chunks, which has also been acquired from a selection of Tal's games, for restricting the number of moves considered. It differs in the details of the representation of the chunks and the implementation of their retrieval. Here, the authors observed a seemingly logarithmic relationship between the frequency of a chunk and the number of occurrences of chunks with that frequency. This seems to confirm the above hypoth-

esis. Walczak (1991) suggest the use of a chunk library for predicting the moves of a (human) opponent. We will discuss his approach in Section 6.

CASTLE (Krulwich 1993) is based on Roger Schank's cognitive theory of explanation patterns (Schank 1986) and its computational model, *case-based reasoning* (Riesbeck and Schank 1989). In particular, it is based on the *case-based planning* model (Hammond 1989), in which plans that have been generalized from past experiences are re-used and, whenever they fail, continuously debugged. CASTLE consists of several modules (Krulwich et al. 1995). The threat detection component is a set of condition-action rules, each rule being specialized for recognizing a particular type of threat in the current focus of attention which is determined by another rule-based component. A counter-planning module analyzes the discovered threats and attempts to find counter-measures. CASTLE invokes learning whenever it encounters an *explanation failure*, e.g. when it turns out that the threat detection and the counter-planning components have failed to detect or prevent a threat. In such a case CASTLE uses a self-model to analyze which of its components is responsible for the failure, tries to find an explanation of the failure, and then employs explanation-based learning to generalize this explanation (Collins et al. 1993). CASTLE has demonstrated that it can successfully learn plans for interposition, discovered attacks, forks, pins and other short-term tactical threats.

Kerner (1995) also describes a case-based reasoning program based on Schank's explanation patterns (Schank 1986). The basic knowledge consists of a hierarchy of strategic chess concepts (like backward pawns etc.) that has been compiled by an expert. Indexed with each concept is an *explanation pattern (XP)*, roughly speaking a variabilized plan that can be instantiated with the parameters of the current position. When the system encounters a new position it retrieves XPs that involve the same concept and adapts the plan for the previous case to the new position using generalization, specialization, replacement, insertion and deletion operators. If the resulting XP is approved by a chess expert it will be stored in the case base.

HOYLE (Epstein 1992) is a game-playing system based on FORR, a general architecture for problem solving and learning (Epstein 1994a). It is able to learn to play a variety of simple two-person, perfect-information games. HOYLE relies on several hierarchically organized Advisors which are able to vote for or against some of the available moves. The Advisors are hand-coded, but game-independent. The importance of the individual Advisors is learned with an efficient, supervised weight-tuning algorithm Epstein (1994b, Epstein (2001) A special Advisor—PATSY—is able to make use of an automatically acquired chunk library, and comments in favor of patterns that are associated with wins and against patterns that are associated with losses.These chunks are acquired using a collection of so-called spatial *templates*, a meta-language that allows to specify which subparts of the current board configuration are interesting to be considered as pattern candidates. Patterns that occur frequently during play are retained and associated with the outcome of the game (Epstein et al. 1996). HOYLE is also able to generalize these patterns into separate, pattern-oriented Advisors. Similar to PATSY, another Advisor—ZONE RANGER—may support moves to a position whose *zones* have positive associations, where a zone is defined as a set of locations that can be reached in a fixed number of moves. The patterns and zones used by PATSY and ZONE RANGER are attempts to capture and model visual perception. There is also some empirical evidence that HOYLE exhibits similar playing and learning behavior

than human game players (Rattermann and Epstein 1995). A detailed description of HOYLE can be found in (Epstein 2001).

5.3 Pattern-based learning systems

There are a variety of other pattern-based playing systems that are not explicitly based on results in cognitive science. Quite well-known is PARADISE (Wilkins 1980, 1982), a pattern-based expert system that is able to solve combinations in chess with a minimum amount of search. It does not use learning but there a few other systems that rely upon the automatic formation of understandable pattern and rule databases that capture important knowledge for game-playing.

The classic example is Waterman's poker player (Waterman 1970). It has two types of rules, *interpretation rules* which map game states to abstract descriptors (by discretizing numerical values) and *action rules* which map state descriptors to actions. The program can learn these rules by accepting advice from the user or from a hand-coded program (cf. Section 5.1) or it can automatically construct rules while playing. Each hand played is rated by an oracle that provides the action that can be proved to be best in retrospect (when all cards are known). After the correct action has been determined, interpretation rules are adjusted so that they map the current state to a state descriptor that matches a user-provided so-called training rule, which is associated with the correct action. The program then examines its action rule base to discover the rule that made the incorrect decision. The mistake is corrected by specializing the incorrect rule, by generalizing one of the rules that are ordered before it or by inserting the training rule before it. During play, the program simply runs through the ordered list of action rules and performs the action recommended by the first rule matching the state description that has been derived with the current set of interpretation rules. In subsequent work, Smith (1983) used the same representation formalism and the same knowledge base for evolving poker players with a genetic algorithm.

Another interesting system is the chess program MORPH (Levinson and Snyder 1991; Gould and Levinson 1994). Its basic representational entity are *pattern-weight pairs (pws)*. Patterns are represented as conceptual position graphs, the edges being attack relationships and the nodes being pieces and important squares. Learned patterns can be generalized by extracting the common subtree of two patterns with similar weights or specialized by adding nodes and edges to the graph. Different patterns can be coupled using *reverse engineering*, so that they result in a chain of actions. The pattern weights are adjusted by a variant of temporal-difference learning, in which each pattern has its own learning rate, which is set by *simulated annealing* (i.e., the more frequently a pattern is updated, the slower becomes its learning rate). MORPH evaluates a move by combining the weights of all matched patterns into a single evaluation function value and selecting the move with the best value, i.e. it only performs a 1-ply look-ahead. The system has been tested against GNUCHESS and was able to beat it occasionally. The same architecture has also been applied to games other than chess (Levinson et al. 1992).

Kojima, Ueda, and Nagano (1997) discuss an approach for learning a set of rules for solving tsume-go problems. These are induced with an evolutionary algorithm, which was set up to reinforce rules that predict a move in a database. New rules are

constructed for situations in which no other rule matches the move played in the current position. Rules with high reliability can spawn children that specialize the rule by adding one condition and inherit some of the parent's activation score. Kojima and Yoshikawa (2001) showed that this approach is able to outperform patterns with fixed receptive fields, similar to those that are often used in Go programs to reduce the complexity of the input (see e.g., Stoutamire 1991; Schraudolph et al. 1994; or Dahl 2001).

5.4 Explanation-based learning

Explanation-based learning (EBL) (Mitchell et al. 1986) has been devised as a procedure that is able to learn patterns in domains with a rich domain theory. The basic idea is that the domain theory can be used to find an *explanation* for a given example move. Generalizing this explanation yields a pattern, to which the learner typically attaches the move that has been played. In the future, the program will play this (type of) move in situations where the pattern applies, i.e., in situations that share those features that are necessary for the applicability of this particular explanation. In game-playing domains, it is typically easy to represent the rules of the game as a formal domain theory. Hence, several authors have investigated EBL approaches that allow to learn useful patterns from a formalization of the rules of the game.

A very early explanation-based learning approach to chess has been demonstrated by Pitrat (1976a, 1976b). His program is able to learn definitions for simple chess concepts like skewer or fork. The concepts are represented as generalized move trees, so-called *procedures*. To understand a move the program tries to construct a move tree using already learned procedures. This tree is then simplified (unnecessary moves are removed) and generalized (squares and pieces are replaced with variables according to some predefined rules), in order to produce a new procedure that is applicable to a variety of similar situations.

Pitrat's program successfully discovered a variety of useful tactical patterns. However, it turned out that the size of the move trees grows too fast once the program has learned a significant number of patterns of varying degrees of utility. This *utility problem* is a widely acknowledged difficulty for explanation-based learning algorithms (Minton 1990) and pattern-learning algorithms in general. Interestingly, it has led Pitrat to the conclusion that the learning module works well, but a more sophisticated method for dealing with the many procedures discovered during learning is needed (Pitrat 1977).

Minton (1984) reports the same problem of learning too many too specialized rules with explanation-based learning for the game of Go-moku. His program performs a backward analysis after the opponent has achieved it goal (to have five stones in a row). It then identifies the pattern that has triggered the rule which has established that the opponent has achieved a goal. From this pattern all features that have been added by the opponent's last move are removed and all conditions that have been necessary to enable the opponent's last move are added. Thus the program derives a pattern that allows to recognize the danger before the opponent has made the devastating move. If the program's own last move has been forced, the pattern is again changed by deleting the effects of this move and adding the prerequisites for this move. This process of

pattern regression is continued until one of the program's moves has not been forced. The remaining pattern will then form the condition part of a rule that recognizes the danger before the forced sequence of moves that has achieved the opponent's goal.

However, the limitation to tactical combinations, where each of the opponent's moves is forced, is too restrictive to discover interesting patterns. Consequently, several authors have tried to generalize these techniques. Puget (1987), e.g., simply made the assumption that all the opponent's moves are optimal (he has reached a goal after all), but presents a technique that is able to include alternative moves when regressing patterns over the program's own moves. However, his technique had to sacrifice sufficiency, i.e., it could no longer be guaranteed that the move associated with the pattern will actually lead to a forced win. *Lazy explanation-based learning* (Tadepalli 1989) does this even more radically by learning over-general, overly optimistic plans, which are generated by simply not checking whether the moves that led to the achievement of a goal were forced. If one of these optimistic plans predicts that its application should achieve a certain goal, but it is not achieved in the current game, the opponent's refutation is generalized to a counter-plan that is indexed with the original, optimistic plan. Thus, the next time this plan is invoked, the program is able to foresee this refutation. Technically, this is quite similar to the case-based planning approach discussed in Section 5.2.

Another problem with such an approach is that in games like chess the ultimate goal (checkmate) is typically too distant to be of practical relevance. Consequently, Flann (1989) describes PLACE, a program that reasons with automatically acquired abstractions in the form of new subgoals and operators that maintain, destroy or achieve such a goal. Complex expressions for the achievement of multiple goals are analyzed by the program and compiled by performing an exhaustive case analysis (Flann 1990, 1992), which is able to generate geometrical constraints that can be used as a recognition pattern for the abstract concept.

People also looked for ways of combining EBL with other learning techniques. Flann and Dietterich (1989) demonstrate methods for augmenting EBL with a similarity-based induction module that is able to inductively learn concepts from multiple explanations of several examples. In a first step, the common subtree of the explanations for each of the examples is computed and generalized with EBL. In a second phase this general concept is inductively specialized by replacing some variables with constants that are common in all examples. Dietterich and Flann (1997) remarked that explanation-based and reinforcement learning are actually quite similar in the way both propagate information from the goal states backward. Consequently, they propose an algorithm that integrates the generalization ability of explanation-based goal regression into a conventional reinforcement learning algorithm. Among others, they demonstrate a successful application of the technique to the KRK chess endgame. Flann (1992) shows similar results on other endings in chess and checkers.

However, because of the above-mentioned problems, most importantly the utility problem and the need for a domain theory that contains more information than a mere representation of the rules of the games, interest in explanation-based learning techniques has faded. Recently, it has been rejuvinated for the game of Go, when Kojima (1995) proposed to use EBL techniques for recognizing forced moves. This technique is described in more detail in (Kojima and Yoshikawa 2001). A related technique is

used in the Go program GOGOL for acquiring rules from tactical goals that are passed to the program in the form of meta-programs (Cazenave 1998).

5.5 Pattern induction

In addition to databases of common openings and huge game collections, which have—so far—been mostly used for the tuning of evaluation functions (Section 4) or the automatic generation of opening books (Section 2), for many games, complete subgames have already been solved, and databases are available in which the game-theoretic value of positions of these subgames can be looked up. In chess, for example, all endgames with up to five pieces and some six-piece endgames have been solved (Thompson 1996). Similar endgame databases have been built for other games like checkers (Lake et al. 1994). Some games are even solved completely, i.e., all possible positions have been evaluated and the game-theoretic value of the starting position has been determined (Allis 1994; Gasser 1995). Many of these databases are readily available, some of them (in the domains of chess, connect-4 and tic-tac-toe) are commonly used as benchmark problems for evaluating machine-learning algorithms.[13] For some of them, not only the raw position information but also derived features, which capture valuable information (similar to terms in evaluation functions), are made available in order to facilitate the formulation of useful concepts.

The earliest work on learning from a chess database is reported by Michalski and Negri (1977), who applied the inductive rule learning algorithm AQ (Michalski 1969) to the KPK database described by Clarke (1977). The positions in the database were coded into 17 attributes describing important relations between the pieces. The goal was to learn a set of rules that are able to classify a KPK position as win or draw. After learning from only 250 training positions, a set of rules was found that achieved 80% predictive accuracy on this task, i.e., it was able to correctly classify 80% of the remaining positions.

Quinlan (1979) describes several experiments for learning rules for the KRKN endgame. In (Quinlan 1983), he used his decision tree learning algorithm ID3 to discover recognition rules for positions of the KRKN endgame that are lost in 2 ply as well as for those that are lost in 3 ply. From less than 10% of the possible KRKN positions, ID3 was able to derive a tree that committed only 2 errors on a test set of 10,000 randomly chosen positions (these errors were later corrected by Verhoef and Wesselius (1987)). Quinlan noted that this achievement was only possible with a careful choice of the attributes that were used to represent the positions. Finding the right set of attributes for the lost-in-2-ply task required three weeks. Adapting this set to the slightly different task of learning lost-in-3-ply positions took almost two months. Thus for the lost-in-4-ply task, which he intended to tackle next, Quinlan experimented with methods for automating the discovery of new useful attributes. However, no results from this endeavor have been published.

A severe problem with this and similar experiments is that, although the learned decision trees can be shown to be correct and faster in classification than exhaustive search algorithms, they are typically also incomprehensible to chess experts. Shapiro

[13] They are available from `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

and Niblett (1982) tried to alleviate this problem by decomposing the learning task into a hierarchy of smaller sub-problems that could be tackled independently. A set of rules is induced for each of the sub-problems which together yield a more understandable result. This process of *structured induction* has been employed to learn correct classification procedures for the KPK and the KPa7KR endgames (Shapiro 1987). An endgame expert helped to structure the search space and to design the relevant attributes. The rules for the KPa7KR endgames were generated without the use of a database as an oracle. The rules were interactively refined by the expert who could specify new training examples and suggest new attributes if the available attributes were not able to discriminate between some of the positions. This rule-debugging process was aided by a self-commenting facility that displayed traces of the classification rules in plain English (Shapiro and Michie 1986). A similar semi-autonomous process for refining the attribute set was used by Weill (1994) to generate decision trees for the KQKQ endgame.

However, the problem of decomposing the search space into easily manageable subproblems is a task that also required extensive collaboration with a human expert. Consequently, there have been several attempts to automate this process. Paterson (1983) tried to automatically structure the KPK endgame using a clustering algorithm. The results were negative, as the found hierarchy had no meaning to human experts. Muggleton (1990) applied the rule learning algorithm DUCE to the KPa7KR task studied by Shapiro (1987). DUCE is is able to autonomously suggest high-level concepts to the user, similar to some of the approaches discussed in Section 4.5.4. It looks for common patterns in the rule base (which initially consists of a set of rules each describing one example board position) and tries to reduce its size by replacing the found patterns with new concepts. In machine learning the autonomous introduction of new concepts during the learning phase is commonly known as *constructive induction* (Matheus 1989). Using constructive induction, DUCE reduces the role of the chess expert to a mere evaluator of the suggested concepts instead of an inventor of new concepts. DUCE structured the KPa7KR task into a hierarchy of 13 concepts defined by a total of 553 rules. Shapiro's solution, however, consisted of 9 concepts with only 225 rules. Nevertheless DUCE's solution was found to be meaningful for a chess expert.

Most of these systems are evaluated with respect to their predictive accuracy, i.e., their ability to generalize from a few examples to new examples. Although these results are quite interesting from a machine-learning point of view, their utility for game-playing is less obvious. In game playing, one is typically not interested in generalizations from a few examples when a complete database is already available. However, it would be a valuable research project to simply compress the available databases into a few (hundred) powerful classification rules that are able to reproduce the classification of the positions stored in the database. Currently, I/O costs are often prohibitive so that during regular searches, an endgame-database lookup is only feasible if the entire database can be kept in main memory. Access to external data (e.g., a database stored on a CD-ROM) is only feasible at the root of the search. Various efficient compression techniques have been proposed for this purpose (Heinz 1999a, 1999b). However, if one only has to store a set of classification rules, perhaps with a few exceptions, the memory requirements could be significantly reduced. Again, the main obstacle to this endeavor is not the lack of suitable machine-learning algorithms—nowadays ML

algorithms are fit for addressing hundreds of thousands of examples and can employ powerful subsampling techniques for larger databases (see (Bain and Srinivasan 1995) for some preliminary experiments on compressing distance-to-mate information in the KRK endgame)—but the lack of availability of appropriate background knowledge for these databases. If the learning algorithms are not given the right features that are useful for formulating the concepts, they cannot be expected to generalize well.

5.6 Learning playing strategies

We define a playing strategy as a simple, interpretable procedure for (successfully) playing a game or subgame. Learning such a strategy is significantly harder than to learn to classify positions as won or lost. Consider the case of a simple chess endgame, like KRK. It is quite trivial to learn a function that recognizes won positions (basically these are all positions where Black cannot capture White's rook and is not stale-mate). However, avoiding drawing variations by keeping one's rook safe is not equivalent to making progress towards mating one's opponent. This problem of having to choose between moves that all look equally good has been termed the *mesa effect* (Minsky 1963).

Bain, e.g, has tried to tackle this problem by learning an optimal player from a perfect database for the KRK endgame (Bain 1994; Bain and Srinivasan 1995). He tried to learn rules that predict the number of plies to a win with optimal play on both sides in the KRK endgame. Such a predictor could be used for a simple playing strategy that simply picks the move that promises the shortest distance to win. He found that it was quite easy to learn rules for discriminating drawn positions from won positions, but that the task got harder the longer the distance to win was. Discriminating positions that are won in 12 plies from positions that are won in 13 plies proved to be considerably harder, at least with the simple geometric concepts that he used as background knowledge for the learner.

On the other hand, human players are able to follow a very simple, albeit suboptimal strategy to win the KRK chess endgame. Every chess novice quickly learns to win this endgame. An automaton that could play this endgame (albeit only from a fixed starting position) was already constructed at the beginning of this century by the Spanish engineer Leonardo Torrès y Quevedo (Vigneron 1914). Thus, it seems to be worth-while to look for simple patterns that allow a player to reliably win the endgame but not necessarily in the optimal way (i.e., with the least number of moves).

PAL (Morales 1991; Morales 1997) uses inductive logic programming techniques to induce complex patterns in first-order logic. It requires a human tutor to provide a few training examples for the concepts to learn, but assists her by generating additional examples with a *perturbation algorithm*. This algorithm takes the current concept description and generates new positions by changing certain aspects like the side to move, the location of some pieces, etc. These examples have to be evaluated by the user before they are used for refining the learned concepts. PAL's architecture is quite similar to Shapiro's structured induction (Section 5.1) but it is able to induce concepts in a more powerful description language. Morales (1994) demonstrated this flexibility by using PAL to learn a playing strategy for the KRK endgame. For this task it was trained by a chess expert that provided examples for a set of useful intermediate concepts. Another

reference system is due to Muggleton (1988). His program uses *sequence induction* to learn strategies that enable it to play a part of the difficult KBBKN endgame, namely the task of freeing a cornered bishop.

It is no coincidence that all of the above examples are from the domain of chess endgames. The reason for this is that on the one hand many perfect chess endgame databases are available, while, on the other hand, these are not well-understood by human players. The most famous example are the attempts of a former Correspondence Chess World Champion and a Canadian Chess Champion to defeat Ken Thompson's perfect KQKR database within 50 moves (Michie 1983; Levy and Newborn 1991) or the attempt of an endgame specialist to defeat a perfect database in the "almost undocumented and very difficult" KBBKN endgame (Roycroft 1988). GM John Nunn's effort to manually extract some of the knowledge that is implicitly contained in these databases resulted in a series of widely acknowledged endgame books (Nunn 1992, 1994b, 1995), but Nunn (1994a) readily admitted that he does not yet understand all aspects of the databases he analyzed. We believe that the discovery of playing strategies for endgame databases is a particularly rewarding topic for further research (Fürnkranz 1997).

6 Opponent Modeling

Opponent modeling is an important, albeit somewhat neglected research area in game playing. The goal is to improve the capabilities of the machine player by allowing it to adapt to its opponent and exploit his weaknesses. Even if a game-theoretical optimal solution to a game is known, a system that has the capability to model its opponent's behavior may obtain a higher reward. Consider, for example, the game of *rock-paper-scissors* aka *RoShamBo*, where either player can expect to win one third of the games (with one third of draws) if both players play their optimal strategies (i.e., randomly select one of their three moves). However, against a player that always plays *rock*, a player that is able to adapt his strategy to always playing *paper* can maximize his reward, while a player that sticks with the "optimal" random strategy will still win only one third of the games (Billings 2000b). In fact, there have been recent tournaments on this game in which the game-theoretically optimal player (the random player) only finished in the middle of the pack because it was unable to exploit the weaker players (Billings 2000a; Egnor 2000).

For the above reason, opponent modeling has recently become a hot topic in *game theory*. Work in this area is beyond the scope of this paper (and the author's expertise); we have to refer the reader to (Fudenberg and Levine 1998) or to Al Roth's on-line bibliography.[14] For some AI contributions to game-theoretic problems see, e.g., (Littman 1994; Monderer and Tennenholtz 1997; Brafman and Tennenholtz 1999; Koller and Pfeffer 1997).

Surprisingly, however, opponent modeling has not yet received much attention in the computer-games community. Billings et al. (1998b) see the reason for that in the fact that "in games like chess, opponent modeling is not critical to achieving high

[14] See `http://www.economics.harvard.edu/~aroth/bib.html#learnbib`

performance." While this may be true for some games, it is certainly not true for all games. "In poker, however," they continue "opponent modeling is essential to success."

Although various other authors have noticed before that minimax search always strives for optimal play against all players while it could be more rewarding to try to maximize the reward against a fallible opponent, Jansen (1990) was probably the first who explicitly tried to exploit this by investigating the connection of search depth and move selection. His basic idea was to accumulate various statistics about the search (like the number of fail-highs or fail-lows in an alpha-beta search) in order to extract some meta-information about how the search progresses. In particular, he suggested to look for *trap* and *swindle* positions, i.e., positions in which one of the opponent's moves looks much better (much worse) than its alternatives, assuming that the opponent's search depth is limited. Trap positions have the potential that the opponent might make a bad move because its refutation is too deep in the search tree, while in a swindle position the player can hope that the opponent overlooks one of his best replies because he does not see the crucial continuation.[15] In subsequent work, Jansen (1992b, 1992a, 1993) has investigated the use of heuristics that occasionally suggest suboptimal moves with swindle potential for the weaker side of KQKR chess endgame. Other researchers proposed techniques to incorporate such opponent models into the system's own search algorithm (Carmel and Markovitch 1993, 1998b; Iida et al. 1994).

To our knowledge, Carmel and Markovitch (1993) were the first to try to automatically *learn* a model of the opponent's move selection progress. First, they proposed a simple algorithm for estimating the depth of the opponent's minimax search. To this end, they employed a comparison training technique (see Section 4.2). For each training position, all moves are evaluated by a limited-depth search. The number of moves that are deemed better than the move actually played is subtracted from the score of the current search depth, while the number of moves that are considered equal or worse is added to the score. This procedure is repeated for all search depths up to a certain maximum. The depth achieving the maximum score is returned. Subsequently, the authors also proposed a way for learning the opponent's (linear) evaluation function by supervised learning: for each possible move in the training positions, the algorithm generates a linear constraint that specifies that the (opponent's) evaluation of the played move has to be at least as good as the alternative under consideration (which the opponent chose not to play). A set of weights satisfying these constraints is then found by linear programming. This procedure is repeated until the program makes no significant progress in predicting the opponent's moves on an independent test set. Again, this is applied to a variety of search depths, where for depths > 1 the dominant positions of the minimax trees of the two moves are used in the linear constraints.[16] Carmel and Markovitch (1993) tested their approach in a checkers player. The opponent model was trained offline on 800 training positions, and the learned models were evaluated by measuring the

[15] Uiterwijk and van den Herik (1994) suggest to also give a bonus for continuations where the proportion of high-quality moves is relatively low. However, it is questionable whether such an approach will produce a high gain. For example, this approach would also award a bonus for piece trades, where moves that do not complete the trade will have much lower evaluation.

[16] The learning of the evaluation function for each individual search depth is quite similar to many of the approaches described in Section 4. The novel aspect here is the learning of several functions, one for each level of search depth, and their use for opponent modeling.

accuracy with which they were able to predict the opponent's moves on an independent test set of another 800 examples. In two out of three runs, the learned model achieved perfect accuracy. Subsequently, the authors integrated their learning algorithm into an opponent model search algorithm, a predecessor of the algorithm described in (Carmel and Markovitch 1998b), where the underlying opponent model was updated after every other game. The learner was soon able to outperform its non-learning opponents.

Later, Carmel and Markovitch (1998a) investigated approaches for learning regular automata that simulate the behavior of their opponents' strategies and applied them to several game-theoretical problems. They were able to show that an agent using their method of inferring a model of the opponent's strategy from its interaction behavior was superior to several non-adaptive and reinforcement-learning agents.

The *Inductive Adversary Modeler (*IAM*)* (Walczak 1991; Walczak and Dankel II 1993) makes the assumption that a human player strives for positions that are cognitively simple, i.e., they can be expressed with a few so-called chunks. Whereas a system like CHUMP (Section 5) bases its own move choices upon a chunk library, IAM tries to model its opponent's chunk library. To that end, it learns a pattern database from a collection of its opponent's games. Whenever it is IAM's turn to move, it looks for nearly completed chunks, i.e. for chunks that the opponent could complete with one move, and predicts that the opponent will play this move. Multiple move predictions are resolved heuristically by preferring bigger and more reliable chunks. In the domain of chess, IAM was able to correctly predict more than 10% of the moves in games of several grandmasters including Botwinnik, Karpov, and Kasparov. As IAM does not always predict a move, the percentage of correctly predicted moves when it actually made a prediction was even higher. The author also discussed these approaches for the games of Go (Walczak and Dankel II 1993) and Hex (Walczak 1992). A variant of the algorithm, which is only concerned with the modeling of the opponent's opening book choices, is discussed in (Walczak 1992) and (Walczak 1996).

As mentioned above, Billings et al. (1998b) have noted that opponent modeling is more crucial to success in poker than it is in other games. There are many different types of poker players, and a good player adapts his strategy to his opponent. For example, trying to call a bluff has a lower chance of winning against a conservative player than against a risky player. Billings et al. (1998b) distinguish between two different opponent modeling techniques: *generic* models capture general playing patterns (like a betting action should increase the probability that a player holds a stronger hand), while *specific* models reflect the individual behavior of the opponents. Both techniques are important for updating the weights upon which LOKI's decisions are based.

At the heart of LOKI's architecture is an evaluation function called the *Triple Generator*, which computes a probability distribution for the next action of a player (*fold*, *check/call*, or *bet/raise*) given his cards and the current state of the game. The other central piece of information is the *Weight Table*, which, for each possible two-card combination, contains the current estimates that each of the opponents holds this particular hand. Generic modeling is performed after each of the opponent's actions by calling the Triple Generator for each possible hand and multiplying the corresponding entry in the weight table with the probability that corresponds to the action the opponent has taken. Thus, LOKI's beliefs about the probabilities of its opponents' hands are held consistent with their actions. Specific modeling is based on statistics that are

kept about the frequency of the opponents' actions in various game contexts. In particular, LOKI computes the median hand value for which an opponent would call a bet and its variance. Hands with estimated values above the sum of mean and variance are not re-weighted (factor $= 1.0$), while hands below the mean minus the variance are practically ignored (factor ≈ 0.0). Values inbetween are interpolated. Thus, LOKI uses information gathered from the individual opponents' past behavior to re-adjust their entries in the Weight Table. More details of the computation of the generic and specific opponent models can be found in (Billings et al. 1998b; Billings et al. 2001).

7 Conclusion

In this paper, we have surveyed research in machine learning for computer game playing. It is unavoidable that such an overview is somewhat biased by the author's knowledge and interests, and our sincere apologies go to all authors whose work had to be ignored due to our space constraints or ignorance. Nevertheless, we hope that we have provided the reader with a good starting point that is helpful for identifying the relevant works to start one's own investigations.

If there is a conclusion to be drawn from this survey, then it should be that research in game playing poses serious and difficult problems which need to be solved with existing or yet-to-be-developed machine learning techniques. Plus, of course, it's fun...

Acknowledgements

I would like to thank Michael Buro, Susan Epstein, Fernand Gobet, Gerry Tesauro, and Paul Utgoff for thoughtful comments that helped to improve this paper. Special thanks to Miroslav Kubat.

Pointers to on-line versions of many of the cited papers can be found in the on-line Bibliography on Machine Learning in Strategic Game Playing at `http://www.ai.univie.ac.at/~juffi/lig/`.

The Austrian Research Institute for Artificial Intelligence is supported by the Austrian Federal Ministry of Education, Science and Culture.

References

AGRAWAL, R., H. MANNILA, R. SRIKANT, H. TOIVONEN, & A. I. VERKAMO (1995). Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. AAAI Press.

AKL, S. G. & M. NEWBORN (1977). The principal continuation and the killer heuristic. In *Proceedings of the 5th Annual ACM Computer Science Conference*, Seattle, WA, pp. 466–473. ACM Press.

ALLIS, V. (1988, October). A knowledge-based approach of Connect-Four — the game is solved: White wins. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

ALLIS, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph. D. thesis, University of Limburg, The Netherlands.

ANGELINE, P. J. & J. B. POLLACK (1994). Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms (GA-93)*, pp. 264–270.

BAIN, M. (1994). *Learning Logical Exceptions in Chess*. Ph. D. thesis, Department of Statistics and Modelling Science, University of Strathclyde, Scotland.

BAIN, M. & A. SRINIVASAN (1995). Inductive logic programming with large-scale unstructured data. In K. Furukawa, D. Michie, and S. H. Muggleton (Eds.), *Machine Intelligence 14*, pp. 233–267. Oxford University Press.

BAXTER, J., A. TRIDGELL, & L. WEAVER (1998a). A chess program that learns by combining TD(lambda) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning (ICML-98)*, Madison, WI, pp. 28–36. Morgan Kaufmann.

BAXTER, J., A. TRIDGELL, & L. WEAVER (1998b). Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal 21*(2), 84–99.

BAXTER, J., A. TRIDGELL, & L. WEAVER (2000, September). Learning to play chess using temporal differences. *Machine Learning 40*(3), 243–263.

BAXTER, J., A. TRIDGELL, & L. WEAVER (2001). Reinforcement learning and chess. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 5, pp. 91–116. Huntington, NY: Nova Science Publishers. To appear.

BEAL, D. F. & M. C. SMITH (1997, September). Learning piece values using temporal difference learning. *International Computer Chess Association Journal 20*(3), 147–151.

BEAL, D. F. & M. C. SMITH (1998). First results from using temporal difference learning in Shogi. In H. J. van den Herik and H. Iida (Eds.), *Proceedings of the First International Conference on Computers and Games (CG-98)*, Volume 1558 of *Lecture Notes in Computer Science*, Tsukuba, Japan, pp. 113. Springer-Verlag.

BERLINER, H. (1979). On the construction of evalution functions for large domains. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI-79)*, pp. 53–55.

BERLINER, H. (1984). Search vs. knowledge: An analysis from the domain of games. In A. Elithorn and R. Banerji (Eds.), *Artificial and Human Intelligence*. New York, NY: Elsevier.

BERLINER, H. & M. S. CAMPBELL (1984). Using chunking to solve chess pawn endgames. *Artificial Intelligence 23*, 97–120.

BERLINER, H., G. GOETSCH, M. S. CAMPBELL, & C. EBELING (1990). Measuring the performance potential of chess programs. *Artificial Intelligence 43*, 7–21.

BHANDARI, I., E. COLET, J. PARKER, Z. PINES, R. PRATAP, & K. RAMANUJAM (1997). Advanced Scout: Data mining and knowledge discovery in NBA data. *Data Mining and Knowledge Discovery 1*, 121–125.

BILLINGS, D. (2000a, March). The first international RoShamBo programming competition. *International Computer Games Association Journal 23*(1), 42–50.

BILLINGS, D. (2000b, March). Thoughts on RoShamBo. *International Computer Games Association Journal 23*(1), 3–8.

BILLINGS, D., D. PAPP, J. SCHAEFFER, & D. SZAFRON (1998b). Opponent modeling in poker. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, pp. 493–498. AAAI Press.

BILLINGS, D., D. PAPP, J. SCHAEFFER, & D. SZAFRON (1998a). Poker as a testbed for machine intelligence research. In R. E. Mercer and E. Neufeld (Eds.), *Advances in Artificial Intelligence (AI-98)*, Vancouver, Canada, pp. 228–238. Springer-Verlag.

BILLINGS, D., L. PEÑA, J. SCHAEFFER, & D. SZAFRON (1999). Using probabilistic knowledge and simulation to play Poker. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pp. 697–703.

BILLINGS, D., L. PEÑA, J. SCHAEFFER, & D. SZAFRON (2001). Learning to play strong poker. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 11, pp. 225–242. Huntington, NY: Nova Science Publishers. To appear.

BISHOP, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford, UK: Clarendon Press.

BOYAN, J. A. (1992). Modular neural networks for learning context-dependent game strategies. Master's thesis, University of Cambridge, Department of Engineering and Computer Laboatory.

BRAFMAN, R. I. & M. TENNENHOLTZ (1999). A near-optimal polynomial time alogorithm for learning in stochastic games. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 734–739.

BRATKO, I. & R. KING (1994). Applications of inductive logic programming. *SIGART Bulletin 5*(1), 43–49.

BRATKO, I. & D. MICHIE (1980). A representation of pattern-knowledge in chess endgames. In M. Clarke (Ed.), *Advances in Computer Chess 2*, pp. 31–54. Edinburgh University Press.

BRATKO, I. & S. H. MUGGLETON (1995, November). Applications of inductive logic programming. *Communications of the ACM 38*(11), 65–70.

BRÜGMANN, B. (1993, March). Monte Carlo Go. Available from `ftp://ftp.cse.cuhk.edu.hk/pub/neuro/GO/mcgo.tex`. Unpublished manuscript.

BURO, M. (1995a). ProbCut: An effective selective extension of the $\alpha - beta$ algorithm. *International Computer Chess Association Journal 18*(2), 71–76.

BURO, M. (1995b). Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research 3*, 373–382.

BURO, M. (1997). The Othello match of the year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal 20*(3), 189–193.

BURO, M. (1998). From simple features to sophisticated evaluation functions. In H. J. van den Herik and H. Iida (Eds.), *Proceedings of the First International Conference on Computers and Games (CG-98)*, Volume 1558 of *Lecture Notes in Computer Science*, Tsukuba, Japan, pp. 126–145. Springer-Verlag.

BURO, M. (1999a). Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In H. J. van den Herik and H. Iida (Eds.), *Games in AI Research*. Maastricht, The Netherlands.

BURO, M. (1999b). Toward opening book learning. *International Computer Chess Association Journal 22*(2), 98–102. Research Note.

BURO, M. (2001). Toward opening book learning. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 4, pp. 81–89. Huntington, NY: Nova Science Publishers. To appear.

CAMPBELL, M. S. (1999, November). Knowledge discovery in Deep Blue. *Communications of the ACM 42*(11), 65–67.

CARMEL, D. & S. MARKOVITCH (1993). Learning models of opponent's strategy in game playing. Number FS-93-02, Menlo Park, CA, pp. 140–147. The AAAI Press.

CARMEL, D. & S. MARKOVITCH (1998a, July). Model-based learning of interaction strategies in multiagent systems. *Journal of Experimental and Theoretical Artificial Intelligence 10*(3), 309–332.

CARMEL, D. & S. MARKOVITCH (1998b, March). Pruning algorithms for multi-model adversary search. *Artificial Intelligence 99*(2), 325–355.

CAZENAVE, T. (1998). Metaprogramming forced moves. In H. Prade (Ed.), *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, Brighton, U.K., pp. 645–649. Wiley.

CHAKRABARTI, S. (2000, January). Data mining for hypertext: A tutorial survey. *SIGKDD explorations 1*(2), 1–11.

CHASE, W. G. & H. A. SIMON (1973). The mind's eye in chess. In W. G. Chase (Ed.), *Visual Information Processing: Proceedings of the 8th Annual Carnegie Psychology Symposium*. New York: Academic Press. Reprinted in (Collins and Smith 1988).

CLARKE, M. R. B. (1977). A quantitative study of king and pawn against king. In M. Clarke (Ed.), *Advances in Computer Chess*, pp. 108–118. Edinburgh University Press.

COLLINS, A. & E. E. SMITH (Eds.) (1988). *Readings in Cognitive Science.* Morgan Kaufmann.

COLLINS, G., L. BIRNBAUM, B. KRULWICH, & M. FREED (1993). The role of self-models in learning to plan. In A. L. Meyrowitz and S. Chipman (Eds.), *Foundations of Knowledge Acquisition: Machine Learning*, pp. 83–116. Boston: Kluwer Academic Publishers.

CRITES, R. & A. BARTO (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning 33*, 235–262.

DAHL, F. A. (2001). Honte, a go-playing program using neural nets. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 10, pp. 205–223. Huntington, NY: Nova Science Publishers. To appear.

DE GROOT, A. D. (1965). *Thought and Choice in Chess*. The Hague, The Netherlands: Mouton.

DE GROOT, A. D. & F. GOBET (1996). *Perception and Memory in Chess: Heuristics of the Professional Eye.* Assen, The Netherlands: Van Gorcum.

DE RAEDT, L. (Ed.) (1995). *Advances in Inductive Logic Programming*, Volume 32 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

DIETTERICH, T. G. (1997, Winter). Machine learning research: Four current directions. *AI Magazine 18*(4), 97–136.

DIETTERICH, T. G. & N. S. FLANN (1997). Explanation-based and reinforcement learning: A unified view. *Machine Learning 28*(2/3), 169–210.

DONNINGER, C. (1996). CHE: A graphical language for expressing chess knowledge. *International Computer Chess Association Journal 19*(4), 234–241.

EGNOR, D. (2000, March). Iocaine Powder. *International Computer Games Association Journal 23*(1), 33–35. Resarch Note.

ENDERTON, H. D. (1991, December). The Golem Go program. Technical Report CMU-CS-92-101, School of Computer Science, Carnegie-Mellon University.

ENGLE, R. W. & L. BUKSTEL (1978). Memory processes among Bridge players of differing expertise. *American Journal of Psychology 91*, 673–689.

EPSTEIN, S. L. (1992). Prior knowledge strengthens learning to control search in weak theory domains. *International Journal of Intelligent Systems 7*, 547–586.

EPSTEIN, S. L. (1994a). For the right reasons: The FORR architecture for learning in a skill domain. *Cognitive Science 18*(3), 479–511.

EPSTEIN, S. L. (1994b). Identifying the right reasons: Learning to filter decision makers. In R. Greiner and D. Subramanian (Eds.), *Proceedings of the AAAI Fall Symposium on Relevance*, pp. 68–71. AAAI Press. Technical Report FS-94-02.

EPSTEIN, S. L. (1994c). Toward an ideal trainer. *Machine Learning 15*, 251–277.

EPSTEIN, S. L. (2001). Learning to play expertly: A tutorial on hoyle. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 8, pp. 153–178. Huntington, NY: Nova Science Publishers. To appear.

EPSTEIN, S. L., J. J. GELFAND, & J. LESNIAK (1996). Pattern-based learning and spatially-oriented concept formation in a multi-agent, decision-making expert. *Computational Intelligence 12*(1), 199–221.

FAWCETT, T. E. & F. PROVOST (1997). Adaptive fraud detection. *Data Mining and Knowledge Discovery 1*(3), 291–316.

FAWCETT, T. E. & P. E. UTGOFF (1992). Automatic feature generation for problem solving systems. In D. Sleeman and P. Edwards (Eds.), *Proceedings of the 9th International Conference on Machine Learning*, pp. 144–153. Morgan Kaufmann.

FAYYAD, U. M., S. G. DJORGOVSKI, & N. WEIR (1996, Summer). From digitized images to online catalogs — data mining a sky survey. *AI Magazine 17*(2), 51–66.

FAYYAD, U. M., G. PIATETSKY-SHAPIRO, & P. SMYTH (1996, Fall). From data mining to knowledge discovery in databases. *AI Magazine 17*(3), 37–54.

FAYYAD, U. M., G. PIATETSKY-SHAPIRO, P. SMYTH, & R. UTHURUSAMY (Eds.) (1995). *Advances in Knowledge Discovery and Data Mining*. Menlo Park: AAAI Press.

FEIGENBAUM, E. A. (1961). The simulation of verbal learning behavior. In *Proceedings of the Western Joint Computer Conference*, pp. 121–132. Reprinted in (Shavlik and Dietterich 1990).

FEIGENBAUM, E. A. & J. FELDMAN (Eds.) (1963). *Computers and Thought*. New York: McGraw-Hill. Republished by AAAI Press/ MIT Press, 1995.

FLANN, N. S. (1989). Learning appropriate abstractions for planning in formation problems. In A. M. Segre (Ed.), *Proceedings of the 6th International Workshop on Machine Learning*, pp. 235–239. Morgan Kaufmann.

FLANN, N. S. (1990). Applying abstraction and simplification to learn in intractable domains. In B. W. Porter and R. Mooney (Eds.), *Proceedings of the 7th International Conference on Machine Learning*, pp. 277–285. Morgan Kaufmann.

FLANN, N. S. (1992). *Correct Abstraction in Counter-Planning: A Knowledge-Compilation Approach*. Ph. D. thesis, Oregon State University.

FLANN, N. S. & T. G. DIETTERICH (1989). A study of explanation-based methods for inductive learning. *Machine Learning 4*, 187–226.

FLINTER, S. & M. T. KEANE (1995). On the automatic generation of case libraries by chunking chess games. In M. Veloso and A. Aamodt (Eds.), *Proceedings of the 1st International Conference on Case Based Reasoning (ICCBR-95)*, pp. 421–430. Springer Verlag.

FOGEL, D. B. (1993). Using evolutionary programming to construct neural networks that are capable of playing tic-tac-toe. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN-93)*, San Francisco, pp. 875–879.

FREY, P. W. (1986). Algorithmic strategies for improving the performance of game playing programs. *Physica D 22*, 355–365. Also published in D. Farmer, A. Lapedes, N. Packard, and B. Wendroff (eds.) *Evolution, Games and Learning: Models for Adaptation in Machine and Nature*, North-Holland, 1986.

FUDENBERG, D. & D. K. LEVINE (1998). *The Theory of Learning in Games*. Series on Economic Learning and Social Evolution. Cambridge, MA: MIT Press.

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

FÜRNKRANZ, J. (1997). Knowledge discovery in chess databases: A research proposal. Technical Report OEFAI-TR-97-33, Austrian Research Institute for Artificial Intelligence.

FÜRNKRANZ, J. & M. KUBAT (Eds.) (1999). *Workshop Notes: Machine Learning in Game Playing*, Bled, Slovenia. 16th International Conference on Machine Learning (ICML-99).

FÜRNKRANZ, J. & M. KUBAT (Eds.) (2001). *Machines that Learn to Play Games*. Huntington, NY: Nova Science Publishers. To appear.

FÜRNKRANZ, J., B. PFAHRINGER, H. KAINDL, & S. KRAMER (2000). Learning to use operational advice. In W. Horn (Ed.), *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00)*, Berlin. In press.

GASSER, R. (1995). *Efficiently Harnessing Computational Resources for Exhaustive Search*. Ph. D. thesis, ETH Zürich, Switzerland.

GEORGE, M. & J. SCHAEFFER (1990). Chunking for experience. *International Computer Chess Association Journal 13*(3), 123–132.

GHERRITY, M. (1993). *A Game-Learning Machine*. Ph. D. thesis, University of California, San Diego, CA.

GINSBERG, M. L. (1998, Winter). Computers, games and the real world. *Scientific American Presents 9*(4). Special Issue on *Exploring Intelligence*.

GINSBERG, M. L. (1999). GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, pp. 584–589.

GOBET, F. (1993). A computer model of chess memory. In *Proceedings of the 15th Annual Meeting of the Cognitive Science Society*, pp. 463–468.

GOBET, F. & P. J. JANSEN (1994). Towards a chess program based on a model of human memory. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 35–60. University of Limburg.

GOBET, F. & H. A. SIMON (2001). Human learning in game playing. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 3, pp. 61–80. Huntington, NY: Nova Science Publishers. To appear.

GOLDBERG, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

GOULD, J. & R. A. LEVINSON (1994). Experience-based adaptive search. In R. S. Michalski and G. Tecuci (Eds.), *Machine Learning: A Multi-Strategy Approach*, pp. 579–604. Morgan Kaufmann.

GREENBLATT, R. D., D. E. EASTLAKE III, & S. D. CROCKER (1967). The Greenblatt chess program. In *Proceedings of the Fall Joint Computer Conference*, pp. 801–810. American Federation of Information Processing Societies. Reprinted in (Levy 1988).

GREER, K. R. C., P. C. OJHA, & D. A. BELL (1999). A pattern-oriented approach to move ordering: the chessmaps heuristic. *International Computer Chess Association Journal 22*, 13–21.

HAMMOND, K. J. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Boston, MA: Academic Press.

HÄTÖNEN, K., M. KLEMETTINEN, H. MANNILA, P. RONKAINEN, & H. TOIVONEN (1996). Knowledge discovery from telecommunication network alarm databases. In *Proceedings of the 12th International Conference on Data Engineering (ICDE-96)*.

HEINZ, E. A. (1999a, March). Endgame databases and efficient index schemes for chess. *International Computer Chess Association Journal 22*(1), 22–32. Research Note.

HEINZ, E. A. (1999b, June). Knowledgeable encoding and querying of endgame databases. *International Computer Chess Association Journal 22*(2), 81–97.

HOLDING, D. H. (1985). *The Psychology of Chess Skill*. Hillsdale, NJ: Lawrence Erlbaum Associates.

HSU, F.-H., T. S. ANANTHARAMAN, M. S. CAMPBELL, & A. NOWATZYK (1990a). Deep Thought. In T. A. Marsland and J. Schaeffer (Eds.), *Computers, Chess, and Cognition*, Chapter 5, pp. 55–78. Springer-Verlag.

HSU, F.-H., T. S. ANANTHARAMAN, M. S. CAMPBELL, & A. NOWATZYK (1990b, October). A grandmaster chess machine. *Scientific American 263*(4), 44–50.

HSU, J.-M. (1985). A strategic game program that learns from mistakes. Master's thesis, Northwestern University, Evanston, IL.

HYATT, R. M. (1999, March). Book learning — a methodology to tune an opening book automatically. *International Computer Chess Association Journal 22*(1), 3–12.

IIDA, H., J. W. H. M. UITERWIJK, H. J. VAN DEN HERIK, & I. S. HERSCHBERG (1994). Thoughts on the application of opponent-model search. In H. J. van den Herik and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 61–78. Maastricht, Holland: Rijksuniversiteit Limburg.

INUZUKA, N., H. FUJIMOTO, T. NAKANO, & H. ITOH (1999). Pruning nodes in the alpha-beta method using inductive logic programming. See Fürnkranz and Kubat (1999).

ISABELLE, J.-F. (1993). Auto-apprentissage à l'aide de résaux de neurones, de fonctions heuristiques utilisées dans les jeux stratégiques. Master's thesis, University of Montreal. In French.

JANSEN, P. J. (1990). Problematic positions and speculative play. In T. A. Marsland and J. Schaeffer (Eds.), *Computers, Chess, and Cognition*, pp. 169–181. New York: Springer-Verlag.

JANSEN, P. J. (1992a). KQKR: Assessing the utility of heuristics. *International Computer Chess Association Journal 15*(4), 179–191.

JANSEN, P. J. (1992b). KQKR: Awareness of a fallible opponent. *International Computer Chess Association Journal 15*(3), 111–131.

JANSEN, P. J. (1993, March). KQKR: Speculatively thwarting a human opponent. *International Computer Chess Association Journal 16*(1), 3–17.

JUNGHANNS, A. & J. SCHAEFFER (1997). Search versus knowledge in game-playing programs revisited. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, pp. 692–697.

KAINDL, H. (1982). Positional long-range planning in computer chess. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 145–167. Pergamon Press.

KASPAROV, G. (1998, March). My 1997 experience with Deep Blue. *International Computer Chess Association Journal 21*(1), 45–51.

KERNER, Y. (1995). Learning strategies for explanation patterns: Basic game patterns with application to chess. In M. Veloso and A. Aamodt (Eds.), *Proceedings of the 1st International Conference on Case-Based Reasoning (ICCBR-95)*, Volume 1010 of *Lecture Notes in Artificial Intelligence*, Berlin, pp. 491–500. Springer-Verlag.

KOJIMA, T. (1995). A model of acquisition and refinement of deductive rules in the game of Go. Master's thesis, University of Tokyo. In Japanese.

KOJIMA, T., K. UEDA, & S. NAGANO (1997). An evolutionary algorithm extended by ecological analogy and its application to the game of Go. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya, Japan, pp. 684–689.

KOJIMA, T. & A. YOSHIKAWA (2001). Acquisition of go knowledge from game records. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 9, pp. 179–204. Huntington, NY: Nova Science Publishers. To appear.

KOLLER, D. & A. J. PFEFFER (1997). Representations and solutions for game-theoretic problems. *Artificial Intelligence 94*(1-2), 167–215.

KRULWICH, B. (1993). *Flexible Learning in a Multi-Component Planning System*. Ph. D. thesis, The Institute for the Learning Sciences, Northwestern University, Evanston, IL. Technical Report #46.

KRULWICH, B., L. BIRNBAUM, & G. COLLINS (1995). Determining what to learn through component-task modeling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 439–445.

KUVAYEV, L. (1997). Learning to play Hearts. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, pp. 837. AAAI Press. Extended Abstract.

LAIRD, J. E., A. NEWELL, & P. S. ROSENBLOOM (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence 33*, 1–64.

LAKE, R., J. SCHAEFFER, & P. LU (1994). Solving large retrograde analysis problems using a network of workstations. In H. J. van den Herik and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 135–162. Maastricht, Holland: Rijksuniversiteit Limburg.

LAVRAČ, N. (1999). Machine learning for data mining in medicine. In W. Horn, Y. Shahar, G. Lindberg, S. Andreassen, and J. Wyatt (Eds.), *Artificial Intelligence in Medicine*, pp. 47–62. Berlin: Springer-Verlag.

LAVRAČ, N. & S. DŽEROSKI (1993). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

LEE, K.-F. & S. MAHAJAN (1988). A pattern classification apporach to evaluation function learning. *Artificial Intelligence 36*, 1–25.

LEE, K.-F. & S. MAHAJAN (1990). The development of a world class Othello program. *Artificial Intelligence 43*, 21–36.

LEOUSKI, A. & P. E. UTGOFF (1996, March). What a neural network can learn about Othello. Technical Report UM-CS-1996-010, Computer Science Department, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA.

LEVINSON, R. A., B. BEACH, R. SNYDER, T. DAYAN, & K. SOHN (1992). Adaptive-predictive game-playing programs. *Journal of Experimental and Theoretical Artificial Intelligence 4*(4).

LEVINSON, R. A. & R. SNYDER (1991). Adaptive pattern-oriented chess. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the 8th International Workshop on Machine Learning (ML-91)*, pp. 85–89. Morgan Kaufmann.

LEVINSON, R. A. & R. SNYDER (1993, September). DISTANCE: Toward the unification of chess knowledge. *International Computer Chess Association Journal 16*(3), 123–136.

LEVY, D. N. (1986). *Chess and Computers*. London: Batsford.

LEVY, D. N. (Ed.) (1988). *Computer Chess Compendium*. London: Batsford.

LEVY, D. N. & D. F. BEAL (Eds.) (1989). *Heuristic Programming in Artificial Intelligence — The First Computer Olympiad*. Chichester, England: Ellis Horwood.

LEVY, D. N. & M. NEWBORN (1991). *How Computers Play Chess*. Computer Science Press.

LITTMAN, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, New Brunswick, NJ, pp. 157–163. Morgan Kaufmann.

LYNCH, M. (1997, June). Neurodraughts: An application of temporal difference learning to Draughts. Final Year Project Report, Department of Computer Science and Information Systems, University of Limerick, Ireland.

MARSLAND, T. A. (1985). Evaluation-function factors. *International Computer Chess Association Journal 8*(2), 47–57.

MATHEUS, C. J. (1989). A constructive induction framework. In *Proceedings of the 6th International Workshop on Machine Learning*, pp. 474–475.

MATSUBARA, H. (Ed.) (1998). *Advances in Computer Shogi 2*. Kyouritsu Shuppan corp. ISBN 4-320-02892-9. In Japanese.

MCCORDUCK, P. (1979). *Machines Who Think — A Personal Inquiry into the History and Prospects of Artificial Intelligence*. San Francisco, CA: W. H. Freeman and Company.

MEULEN, M. V. D. (1989). Weight assessment in evaluation functions. In D. F. Beal (Ed.), *Advances in Computer Chess 5*, pp. 81–89. Amsterdam: Elsevier.

MICHALSKI, R. S. (1969). On the quasi-minimal solution of the covering problem. In *Proceedings of the 5th International Symposium on Information Processing (FCIP-69)*, Volume A3 (Switching Circuits), Bled, Yugoslavia, pp. 125–128.

MICHALSKI, R. S., I. BRATKO, & M. KUBAT (Eds.) (1998). *Machine Learning and Data Mining: Methods and Applications*. John Wiley & Sons.

MICHALSKI, R. S. & P. NEGRI (1977). An experiment on inductive learning in chess end games. In E. W. Elcock and D. Michie (Eds.), *Machine Intelligence 8*, pp. 175–192. Chichester, U.K.: Ellis Horwood.

MICHIE, D. (1961). Trial and error. In S. A. Barnett and A. McLaren (Eds.), *Science Survey, Part 2*, pp. 129–145. Harmondsworth, U.K.: Penguin. Reprinted in (Michie 1986).

MICHIE, D. (1963). Experiments on the mechanization of game-learning – Part I. Characterization of the model and its parameters. *The Computer Journal 6*, 232–236.

MICHIE, D. (1982). Experiments on the mechanization of game-learning 2 – rule-based learning and the human window. *The Computer Journal 25*(1), 105–113.

MICHIE, D. (1983). Game-playing programs and the conceptual interface. In M. Bramer (Ed.), *Computer Game-Playing: Theory and Practice*, Chapter 1, pp. 11–25. Chichester, England: Ellis Horwood.

MICHIE, D. (1986). *On Machine Intelligence* (2nd Edition ed.). Chichester, UK: Ellis Horwood Limited.

MICHIE, D. & I. BRATKO (1991, March). Comments to 'chunking for experience'. *International Computer Chess Association Journal 18*(1), 18.

MINSKY, M. (1963). Steps toward artificial intelligence. In E. A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill.

MINSKY, M. & S. A. PAPERT (1969). *Perceptrons : Introduction to Computational Geometry*. MIT Press. Expanded Edition 1990.

MINTON, S. (1984). Constraint based generalization: Learning game playing plans from single examples. In *Proceedings of the 2nd National Conference on Artificial Intelligence (AAAI-84)*, Austin, TX, pp. 251–254.

MINTON, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence 42*, 363–392.

MITCHELL, D. H. (1984). Using features to evaluate positions in experts' and novices' Othello games. Master's thesis, Northwestern University, Evanston, IL.

MITCHELL, T. M. (1997a, Fall). Does machine learning really work? *AI Magazine 18*(3), 11–20.

MITCHELL, T. M. (1997b). *Machine Learning*. McGraw Hill.

MITCHELL, T. M., R. M. KELLER, & S. KEDAR-CABELLI (1986). Explanation-based generalization: A unifying view. *Machine Learning 1*(1), 47–80.

MITCHELL, T. M., P. E. UTGOFF, & R. BANERJI (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. Michalski, J. Carbonell, and T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Chapter 6. San Mateo, CA: Morgan Kaufmann.

MONDERER, D. & M. TENNENHOLTZ (1997). Dynamic non-bayesian decision making. *Journal of Artificial Intelligence Research 7*, 231–248.

MORALES, E. (1991). Learning features by experimentation in chess. In Y. Kodratoff (Ed.), *Proceedings of the 5th European Working Session on Learning (EWSL-91)*, pp. 494–511. Springer Verlag.

MORALES, E. (1994). Learning patterns for playing strategies. *International Computer Chess Association Journal 17*(1), 15–26.

MORALES, E. (1997). PAL: A pattern-based first-order inductive system. *Machine Learning 26*(2-3), 227–252. Special Issue on Inductive Logic Programming.

MORIARTY, D. & R. MIIKKULAINEN (1994). Evolving neural networks to focus minimax search. In *Proceedings of 12th National Conference on Artificial Intelligence (AAAI-94)*, pp. 1371–1377.

MOSTOW, D. J. (1981). *Mechanical Transformation of Task Heuristics into Operational Procedures*. Ph. D. thesis, Carnegie Mellon University, Department of Computer Science.

MOSTOW, D. J. (1983). Machine transformation of advice into a heuristic search procedure. In *Machine Learning: An Artificial Intelligence Approach*, pp. 367–403. Morgan Kaufmann.

MUGGLETON, S. H. (1988). Inductive acquisition of chess strategies. In J. E. Hayes, D. Michie, and J. Richards (Eds.), *Machine Intelligence 11*, Chapter 17, pp. 375–387. Clarendon Press.

MUGGLETON, S. H. (1990). *Inductive Acquisition of Expert Knowledge*. Turing Institute Press. Addison-Wesley.

MUGGLETON, S. H. (Ed.) (1992). *Inductive Logic Programming*. London: Academic Press Ltd.

MUGGLETON, S. H. & L. DE RAEDT (1994). Inductive Logic Programming: Theory and methods. *Journal of Logic Programming 19,20*, 629–679.

MÜLLER, M. (1999). Computer Go: A research agenda. *International Computer Chess Association Journal 22*(2), 104–112.

NAKANO, T., N. INUZUKA, H. SEKI, & H. ITOH (1998). Inducing Shogi heuristics using inductive logic programming. In D. Page (Ed.), *Proceedings of the 8th International Conference on Inductive Logic Programming (ILP-98)*, Madison, WI, pp. 155–164. Springer.

NEWBORN, M. (1996). *Kasparov versus Deep Blue: Computer Chess Comes of Age*. New York: Springer-Verlag.

NITSCHE, T. (1982). A learning chess program. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 113–120. Pergamon Press.

NUNN, J. (1992). *Secrets of Rook Endings*. Batsford.

NUNN, J. (1994a). Extracting information from endgame databases. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 19–34. Maastricht, The Netherlands: University of Limburg.

NUNN, J. (1994b). *Secrets of Pawnless Endings*. Batsford.

NUNN, J. (1995). *Secrets of Minor-Piece Endings*. Batsford.

OPDAHL, A. L. & B. TESSEM (1994). Long-term planning in computer chess. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*. University of Limburg.

PATERSON, A. (1983). An attempt to use CLUSTER to synthesise humanly intelligible subproblems for the KPK chess endgame. Technical Report UIUCDCS-R-83-1156, University of Illinois, Urbana, IL.

PITRAT, J. (1976a). A program to learn to play chess. In Chen (Ed.), *Pattern Recognition and Artificial Intelligence*, pp. 399–419. New York: Academic Press.

PITRAT, J. (1976b). Realization of a program learning to find combinations at chess. In J. C. Simon (Ed.), *Computer Oriented Learning Processes*, Volume 14 of *NATO Advanced Study Institute Series, Series E: Appied Science*. Leyden: Noordhoff.

PITRAT, J. (1977). A chess combination program which uses plans. *Artificial Intelligence 8*, 275–321.

PLAAT, A. (1996). *Research Re: Search and Re-Search*. Ph. D. thesis, Erasmus University, Rotterdam, The Netherlands.

POLLACK, J. B. & A. D. BLAIR (1998). Co-evolution in the successful learning of backgammon strategy. *Machine Learning 32*(1), 225–240.

PUGET, J.-F. (1987). Goal regression with opponent. In *Progress in Machine Learning*, pp. 121–137. Sigma Press.

QUINLAN, J. R. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Ed.), *Expert Systems in the Micro Electronic Age*, pp. 168–201. Edinburgh University Press.

QUINLAN, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, pp. 463–482. Palo Alto: Tioga.

RATTERMANN, M. J. & S. L. EPSTEIN (1995). Skilled like a person: A comparison of human and computer game playing. In *Proceedings of the 17th Annual Conference of the Cognitive Science Society*, Pittsburgh, PA, pp. 709–714. Lawrence Erlbaum Associates.

REITMAN, J. S. (1976). Skilled perception in Go: Deducing memory structures from inter-response times. *Cognitive Psychology 3*, 336–356.

RIESBECK, C. K. & R. C. SCHANK (1989). *Inside Case-Based Reasoning*. Hillsdale, NJ: Lawrence Erlbaum Associates.

ROBERTIE, B. (1992). Carbon versus silicon: Matching wits with TD-Gammon. *Inside Backgammon 2*(2), 14–22.

ROBERTIE, B. (1993). *Learning from the Machine: Robertie vs. TD-Gammon*. Arlington, MA: Gammon Press.

ROSENBLATT, F. (1958). The perceptron: a probabalistic model for information storage and organization in the brain. *Psychological Review 65*, 386–408.

ROYCROFT, A. J. (1988). Expert against oracle. In J. E. Hayes, D. Michie, and J. Richards (Eds.), *Machine Intelligence 11*, pp. 347–373. Oxford, UK: Oxford University Press.

RUMELHART, D. E. & J. L. MCCLELLAND (Eds.) (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations. Cambridge, MA: MIT Press.

SAMUEL, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development 3*(3), 211–229. Reprinted in (Feigenbaum and Feldman 1963) (with an additional appendix) and in (Shavlik and Dietterich 1990).

SAMUEL, A. L. (1960). Programming computers to play games. In *Advances in Computers 1*, pp. 165–192. Academic Press.

SAMUEL, A. L. (1967). Some studies in machine learning using the game of checkers. ii - recent progress. *IBM Journal of Research and Development 11*(6), 601–617.

SCHAEFFER, J. (1983). The history heuristic. *International Computer Chess Association Journal 6*(3), 16–19.

SCHAEFFER, J. (1986). *Experiments in Knowledge and Search*. Ph. D. thesis, University of Waterloo, Canada.

SCHAEFFER, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Pattern Analysis and Machine Intelligence 11*(11), 1203–1212.

SCHAEFFER, J. (1997). *One Jump Ahead — Challenging Human Supremacy in Checkers*. New York: Springer-Verlag.

SCHAEFFER, J. (2000). The games computers (and people) play. In M. V. Zelkowitz (Ed.), *Advances in Computers*, Volume 50, pp. 189–266. Academic Press.

SCHAEFFER, J., J. CULBERSON, N. TRELOAR, B. KNIGHT, P. LU, & D. SZAFRON (1992). A world championship caliber checkers program. *Artificial Intelligence 53*(2-3), 273–289.

SCHAEFFER, J., R. LAKE, P. LU, & M. BRYANT (1996). Chinook: The world man-machine checkers champion. *AI Magazine 17*(1), 21–29.

SCHAEFFER, J. & A. PLAAT (1997, June). Kasparov versus Deep Blue: The rematch. *International Computer Chess Association Journal 20*(2), 95–101.

SCHANK, R. C. (1986). *Explanation Patterns: Understanding Mechanically and Creatively*. Hillsdale, NJ: Lawrence Erlbaum.

SCHERZER, T., L. SCHERZER, & D. TJADEN (1990). Learning in Bebe. In T. A. Marsland and J. Schaeffer (Eds.), *Computers, Chess, and Cognition*, Chapter 12, pp. 197–216. Springer Verlag.

SCHMIDT, M. (1994). Temporal-difference learning and chess. Technical report, Computer Science Department, University of Aarhus, Aarhus, Denmark.

SCHRAUDOLPH, N. N., P. DAYAN, & T. J. SEJNOWSKI (1994). Temporal difference learning of position evaluation in the game of go. In J. D. Cowan, G. Tesauro, and J. Alspector (Eds.), *Advances in Neural Information Processing 6*, pp. 817–824. San Francisco: Morgan Kaufmann.

SHAPIRO, A. D. (1987). *Structured Induction in Expert Systems.* Turing Institute Press. Addison-Wesley.

SHAPIRO, A. D. & D. MICHIE (1986). A self commenting facility for inductively synthesized endgame expertise. In D. F. Beal (Ed.), *Advances in Computer Chess 4*, pp. 147–165. Oxford, U.K.: Pergamon Press.

SHAPIRO, A. D. & T. NIBLETT (1982). Automatic induction of classification rules for a chess endgame. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 73–92. Oxford, U.K.: Pergamon Press.

SHAVLIK, J. W. & T. G. DIETTERICH (Eds.) (1990). *Readings in Machine Learning*. Morgan Kaufmann.

SHEPPARD, B. (1999, November/December). Mastering Scrabble. *IEEE Intelligent Systems 14*(6), 15–16. Research Note.

SIMON, H. A. & M. BARENFELD (1969). Information-processing analysis of perceptual processes in problem solving. *Psychological Review 76*(5), 473–483.

SIMON, H. A. & K. GILMARTIN (1973). A simulation of memory for chess positions. *Cognitive Psychology 5*, 29–46.

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

SLATE, D. J. & L. R. ATKIN (1983). Chess 4.5 — the northwestern university chess program. In *Chess Skill in Man and Machine* (2 ed.)., Chapter 4, pp. 82–118. Springer-Verlag.

SMITH, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI-83)*, Los Altos, CA, pp. 421–425. Morgan Kaufmann.

SOMMERLUND, P. (1996, May). Artificial neural nets applied to strategic games.

STOUTAMIRE, D. (1991). Machine learning, game play, and Go. Technical Report TR-91-128, Center for Automation and Intelligent Systems Research, Case Western Reserve University.

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

SUTTON, R. S. & A. BARTO (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

TADEPALLI, P. V. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 694–700. Morgan Kaufmann.

TESAURO, G. (1989a). Connectionist learning of expert preferences by comparison training. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems 1 (NIPS-88)*, pp. 99–106. Morgan Kaufmann.

TESAURO, G. (1989b). Neurogammon: a neural-network backgammon learning program. In D. N. L. Levy and D. F. Beal (Eds.), *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pp. 78–80. Ellis Horwood.

TESAURO, G. (1990). Neurogammon: A neural network backgammon program. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-90)*, Volume III, San Diego, CA, pp. 33–39. IEEE.

TESAURO, G. (1992a). Practical issues in temporal difference learning. *Machine Learning 8*, 257–278.

TESAURO, G. (1992b). Temporal difference learning of backgammon strategy. *Proceedings of the 9th International Conference on Machine Learning*, 451–457.

TESAURO, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6*(2), 215–219.

TESAURO, G. (1995, March). Temporal difference learning and TD-Gammon. *Communications of the ACM 38*(3), 58–68.

TESAURO, G. (1998). Comments on 'Co-evolution in the successful learning of backgammon strategy'. *Machine Learning 32*(3), 241–243.

TESAURO, G. (2001). Comparison training of chess evaluation functions. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 6, pp. 117–130. Huntington, NY: Nova Science Publishers. To appear.

TESAURO, G. & T. J. SEJNOWSKI (1989). A parallel network that learns to play backgammon. *Artificial Intelligence 39*, 357–390.

THOMPSON, K. (1996). 6-piece endgames. *International Computer Chess Association Journal 19*(4), 215–226.

THRUN, S. (1995). Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*, pp. 1069–1076. Cambridge, MA: The MIT Press.

TIGGELEN, A. V. (1991). Neural networks as a guide to optimization. The chess middle game explored. *International Computer Chess Association Journal 14*(3), 115–118.

TUNSTALL-PEDOE, W. (1991). Genetic algorithms optimizing evaluation functions. *International Computer Chess Association Journal 14*(3), 119–128.

UITERWIJK, J. W. H. M. & H. J. VAN DEN HERIK (1994). Speculative play in computer chess. In H. J. van den Herik and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 79–90. Maastricht, Holland: Rijksuniversiteit Limburg.

UTGOFF, P. E. (2001). Feature construction for game playing. In J. Fürnkranz and M. Kubat (Eds.), *Machines that Learn to Play Games*, Chapter 7, pp. 131–152. Huntington, NY: Nova Science Publishers. To appear.

UTGOFF, P. E. & J. CLOUSE (1991). Two kinds of training information for evaluation function learning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, Anaheim, CA, pp. 596–600. AAAI Press.

UTGOFF, P. E. & P. S. HEITMAN (1988). Learning and generalizing move selection preferences. In H. Berliner (Ed.), *Proceedings of the AAAI Spring Symposium on Computer Game Playing*, Stanford University, pp. 36–40.

UTGOFF, P. E. & D. PRECUP (1998). Constructive function approximation. In H. Liu and H. Motoda (Eds.), *Feature Extraction, Construction and Selection: A Data Mining Perspective*, Volume 453 of *The Kluwer International Series in Engineering and Computer Science*, Chapter 14. Kluwer Academic Publishers.

VAN DEN HERIK, H. J. (1988). *Informatica en het Menselijk Blikveld*. Maastricht, The Netherlands: University of Limburg. In Dutch.

VERHOEF, T. F. & J. H. WESSELIUS (1987). Two-ply KRKN: Safely overtaking Quinlan. *International Computer Chess Association Journal 10*(4), 181–190.

VIGNERON, H. (1914). Robots. *La Nature*, 56–61. In French. Reprinted in English in (Levy 1986) and (Levy 1988).

WALCZAK, S. (1991). Predicting actions from induction on past performance. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the 8th International Workshop on Machine Learning (ML-91)*, pp. 275–279. Morgan Kaufmann.

WALCZAK, S. (1992). Pattern-based tactical planning. *International Journal of Pattern Recognition and Artificial Intelligence 6*(5), 955–988.

WALCZAK, S. (1996). Improving opening book performance through modeling of chess opponents. In *Proceedings of the 24th ACM Annual Computer Science Conference*, pp. 53–57.

WALCZAK, S. & D. D. DANKEL II (1993). Acquiring tactical and strategic knowledge with a generalized method for chunking of game pieces. *International*

Journal of Intelligent Systems 8(2), 249–270. Reprinted in K. Ford and J. Bradshaw (Eds.) *Knowledge Acquisition As Modeling*, New York: Wiley.

WATERMAN, D. A. (1970). A generalization learning technique for automating the learning of heuristics. *Artificial Intelligence 1*, 121–170.

WATKINS, C. J. & P. DAYAN (1992). Q-learning. *Machine Learning 8*, 279–292.

WEILL, J.-C. (1994). How hard is the correct coding of an easy endgame. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 163–176. University of Limburg.

WIDMER, G. (1996). Learning expressive performance: The structure-level approach. *Journal of New Music Research 25*(2), 179–205.

WILKINS, D. E. (1980). Using patterns and plans in chess. *Artificial Intelligence 14*(3), 165–203.

WILKINS, D. E. (1982). Using knowledge to control tree search searching. *Artificial Intelligence 18*(1), 1–51.

WINKLER, F.-G. & J. FÜRNKRANZ (1998, March). A hypothesis on the divergence of AI research. *International Computer Chess Association Journal 21*(1), 3–13.

WITTEN, I. H. & E. FRANK (2000). *Data Mining — Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers.

WOLFF, A. S., D. H. MITCHELL, & P. W. FREY (1984). Perceptual skill in the game of Othello. *Journal of Psychology 118*, 7–16.

ZOBRIST, A. L. & F. R. CARLSON (1973, June). An advice-taking chess computer. *Scientific American 228*(6), 92–105.

This paper provides a survey of previously published work on machine learning in game playing. The material is organized around a variety of problems that typically arise in game playing and that can be solved with machine learning methods. This approach, we believe, allows both, researchers in game playing to find appropriate learning techniques for helping to solve their problems as well as machine learning researchers to identify rewarding topics for further research in game-playing domains. The chapter covers learning techniques that range from neural networks to decision tree learning in games Machine learning and developing a system, a game, or a machine learning platform is less like programming, and a lot more like testing or teaching a child. And to put it simply, machine learning in a machine learning platform is software that learns and produces-- so produces just like classic software-- but this is a software system that learns and produces without being explicitly programmed. And if you were classically trained as a software engineer or a code developer, learning machine language is a little different. And sometimes a lot different. Because you're not explicitly coding.Â So I went through two case studies of machine learning being used very differently in games. I've got five I know of in detail, and there are a lot more. I mentioned the AlphaGo project.