Temporal Difference Learning
for Nondeterministic Board Games

Glenn F. Matthews and Khaled Rasheed
Department of Computer Science, University of Georgia, Athens, GA, USA

Abstract – We use temporal difference (TD) learning to train neural networks for four nondeterministic board games: backgammon, hypergammon, pachisi, and Parcheesi. We investigate the influence of two variables on the development of these networks: first, the source of training data, either learner-vs.-self or learner-vs.-other game play; second, the choice of attributes used: a simple encoding of the board layout, a set of derived features, or a combination of these. Experimental results show that the TD learning approach is viable for all four games, that learner-vs.-self play can provide highly effective training data, and that the combination of raw and smart features allows for the development of stronger players.

Keywords: Temporal difference learning, Board games, Neural networks, Learning environments, Input encoding

1 Introduction

After Tesauro [1] applied temporal difference (TD) learning [2] to backgammon to produce the skilled backgammon-playing neural network TD-Gammon, others have applied this approach to other board games, including Go, chess, and Othello. However, these other games have generally required specialized variants of the TD learning algorithm [3] or use of extensive domain-specific knowledge [4, 5, 6] to achieve skilled play, in contrast to the straightforward and nearly generic approach that Tesauro was able to use for backgammon.

Baxter et al. [3] have proposed reasons why backgammon is particularly well suited to TD learning, including:

Representation smoothness: Small changes in board position typically result in small changes in game state.

Randomness, which forces the learner to explore the game’s state space, preventing stagnation.

No need for substantial lookahead: It is not necessary to predict the future state of the game to any great depth in order to select the most valuable move to make now.

Given that chess, Othello, and Go each lack these features, it is unsurprising that TD learning has been ineffective for those games. More surprising is the fact that backgammon seems to be the only board game with this set of properties to which TD learning has been applied. This research seeks to improve this situation by studying four games of this sort, specifically: backgammon, to serve as a baseline for comparison; hypergammon (also called hyper-backgammon), a simple backgammon variant; pachisi, a traditional Indian game, and Parcheesi, a commercial American variant of pachisi.

Various experimental approaches have been used within the realm of research involving TD learning, neural networks, and backgammon. Two variables among these have been the learning environment, that is, the source of the training data (self-play by the network being trained [7, 8, 9], play by the network against another opponent [10], or observation of games played by other players [1, 11]), and the representation of the game state (board position) used as inputs for the neural network (simple representation of the board position [1, 7, 8, 9], derived “smart” features [10, 12], or some combination thereof [13]). In this research, we have studied the effect of these two variables on the training of neural networks to play each of the aforementioned four games.

2 Board Games

All four of these games have a basic goal of being the first to bring all of one’s pieces to a designated location, but they differ in many details.

2.1 Backgammon and Hypergammon

Backgammon and hypergammon are each played by two players on a board consisting of 24 spaces (“points”). All rules of play are the same for both games, but in backgammon each player has 15 pieces, while in hypergammon [14] each player has only 3 pieces, making for a faster and more unpredictable game.

Movement is determined by rolling two dice, where each die indicates the distance a piece may be moved; on a roll of doubles, four pieces may be moved instead of two. Players’ pieces move in opposite directions, and after moving all his pieces to one end of the board (his
“home board”), a player may begin using the roll of the dice to remove them from the board. The first player to remove all his pieces wins.

A point containing at least two pieces is safe from enemy attack, but a point occupied by a single piece (a “blot”) is undefended. If a player lands on an enemy blot, it is moved to the “bar”, an off-limits area of the board. Pieces on the bar must be brought back onto the board (reentering into the opponent’s home board) before their owner can move any other pieces.

### 2.2 Pachisi

Pachisi [15] is played on a cross-shaped board (Figure 1) with 68 spaces around the perimeter and 7 spaces inside each arm leading to a central area. The game is played by two teams of two players, with four pieces per player. Pieces begin in the center and proceed down one arm, counterclockwise around the perimeter, then back up the same arm and into the center once more. All eight pieces on a team must complete this route to win.

![Figure 1: Pachisi board.](image)

Piece movement is determined by tossing six coins; depending on the number of “heads”, the player may move one piece a distance of 2, 3, 4, 5, 6, 10, or 25 spaces. Certain counts also grant a “grace,” which allows the player to bring a piece out of the center and then take another turn.

A player’s pieces are safe when on the path in the center of an arm, as no other player may enter there. On the outer board, pieces belonging to members of the same team may share the same space, but a piece landing on any number of enemy pieces captures all of those pieces, sending them back to start again. The “castle” spaces (marked with diamonds) are an exception to this rule; a piece may not land on a castle space that is occupied by any enemy pieces.

### 2.3 Parcheesi

Parcheesi [16] uses a board very similar to pachisi, but the players do not form teams. Each player’s pieces begin off the board and enter onto the “castle” square located counterclockwise from their color’s arm, proceeding clockwise around the board and then up the arm into the central “home” area.

Players roll a pair of dice to move, where each die indicates the distance to move one piece. A roll of five is needed to enter a piece onto the board, and “doubles” earns an extra turn. A player who has entered all four pieces and rolls doubles gets to make four moves, using the numbers on the top and bottom of each die. A player also receives bonus moves, taken immediately, in two cases: a 10-space bonus move when a piece reaches home, and a 20-space bonus move when a capture is made by landing on a lone enemy piece.

As in pachisi, a piece on a castle square may not be captured, but furthermore, in Parcheesi, two allied pieces on any single space form a “blockade,” and no pieces belonging to any player may land on or pass a blockade.

### 3 Game state representations

The initial version of *TD-Gammon* used a truncated partial unary encoding of the raw board state as neural network inputs [1]. In this representation, for each space on the board and for each player, three inputs (each with value 0 or 1) represented the absence or presence of at least 1, 2, or 3 of that player’s pieces, and a fourth integer input represented the number of the player’s pieces present if greater than three. Additional integer inputs represented pieces on the bar or removed from the board, and a pair of binary inputs indicated which player was next to play. Much research since has used similar encodings [7, 8, 9, 13].

In a later version, Tesauro [13] added inputs representing various derived “smart” features of the game state to improve the performance of the program. Others have used similar approaches, even exclusively using derived features in some cases [10, 12]. Yet there has previously been no direct comparison of the effectiveness of these three possible approaches: raw encoding, smart features, or both.

#### 3.1 Unary encoding

For backgammon and hypergammon, we used a truncated unary encoding identical to that of Tesauro [1],
save that we omitted the two “next to play” inputs, which we consider to better fit into the category of “smart” features. Neural networks using this representation thus had 196 inputs instead of Tesauro’s 198.

In pachisi, the number of spaces available to each player is much higher than in backgammon (83 versus 24). Therefore, to keep the number of inputs small, we truncated the encoding to two unary inputs per space per player, ignoring any additional pieces present. (This turned out to be a mistake; see section 6.1.) Combined with the possibility of up to four pieces at start or at finish for each player, these pachisi networks had 696 inputs.

In Parcheesi, no more than two pieces may share a space, so it is unnecessary to truncate the unary encoding. Since each player may have pieces on any of 75 spaces or up to four pieces at the start or finish, these Parcheesi networks had 632 inputs.

3.2 Smart features

After considering the derived board features that had been used with backgammon [8, 17], we selected eight generic features applicable to all four games, including both simple linear functions of the game state and more complex probabilistic features. Each feature represented some aspect of the game state as applicable to a single player, so one copy of this feature set was calculated for each player:

— Turn indicator: set to 1 for the next player to move.
— Pip count: the total number of spaces the player’s pieces need to move in order to win the game.
— Worst piece and best piece: the distance of the player’s pieces farthest from and closest to the goal.
— Contact: The number of enemy pieces the player’s pieces currently needed to pass to reach their goal. (For example, if one piece needed to pass three enemy pieces to reach the goal, and a second piece needed to pass two enemies, the contact value would be five.)
— Exposure: the number of the player’s pieces in vulnerable positions, independent of whether any enemy was currently in position to attack them.
— Hit probability: the probability that any enemy might, on their next turn, be able to hit any vulnerable piece belonging to this player.
— Blockade probability: the probability of making a roll that could not be used to make a complete legal move from the current position.

Each of these features was normalized to the range [0, 1] by dividing it by its theoretical maximum value.

In backgammon and hypergammon, since the players’ pieces move in opposite directions, the contact value for each player is always equal. Therefore, only one value for this feature was needed, and so the derived feature set for backgammon and hypergammon consisted of 15 values. In pachisi and Parcheesi, no such symmetry exists, so each player received one value for each feature, for a total of 32 values.

3.3 Combined representation

A combined representation was created by simply concatenating the two sets of inputs. Thus, for backgammon and hypergammon, the combined feature set consisted of 211 values, for pachisi, 728 values, and for Parcheesi, 664 values.

4 Learning environments

In previous machine learning research in backgammon, some have preferred a self-directed approach in which the learning player plays itself to generate training data [7, 8, 9], while others have preferred to have the learner play against an existing “expert” player as it learns [10]. A third possibility, using training data generated without any input from the learner, has already been shown to be generally inferior [1, 11].

Although training with input from an expert should arguably lead to faster learning, the drawback is that the expert may itself make errors [13]. The learner may learn to imitate this behavior, making the same mistakes as the “expert”, or may learn a style of play that performs well against this particular opponent’s weak areas but poorly in general.

By contrast, when the learner directs its own training, any weaknesses that it may discover in its “opponent” are in fact its own weaknesses, which it should then learn to eliminate. However, there is still the possibility of over-specialization, producing a player that can fight itself to a standstill but doesn’t do well against other players.

We chose to compare the relative effectiveness of these two types of learning environments. Since one of the challenges of training with an expert is selecting an appropriate expert, we modeled this uncertainty by training different learners against either a skilled or a thoroughly unskilled “expert.” For pachisi and Parcheesi, which have four players instead of two, the learner played with three identical opponents in each case.

To serve as an unskilled “expert” for all four games, we created RandomPlayer, which simply chooses randomly among all legal moves in a given position.

For a skilled backgammon opponent, we selected the program pubeval [18], which is freely available and has been widely used in past research [7, 8, 10]. Since hypergammon is a subset of backgammon, we used pubeval for that game as well.
Since there has not been the same level of study of pachisi and Parcheesi, no standard “expert” player exists. We designed and implemented HeuristicPachisiPlayer and HeuristicParcheesiPlayer to fill this role. These simple players were based upon maximizing piece advancement at each move, with secondary goals such as placing pieces on castle spaces to act as tiebreakers between otherwise equivalent moves.

5 Experimental procedure

5.1 General setup

For each game, five combinations of the two experimental variables were tested:

1. Self-directed learning with unary board encoding
2. Self-directed learning with derived features
3. Self-directed learning with combined feature set
4. Learning vs. RandomPlayer with unary encoding
5. Learning vs. “expert” with unary encoding

The learning game player in each experiment consisted of a neural network with one input for each feature in the selected feature set, a single hidden layer of ten units, and one output unit per player (representing the perceived strength of that player’s position). Sigmoidal activation functions were used for the hidden and output units, and all network weights were initialized randomly in the range [−1.0, 1.0]. To play the game, after each roll, the board position resulting from each possible move was fed into the neural network, and the move resulting in the highest output value for that player was chosen.

5.2 Learning

The TD(λ) algorithm by Sutton [19] was used for learning, with parameters of λ = 0.7 and α = 0.3. (This is the same exponential weighting factor used in [1], but a faster learning rate). TD(λ) was applied to the network after each move of the game, and at the end of each training game, TD(λ) was applied once more to train the network against output values of 0.1 for each losing player and 0.9 for each winning player.

Each network was trained for 50,000 games, and as learning proceeded, we periodically recorded the network weights for later evaluation. Each learning experiment was repeated five times, using a different random set of initial network weights and different random dice rolls in each case. In most cases all five runs produced networks whose evaluated skill levels were consistent with one another to within a few percent.

5.3 Evaluation

Since these are nondeterministic games, we could not simply assume that the winner of a single match would necessarily be the stronger player. We performed some experiments that suggested that playing 5,000 games provided sufficient data to estimate the relative strength of players to within 3%.

For backgammon and hypergammon, this opponent was pubeval. For pachisi, a team consisting of two identical copies of the trained network played against a team of two HeuristicPachisiPlayers. For Parcheesi, the trained network played against three HeuristicParcheesiPlayers. Thus, for backgammon, hypergammon, and pachisi, 50% wins by the network indicated that it had achieved performance on par with its opponent, while for Parcheesi, 25% wins indicated parity.

6 Results

6.1 Game state representations

Table 1: Performance of game state representations

<table>
<thead>
<tr>
<th>State representation</th>
<th>After 5,000</th>
<th>After 50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backgammon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary encoding</td>
<td>17.3±5.3%</td>
<td>42.6±4.2%</td>
</tr>
<tr>
<td>Smart features</td>
<td>28.1±0.8%</td>
<td>31.0±1.3%</td>
</tr>
<tr>
<td>Combination</td>
<td>24.2±4.9%</td>
<td>49.4±1.6%</td>
</tr>
<tr>
<td><strong>Hypergammon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary encoding</td>
<td>43.8±1.8%</td>
<td>58.0±1.2%</td>
</tr>
<tr>
<td>Smart features</td>
<td>62.6±0.7%</td>
<td>65.4±1.2%</td>
</tr>
<tr>
<td>Combination</td>
<td>57.8±1.2%</td>
<td>67.7±0.5%</td>
</tr>
<tr>
<td><strong>Pachisi</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary encoding</td>
<td>0.1±0.1%</td>
<td>18.5±10.1%</td>
</tr>
<tr>
<td>Smart features</td>
<td>55.1±1.2%</td>
<td>64.9±1.8%</td>
</tr>
<tr>
<td>Combination</td>
<td>7.4±2.4%</td>
<td>63.9±12.2%</td>
</tr>
<tr>
<td><strong>Parcheesi</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary encoding</td>
<td>10.6±1.2%</td>
<td>34.0±1.3%</td>
</tr>
<tr>
<td>Smart features</td>
<td>25.8±5.4%</td>
<td>30.4±1.2%</td>
</tr>
<tr>
<td>Combination</td>
<td>17.3±1.7%</td>
<td>41.2±1.0%</td>
</tr>
</tbody>
</table>

Figure 2 shows the skill level (mean of 5 runs) achieved as self-directed training proceeded for networks using each game state representation. Table 1 compares the strength of each network (mean and standard deviation of five runs) at two stages: after 5,000 games, at which point most rapid initial learning had already taken place, and after 50,000 games, at which point performance had mostly leveled off.
We observe that the results for pachisi show slow initial learning and poor maximum performance for the learner using the unary encoding, and inconsistent maximum performance for both the unary encoding and the combined feature set. This reflects a flaw in the unary encoding selected for that game.

By truncating the representation as we did, pieces could “disappear” — when many pieces shared a space, all but the first two would be hidden from the learner. With this representation, moving into such a position might appear to be advantageous, as it could “remove” a previously vulnerable piece from the board. However, in pachisi, such a position would often be very weak, since an opponent landing on this highly occupied space will gain a tremendous advantage by sending so many pieces back to start.

We also observe that networks using the smart features alone achieved excellent performance within the first 5,000 games of training, while those that used the unary features in some way still had much to learn at that stage. It appears that smart features alone were generally not sufficient to capture every nuance of strategy, but they came close for hypergammon (likely due to its simplicity) and pachisi (most likely due to the aforementioned weaknesses of the unary representation selected for that game).

6.2 Learning environments

Figure 3 shows the skill level (mean of 5 runs) achieved as training proceeded for each game under each learning environment, using the unary board encoding as network inputs. Table 2 compares the strength of each network (mean and standard deviation of five runs) at two stages: after 5,000 games, at which point most rapid initial learning had already taken place, and after 50,000 games, at which point performance had mostly leveled off.

We observe that in all four games, learning by playing against RandomPlayer produced a significantly weaker network than the other two learning environments. We
believe that this shows that once the learner managed to surpass the skill level of its opponent, its subsequent learning was severely hampered by the lack of a serious challenge to train against. Although this is a worst-case scenario, these results should serve as a caution against the use of any non-learning opponent for training.

We also observe that training against the “strong” opponent produced a network of at best slightly superior skill versus that very same opponent than a network trained by self-play that had never competed against that opponent before. It seems plausible to argue that this performance gap would narrow (or even be reversed) against any other opponent.

It is worth noting that these observations apply even to pachisi — despite the previously discussed flaws in the unary board encoding used for this game, RandomPlayer produced a significantly weaker player than either of the other two learning environments, which were approximately equivalent.

Table 2: Performance of learning environments

<table>
<thead>
<tr>
<th>Learning environment</th>
<th>After 5,000</th>
<th>After 50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backgammon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learner-vs.-self</td>
<td>17.3±5.3%</td>
<td>42.6±4.2%</td>
</tr>
<tr>
<td>Learner-vs.-Random</td>
<td>4.9±0.4%</td>
<td>5.5±0.5%</td>
</tr>
<tr>
<td>Learner-vs.-pubeval</td>
<td>30.4±1.6%</td>
<td>45.2±1.3%</td>
</tr>
<tr>
<td><strong>Hypergammon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learner-vs.-self</td>
<td>43.8±1.8%</td>
<td>58.0±1.2%</td>
</tr>
<tr>
<td>Learner-vs.-Random</td>
<td>38.1±0.9%</td>
<td>40.1±0.9%</td>
</tr>
<tr>
<td>Learner-vs.-pubeval</td>
<td>50.0±2.3%</td>
<td>60.3±0.9%</td>
</tr>
<tr>
<td><strong>Pachisi</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learner-vs.-self</td>
<td>0.1±0.1%</td>
<td>18.5±10.1%</td>
</tr>
<tr>
<td>Learner-vs.-Random×3</td>
<td>0.1±0.1%</td>
<td>5.7±5.3%</td>
</tr>
<tr>
<td>Learner-vs.-Heuristic×3</td>
<td>0.3±0.3%</td>
<td>16.0±6.3%</td>
</tr>
<tr>
<td><strong>Parcheesi</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learner-vs.-self</td>
<td>10.6±1.2%</td>
<td>34.0±1.3%</td>
</tr>
<tr>
<td>Learner-vs.-Random×3</td>
<td>15.3±2.3%</td>
<td>26.0±0.7%</td>
</tr>
<tr>
<td>Learner-vs.-Heuristic×3</td>
<td>15.2±1.8%</td>
<td>33.6±0.8%</td>
</tr>
</tbody>
</table>
7 Conclusions

We argue that the presented experimental results clearly demonstrate that TD(λ) can be used successfully to train neural networks as players of nondeterministic board games. Furthermore, for such games, the basic TD(λ) algorithm suffices, without any of the specialized modifications that have been necessary for success with games such as chess.

It is also clear that in such games, a self-directed training approach is capable of producing very strong players without any input from existing “experts;” indeed, if the “experts” have significant limitations, their input may severely hamper learning.

Furthermore, both naive unary board encodings and derived “smart” features can be valuable as neural network inputs to produce strong players, so long as the encoding and feature set are selected wisely. In general, combining the two yields the strongest players given sufficient training time, but if only a short training period is possible, a good “smart” feature set by itself can produce a reasonably strong player in that time frame.

References


