Abstract

Game playing is a major application area for research in artificial intelligence in general (Schaeffer and van den Herik 2002) and for machine learning in particular (Fürnkranz and Kubat 2001). Traditionally, the field is concerned with learning in strategy games such as tic-tac-toe (Michie 1963), checkers (Samuel’s checkers player), backgammon (TD-Gammon), chess (Baxter et al. 2000; Björnsson and Marsland 2003; Donninger and Lorenz 2006; Sadikov and Bratko 2006), Go (Silver et al. 2016), Othello (Buro 2002), poker (Billings et al. 2002), or bridge (Amit and Markovitch 2006). However, recently computer and video games have received increased attention (Laird and van Lent 2001; Spronck et al. 2006; Ponsen et al. 2006).

Motivation and Background

Since the early days of the field, game-playing applications have been popular test beds for machine learning. This has several reasons:

- **Games allow to focus on intelligent reasoning.** Other components of intelligent agents, such as perception or physical actions, can be ignored.
- **Games are easily accessible.** A typical game-playing environment can be implemented within a few days, often hours. Exceptions are real-time computer games, for which only a few open-source test beds exist.
- **Games are very popular.** It is not very hard to describe the agent’s task to the general public, and they can easily appreciate the achieved level of intelligence.

There are various types of problems that keep reoccurring in game-playing applications, for which solutions with machine learning methods are desirable, including opening book learning, learning of evaluation functions, player modeling, and others, which will be dealt with in the following.
literature. A more detailed survey can be found in Fürnkranz (2001).

Learning of Evaluation Functions
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 features of the current board position (e.g., the number of pieces of each 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. Most frequently, these parameters are combined linearly, so that the learning task is to adjust the weights of a weighted sum. The main problem is that there are typically no direct target values that could be used as training signals. Exceptions are games or endgames that have been solved completely, which are treated further below. However, in general, algorithms use preference learning (where pairs of moves or positions are labeled according to which one is preferred by an expert player) or reinforcement learning (where moves or positions are trained based on information about the eventual outcome of the game) for tuning the evaluation functions.

The key problem with reinforcement learning approaches is the credit assignment problem, i.e., even though a game has been won (lost), there might be bad (good) moves in the game. Reinforcement learning takes a radical stance at this problem, giving all positions the same reinforcement signal, hoping that erroneous signals will be evened out over time. An early classic in this area is MENACE, a tic-tac-toe player that simulates reinforcement learning with delayed rewards (Michie 1963) using a stack of matchboxes, one for each position, each containing a number of beads in different colors, which represent the different legal moves in the position. Moves are selected by randomly drawing a bead out of the box that represents the current position. After a game is won, all played moves are reinforced by adding beads of the corresponding colors to these boxes; in the case of a lost game, corresponding beads are removed, thereby decreasing the probability that the same move will be played again.

The premier example of a system that has tuned its evaluation function to expert strength by playing millions of games against itself is the backgammon program TD-Gammon. Its key innovation was the use of a neural network instead of a position table, so that the reinforcement signal can be generalized to new unseen positions. Many authors have tried to copy TD-GAMMON’s learning methodology to other games (Ghory 2004). 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 facts that the dice rolls guarantee sufficient variability, which allows to use training by self-play without the need for an explicit exploration/exploitation trade-off, and that it only requires a very limited amount of search, which allows to ignore the dependencies of search algorithm and search heuristic. These points have, for example, been addressed with limited success in the game of chess, where the program KNIGHTCAP (Baxter et al. 2000) integrates temporal difference learning into a game-tree search by using the final positions of the principal variation for updates and by using play on a game server for exploration.

Many aspects of evaluation function learning are still discussed in the current literature, including whether there are alternatives to reinforcement learning (e.g., evolutionary algorithms), which training strategies should be used (e.g., self-play vs. play against a teacher), etc. One of the key problems that has already been mentioned in Samuel’s Checkers Player, namely, the automated construction of useful features, remains still largely unsolved. Some progress has, e.g., been made in the game of Othello, where a simple algorithm, very much like APriori, has been shown to produce valuable conjunctions of basic features (Buro 2002).
Learning Search Control
A more challenging but considerably less investigated task is to automatically tune the various parameters that control the search in game-playing programs. These parameters influence, for example, how aggressive the search algorithm is in pruning unpromising parts of the search tree and which lines are explored in more depth. The key problem here is that these parameters are intertwined with the search algorithm and cannot be optimized independently, making the process very tedious and expensive.

There have been a few attempts to use explanation-based learning to automatically learn predicates that indicate which branches of the search tree are the most promising to follow. These approaches are quite related to various uses of explanation-based learning in planning, but these could not be successfully carried over to game-tree search.

Björnsson and Marsland (2003) present a gradient descent approach that minimizes the total number of game positions that need to be searched in order to successfully solve a number of training problems. The idea is to adjust each parameter in proportion to its sensitivity to changes in the number of searched nodes, which is estimated with additional searches. The amount of positions that can be searched for each training position is bounded in order to avoid infinite solution times for individual problems, and simulated annealing is used to ensure convergence.

Monte Carlo Tree Search
Automated tuning of evaluation functions and search control parameters does not work well for all games. For many years, research in computer Go has not made much progress with conventional search-based and pattern learning algorithms. However, a breakthrough came when Monte Carlo techniques could be combined with tree search algorithms. The basic algorithm interleaves four phases (Browne et al. 2012):

1. Selection: select a node of the current search tree for expansion
2. Expansion: generate one (or more) of the successor nodes for the selected node
3. Simulation: starting from these nodes, simulate a game until a terminal state is reached
4. Backpropagation: propagate the observed result back to the root of the tree

The best known of such methods, UCT, may be viewed as the extension of the UCB method for solving $k$-armed bandit problems to search trees (Kocsis and Szepesvári 2006). In the selection phase, UCT computes the following term for choosing the next node at each interior node of the current tree:

$$ UCT = \bar{X}_j + C \cdot \sqrt{\frac{2 \ln n}{n_j}} $$

(1)

where $X_j$ is the average reward that has been observed at node $j$, $n_j$ is the number of times the node has been visited, and $n$ is the number of times its predecessor has been visited. Clearly, it can be seen that nodes with a high utility are generally preferred (exploitation), but the second term also increases the chances that nodes that have been rarely visited are selected (exploration). The parameter $C$ can be adjusted to trade off exploration and exploitation. From the selected node, a single random rollout is conducted, and its outcome is used to adapt the $X_j$ values in all visited nodes in the search tree.

MCTS is generally applicable but has been particularly successful in game playing, most notably in Computer Go. In particular, AlphaGo (Silver et al. 2016), which employs deep learning for training value networks to evaluate positions and policy networks to bias the simulation phase of MCTS towards promising moves, became the first computer player to beat a world-class Go player in a celebrated 5-game match in March 2016.

Opening 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.

Commercial game-playing programs, in particular chess programs, have thus resorted to tools that support the automatic construction of opening from large game databases. The key challenge here is that one cannot rely on statistical information alone: a move that has been successfully employed in hundreds of games may be refuted in a single game. Donninger and Lorenz (2006) describe an approach that evaluates the “goodness” of a move based on a heuristic formula that has been found by experimentation. This value is then added to the result of a regular alpha-beta search. The technique has been so successful that the chess program HYDRA, probably the strongest chess program today, has abandoned conventionally large man-made (and therefore error-prone) error books. Similar techniques have also been used in games like Othello (Buro 2002).

Pattern Discovery
In addition to databases of common openings and huge game collections, which are mostly used for the tuning of evaluation functions or the automatic generation of opening books (see above), many games or subgames have already been solved, i.e., databases are available in which the game-theoretic value of positions of these subgames can be looked up. For example, in chess all endgames with up to six pieces and in checkers all ten-piece endgames have been solved (Schaeffer et al. 2003). Other games, such as Connect-4, are solved completely, i.e., all possible positions have been evaluated, and the game-theoretic value of the starting position has been determined. Many of these databases are readily available; some of them (in the domains of chess, Connect-4, and tic-tac-toe) are part of the UCI repository for machine learning databases.

The simplest learning task is to train a classifier that is able to decide whether a given game position is a game-theoretical win or loss (or draw). In many cases, this is insufficient. For example, in the chess endgame king-rook-king, any position in which the white rook cannot be immediately captured and in which black is not a stalemate is, in principle, won by white. However, in order to actually win the game, it is not sufficient to simply make moves that avoid rook captures and stalemates. Thus, most databases contain the maximal number of moves that are needed for winning the position. Predicting this is a much harder, largely unsolved problem (some recent work can be found in Sadikov and Bratko 2006). In addition to the game-specific knowledge that could be gained by the extraction of patterns that are indicative of won positions, another major application could be a knowledge-based compression of these databases (the collection of all perfect-play chess endgame databases with up to six men is 1.2 TB in a very compressed database format; the win/loss checkers databases with up to ten men contain about $4 \times 10^{13}$ positions compressed into 215 GB Schaeffer et al. 2003).

Player Modeling
Player modeling is an important research area in game playing, which can serve several purposes. The goal of opponent modeling 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. One of the grand challenges in this line of work is a game like poker, where opponent modeling is crucial to improve over game-theoretical optimal play (Billings et al. 2002).
Player modeling is also of increasing importance in commercial computer games (see below). For one, behavioral cloning techniques could be used to increase the playing strength or credibility of artificial characters by copying the strategies of expert human players. Moreover, the playing strength of the characters can be adapted to the increasing skill level of the human player. Finally, agents that can be trained by nonprogrammers can also play an important role. For example, in massive multiplayer online role-playing games (MMORPGs), an avatar that is trained to simulate a user’s game-playing behavior could take his creator’s place at times when the human player cannot attend to his game character.

Commercial Computer Games
In recent years, the computer game industry has discovered artificial intelligence as a necessary ingredient to make games more entertaining and challenging, and, vice versa, AI has discovered computer games as an interesting and rewarding application area (Laird and van Lent 2001). In comparison to conventional strategy games, computer game applications are more demanding, as the agents in these games typically have to interact with a large number of partner or enemy agents in a highly dynamic, real-time environment, with incomplete knowledge about its states. Tasks include off-line or online player modeling (see above), virtual agents with learning capabilities, optimization of plans and processes, etc.

Computer players in games are often controlled with scripts. Dynamic scripting (Spronck et al. 2006) is an online reinforcement learning technique that is designed to be integrated into scripting languages of game-playing agents. Contrary to conventional reinforcement learning agents, it updates the weights of all actions for a given state simultaneously. This sacrifices guaranteed convergence, but this is desirable in a highly dynamic game environment. The approach was successfully applied to improving the strength of computer-controlled characters and increasing the entertainment value of the game by automated scaling of the difficult level of the game AI to the human player’s skill level. Similar to the problem of constructing suitable features for the use in evaluation functions, the basic tactics of the computer player had to be handcoded. Ponsen et al. (2006) extend dynamic scripting with an evolutionary algorithm for automatically constructing the tactical behaviors.

Machine learning techniques are not only used for controlling players but also for tasks like skill estimation, for example, TrueSkill™ (Herbrich et al. 2007), a Bayesian skill rating system which is used for ranking players in games on the Microsoft’s Xbox 360. SAGA-ML (Southey et al. 2005) is a machine learning system for supporting game designers in improving the playability of a game.

Despite the large commercial potential, research in this area has just started, and the number of workshops and publications on this topic is rapidly increasing. For a list of commercial games using AI techniques, we refer to http://www.gameai.com.

Cross References
- Samuel’s Checkers Player
- TD-Gammon

Recommended Reading


