USING TEMPORAL DIFFERENCE LEARNING TO TRAIN PLAYERS OF NONDETERMINISTIC BOARD GAMES

by

GLENN F. MATTHEWS

(Under the direction of Khaled Rasheed)

ABSTRACT

Temporal difference learning algorithms have been used successfully to train neural networks to play backgammon at human expert level. This approach has subsequently been applied to deterministic games such as chess and Go with little success, but few have attempted to apply it to other nondeterministic games. We use temporal difference learning to train neural networks for four such games: backgammon, hypergammon, pachisi, and Parcheesi. We investigate the influence of two training variables on these networks: the source of training data (learner-versus-self or learner-versus-other game play) and its structure (a simple encoding of the board layout, a set of derived board features, or a combination of both of these). We show that this approach is viable for all four games, that self-play can provide effective training data, and that the combination of raw and derived features allows for the development of stronger players.

INDEX WORDS: Temporal difference learning, Board games, Neural networks, Machine learning, Backgammon, Parcheesi, Pachisi, Hypergammon, Hyper-backgammon, Learning environments, Board representations, Truncated unary encoding, Derived features, Smart features

USING TEMPORAL DIFFERENCE LEARNING TO TRAIN PLAYERS OF NONDETERMINISTIC BOARD GAMES

by

GLENN F. MATTHEWS

B.S., Georgia Institute of Technology, 2004

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2006

USING TEMPORAL DIFFERENCE LEARNING TO TRAIN PLAYERS OF NONDETERMINISTIC BOARD GAMES

by

GLENN F. MATTHEWS

Approved:

Major Professor:	Khaled Rasheed
Committee:	Walter D. Potter
	Prashant Doshi

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2006

DEDICATION

To Heather.

Thank you for everything.

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

CHAPTER 1

INTRODUCTION

The use of machine learning techniques to develop skilled computer players of board games has a long history,[1] dating back at least to the research by Samuel (1959) with checkers. One of the most famous successes in this area was the work of Tesauro (1992), in applying temporal difference learning to train a neural network to play backgammon.

Temporal difference (TD) learning is a class of algorithms designed for prediction learning problems, in which past experience of a system is used to learn to predict its future behavior. Although formalized by Sutton (1988, 1989), these algorithms were first used by Samuel (1959) in his checkers-playing program, and gained prominence after they were applied by Tesauro (1992) to playing the game of backgammon.

Tesauro used TD learning to train a simple multi-layer neural network to play backgammon. The resulting program, TD-Gammon, surpassed any other backgammon program in existence when it was created, played at a world-class human level for its time, and has served as a design template for several commercial backgammon-playing programs that continue to play at world-class levels (Tesauro 2002).

The tremendous success of TD-Gammon inspired many others to apply TD learning to other board games, including Go, chess, and Othello, among others, but they were generally far less successful. Those that achieved skilled levels of play did so only by inventing variants of the TD learning algorithms (Baxter et al. 1998) or incorporating extensive domain-specific knowledge into their neural network architecture (Schraudolph et al. 1994; Thrun 1995; Leouski 1995), in contrast to the straightforward and nearly generic approach that worked so well for Tesauro.

[1] At least, long compared to the history of other areas in computer science; board games themselves are orders of magnitude older.

Baxter et al. (1998) proposed a number of reasons why backgammon is particularly well suited to TD learning:

speed of play The learner can play and/or observe thousands of games.

representation smoothness Small changes in board position or other aspects of the game representation typically have similarly small effect on the state of the game.

randomness Factors beyond the learner's control force some exploration of 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 then that chess, Othello, checkers, and Go are all deterministic games in which a small difference in board position can drastically affect gameplay and which involve substantial lookahead, it is unsurprising that the application of TD learning to these games has not met with much success. Instead, it would seem that success is likely to be found by applying TD learning to games that are similar to backgammon; more specifically, by investigating other games that are nondeterministic in nature, not strongly affected by small shifts in board position, and do not require substantial lookahead.

Surprisingly, backgammon seems to be the *only* board game of this sort that has been subjected to machine learning research; all of the other well-studied games (chess, checkers, Othello, and Go, primarily) are deterministic in nature. The research presented here seeks to broaden this field somewhat by studying several nondeterministic games, specifically: backgammon, since it has already been shown conclusively to be susceptible to such an approach and so can serve as a baseline for comparing other games to; hypergammon (also called hyper-backgammon), a much-simplified backgammon variant; pachisi, a traditional game of India, and Parcheesi, a modern, commercial, American variant of pachisi.

Even within the fairly narrow realm of research involving TD learning, neural networks, and backgammon, there have been a variety of experimental approaches used by past researchers. Two

variables among these approaches have been the learning environment, that is, the source of the training data (self-play by the network being trained, play by the network against another opponent, and games played by other players), and the representation of the game state (board position) to use as inputs for the neural network (naïve representations of the board position, derived "smart" features, or some combination thereof). Therefore, 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.

The remainder of this thesis is organized into the following chapters:

- An explanation of the TD learning algorithm and its application to neural networks.
- An introduction to the four games studied in this work.
- An overview of past research into nondeterministic board games and machine learning.
- An overview of our research approach.
- Experiments involving the learning environment and their results.
- Experiments involving the board representation and its results.
- Conclusions and possible directions for future work.

CHAPTER 2

TEMPORAL DIFFERENCE LEARNING AND NEURAL NETWORKS

In temporal difference (TD) learning, a learner is given a series of successive observations of a system, terminating in a final outcome or state. After each observation is provided, the learner makes a prediction of the final outcome. Then, in order to learn, this prediction is compared, not to the actual outcome, which is unknown to the learner at this time, but to the previous predictions made thus far. Since the system is nearer to its final state than it was previously, the learner assumes that its most recent prediction is more accurate than its previous predictions, and updates itself so that the previous predictions, if made again, would be more similar to the most recent prediction.

2.1 THE TD(λ) ALGORITHM

Sutton (1988) defined a family of TD learning procedures, TD(λ), based on exponential weighting of past predictions to modify more recent predictions more strongly than older predictions. Sutton formalized TD(λ) with the equation

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \,, \tag{2.1}$$

where w is the vector of weights that affect the prediction, t is the current observation time, α is a learning rate constant, P_t is the prediction made at time t, λ is the exponential weighting factor ($0 \leq \lambda \leq 1$), and $\nabla_w P_k$ is the vector of partial derivatives of P_k with respect to each component of w.

Furthermore, as Sutton noted, the summation in equation 2.1 only changes incrementally as t increases, and so can be calculated quite efficiently. Defining e_t to be the value of that sum at time

t, then

$$\begin{aligned} e_{t+1} &= \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla_w P_k \\ &= \nabla_w P_{t+1} + \sum_{k=1}^{t} \lambda^{t+1-k} \nabla_w P_k \\ &= \nabla_w P_{t+1} + \lambda e_t \ , \end{aligned}$$

and so

$$\Delta w_t = \alpha (P_{t+1} - P_t)(\nabla_w P_t + \lambda e_{t-1}) \ . \tag{2.2}$$

2.2 NEURAL NETWORKS

Neural networks (NNs) are a well-known machine learning and function approximation tool (Mitchell 1997). A neural network consists of a set of input units, a set of output units, optionally a set of intermediate "hidden" units, and weighted connections from one unit to another. Although arbitrary structure is possible, most networks ("feed-forward" NNs) use a layered structure in which units are grouped into layers (one input layer, one output layer, and zero or more hidden layers) and each of the units in one layer connects to each of the units in the next layer, but no others (Figure 2.1).

Aside from the input units, whose value is set externally, the value of each unit (or neuron) is obtained by taking the weighted sum of the values of each unit that has a connection to it, then applying a thresholding function of some sort to this sum to map it to a desired range of values (Figure 2.2). Most neural networks use a sigmoid thresholding function to map the value to the range (0, 1), although Tesauro's *pubeval* can be thought of as a neural network with no hidden layer and no thresholding function (Tesauro 1993).

2.3 APPLYING TD(λ) TO NEURAL NETWORKS

Using multi-layer neural networks as learners, the application of TD(λ) is slightly more complex (Sutton 1989). Since some network weights do not affect some network outputs, and the effect

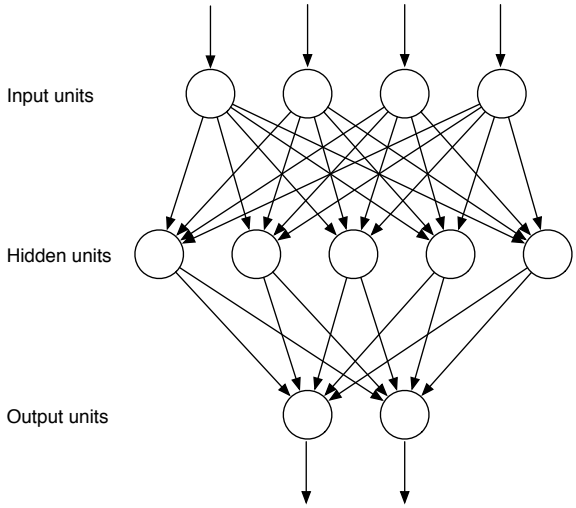

Figure 2.1: A simple neural network with one hidden layer.

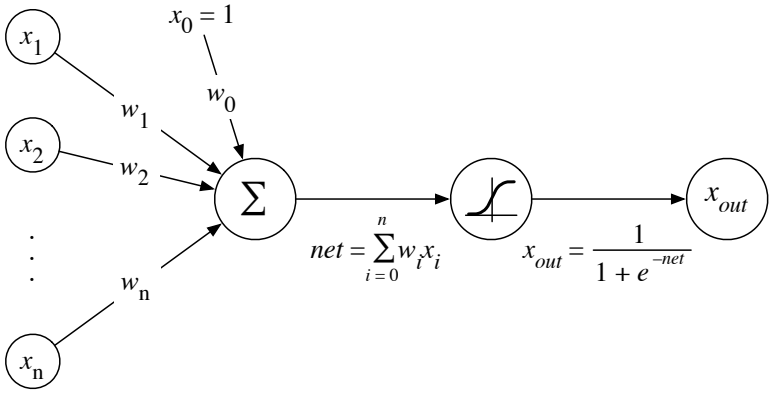

Figure 2.2: A neuron with sigmoid thresholding, after Mitchell (1997).

of other weights depends on the strength of other weights, it is necessary to define e^t_{jkl}, the "eligibility" (or influence) of the weight from unit j to unit k with respect to output unit l at time t. Given a basic three-layer network (one layer each of inputs, hidden units, and outputs), the eligibility of the weight from a hidden unit h to an output unit o with respect to that same output is

$$e^t_{hoo} = \lambda e^{t-1}_{hoo} + y^t_o(1-y^t_o)y^t_h \,, \tag{2.3}$$

while the eligibility of that weight with respect to any other output is zero. Similarly, the eligibility for the weight from an input unit i to a hidden unit h with respect to any particular output o is

$$e^t_{iho} = \lambda e^{t-1}_{iho} + (y^t_o(1-y^t_o))w^t_{ho}(y^t_h(1-y^t_h))y^t_i \,, \tag{2.4}$$

where y^t_x is the output value of unit x of the network at time t.

If O is the set of all outputs of the network, then the weight update rule is

$$w^{t+1}_{jk} = w^t_{jk} + \alpha \sum_{l \in O} (P^{t+1}_l - P^t_l) e^t_{jkl} \,. \tag{2.5}$$

CHAPTER 3

BOARD GAMES

The four board games studied in this research are backgammon, hypergammon, pachisi, and Parcheesi. Although all are nondeterministic games in which the fundamental objective is to be the first to bring all of one's playing pieces to a designated goal location, they are different in many details (Table 3.1). We present in this chapter a brief overview of the significant features of these games; for full rules of each game, see the relevant references.

3.1 BACKGAMMON

Backgammon (Keith 2006) is a two-player game played on a board consisting of 24 spaces or "points" with 15 pieces per player (Figure 3.1). Movement is based on a roll of two 6-sided dice, where the roll of each die indicates a distance a single piece may be moved, and in the case of a roll of doubles, four pieces may be moved. The players move their pieces in opposite directions; after bringing all 15 pieces to his "home board" (the last 6 spaces at his end of the board), a player may begin "bearing off" the pieces, bringing them "home" and off the board (Figure 3.2). The first player to bear off all 15 of his pieces wins the game.

There are several key strategic features of the game. Although a point containing at least two of a player's pieces is safe from enemy attack, a point occupied by a single piece (a "blot") is undefended. If a player lands one of his pieces on a blot, the enemy piece is moved to the "bar", an off-limits area of the board that is conceptually located just beyond the far end of the board. A player with pieces on the bar must bring them back onto the board (reentering them into his opponent's home board area) before he is permitted to move any of his other pieces. Therefore, it is important defensively to keep pieces in groups to protect them, and an effective offensive

Table 3.1: Comparison of board games studied.

	Backgammon	Hypergammon	Pachisi	Parcheesi
Number of players	2	2	4 (in teams of 2)	4
Pieces per player	15	3	4	4
Number of board spaces accessible by all players	24	24	68	68
Number of board spaces private to each player	2 per player	2 per player	16 per player	9 per player
Direction of movement	opposite directions	opposite directions	same direction	same direction
Moves per turn	up to 4	up to 4	1	up to 4
Bonus moves possible	no	no	no	yes
Bonus turns possible	no	no	yes	yes
Declining to move permitted	no	no	yes	no
Randomizer	Two 6-sided dice	Two 6-sided dice	Six cowry shells (coin flips)	Two 6-sided dice

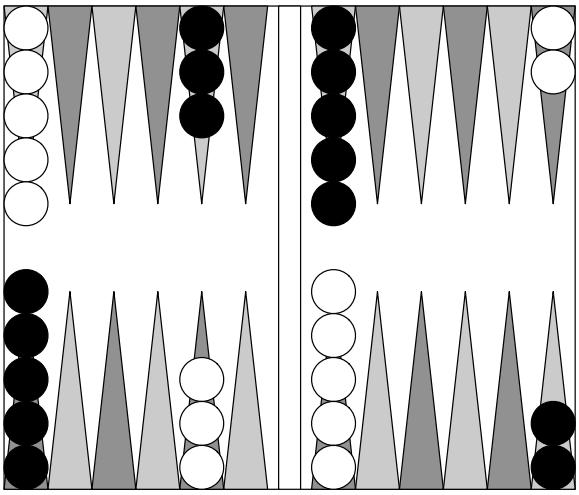

Figure 3.1: The backgammon board and initial setup of pieces.

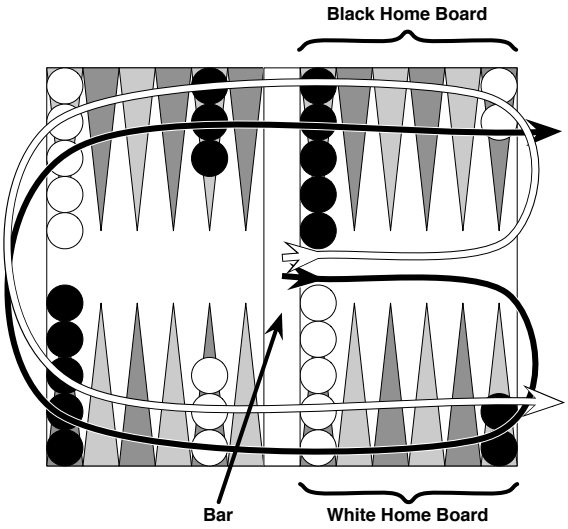

Figure 3.2: Backgammon board features and piece movement.

strategy can be to send enemy pieces to the bar and then defend one's home board spaces so as to make it difficult or impossible for the opponent to reenter those pieces.

In discussion of backgammon, as well as the design of various past backgammon programs, a distinction is drawn between "contact" positions, in which the players' pieces can interact, and offensive and defensive concerns come into play, and "race" positions, endgame positions in which no such interaction may occur and the only concern is to bear off pieces as quickly as possible.

3.2 HYPERGAMMON

Hypergammon (Sconyers 2006) is a simplified variant of backgammon. All rules of play are the same, but each player uses only 3 pieces instead of the 15 used by backgammon, and the initial setup is different (compare Figures 3.1 and 3.3). Since there are far fewer pieces available, the elaborate defensive positions typical of backgammon do not often come into play in hypergammon, and gameplay is both faster and more dominated by chance.

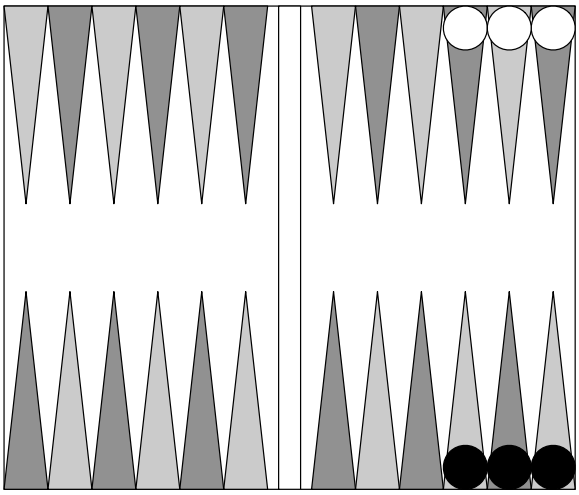

Figure 3.3: The hypergammon board and initial setup of pieces.

3.3 PACHISI

Pachisi is a traditional game of India and as such, has somewhat variable rules, so we elected to use the set of rules suggested by Masters Games Ltd. (1999). The game is played on a cross-shaped board (Figure 3.4) with 68 squares around the outer edge of the cross plus 7 squares on the interior of each arm and a central area called the "Charkoni." The game is played by four players (in teams of 2) who control four pieces each. The pieces all begin in the Charkoni and proceed down the center of the appropriately-colored arm, counterclockwise around the outside of the board, then back up the same arm and into the Charkoni once more (Figure 3.5). All eight pieces on a team must circle the board and return to the Charkoni for that team to win the game.

Unlike the other games studied, Pachisi does not use modern dice, but instead uses a throw of six cowry shells, which is essentially equivalent to the toss of six coins as each cowry may land "up" or "down." On his turn, a player moves one piece some distance based on the number of "up" cowries (Table 3.2). Certain throws of the cowries grant a "grace," which has two features: first,

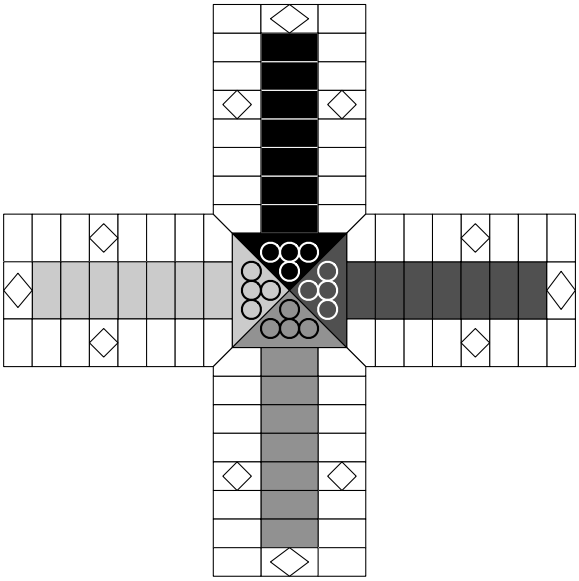

Figure 3.4: The pachisi board and initial setup of pieces.

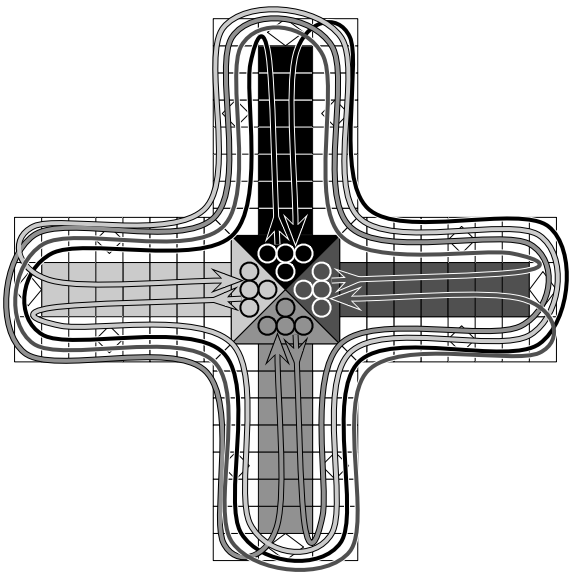

Figure 3.5: Pachisi piece movement.

Table 3.2: Possible moves in Pachisi

# of “up” cowries	Distance moved
0	25 + grace
1	10 + grace
2	2
3	3
4	4
5	5
6	6 + grace

it allows the player to move a piece out of the Charkoni, and second, it grants the player another throw of the cowries and another turn.

Uniquely among the games studied, in Pachisi a player may opt to pass rather than move any piece on his turn. Furthermore, at the player’s discretion, a piece completing its circuit of the board may continue to circle around the outside of the board rather than entering the path back into the Charkoni. Due to the team play aspect of pachisi, this may be a useful strategy, as a player with all of his pieces returned to the Charkoni is unable to take any action to assist his teammate.

A player’s pieces are completely safe when moving in either direction along the path between the outer board and the Charkoni, as no other player’s pieces may enter that area. Any number of pieces belonging to members of the same team may share the same space on the outer board, but there is no safety in numbers, as a piece landing on any number of enemy pieces captures all of those pieces, sending them all back to the Charkoni to begin again. The only exception to this rule are the twelve “castle” spaces (marked with diamonds on Figure 3.4); a piece may not land on a castle space that is occupied by any number of enemy pieces.

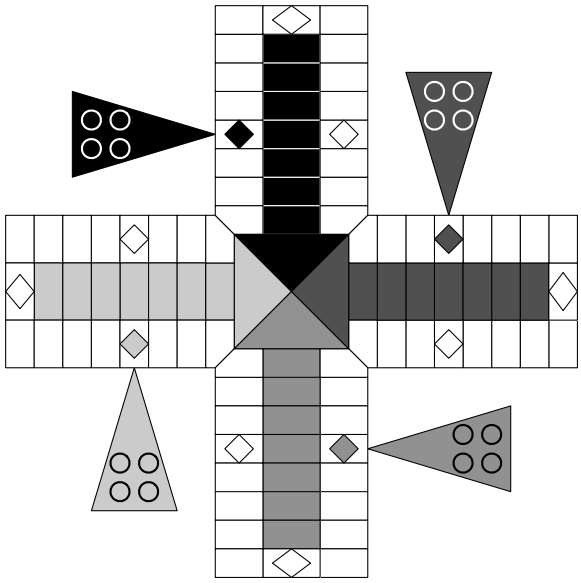

Figure 3.6: The Parcheesi board and initial setup of pieces.

3.4 PARCHEESI

Parcheesi (Selchow & Righter Company 1987) uses a board layout very similar to that of its parent, pachisi (Figure 3.6). Unlike pachisi, in Parcheesi pieces do not begin in the central area (here called "home"), but instead begin in a separate starting area and enter from there directly onto the outer board at the first castle square counterclockwise from their designated path to home, proceeding clockwise from there (Figure 3.7). Team play is an optional rule in Parcheesi, but was not used in this research, in order to emphasize the differences between Parcheesi and pachisi and allow for the study of a game with more than two competing sides.

Piece movement is controlled by the roll of a pair of dice, where each die indicates the distance to move one piece. A roll of five (on one die or as the sum of both dice) may be used to enter a piece onto the board, and a roll of doubles entitles the player to an extra turn, up to a maximum of three consecutive turns. A player who has entered all four of his pieces and rolls doubles gets to

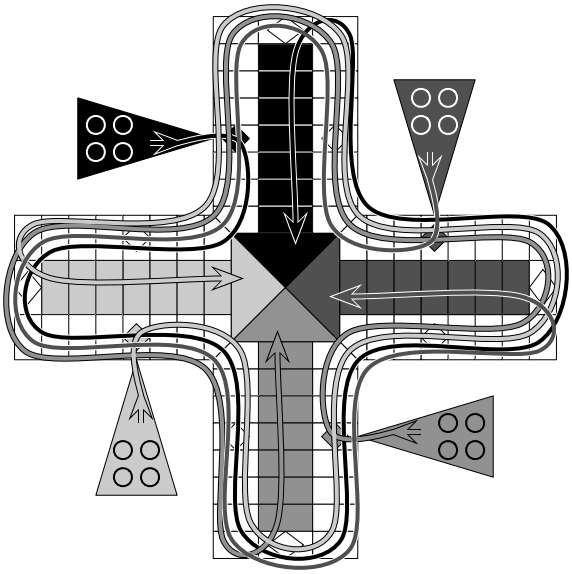

Figure 3.7: Parcheesi piece movement.

make four moves with that roll: the number showing on each die and the number on the underside of each die as well (hence, since opposite sides of the die always add to 7, a total of 14 spaces will be moved on any roll of doubles).

Uniquely to Parcheesi, a player receives bonus moves, to be taken immediately, under two circumstances: a 10-space bonus move when a piece is moved home (by exact count only) and a 20-space bonus move when a capture is made by landing a piece on a single 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 over a blockaded space.

3.5 Conclusion

These four games have nondeterminism and movement along a board in common, but differ in many specifics of their board layout and rules of play. They provide a small sample of the many various nondeterministic board games that exist.

CHAPTER 4

PREVIOUS RESEARCH INVOLVING MACHINE LEARNING AND NONDETERMINISTIC BOARD GAMES

From a machine learning standpoint, a nondeterministic board game can be thought of as a Markov decision process (MDP) (Kaelbling et al. 1996) in which the reward function is zero for all transitions except those transitions that lead to an end state of the game. For these games, where substantial lookahead is not generally feasible, greedy strategies for both learning and game playing will probably be as successful as any other approach, but any learning algorithm to be used for these games must be one that can cope with the delayed rewards inherent in these games.

4.1 MACHINE LEARNERS AND LEARNING TECHNIQUES USED

4.1.1 BACKPROPAGATION

Perhaps the earliest application of machine learning techniques to nondeterministic board games was the application of backpropagation neural networks (Mitchell 1997) to backgammon by Tesauro and Sejnowski (1989), who trained networks to act as move selectors, using a set of 3202 backgammon moves, a subset of which were hand-labeled with scores from -100 to 100, representing the quality of that move. Up to 20 cycles through the training data were required to achieve optimal performance.

In order to improve the performance of the network, *Neurogammon*, beyond a basic level, Tesauro and Sejnowski had to observe the network and recognize mistakes that it made in playing the game, then add hand-crafted examples to the training set to correct these mistakes. In some cases, many such examples had to be added in order to train the network to correctly distinguish

between such mistakes and other game positions where the "mistake" would in fact be the correct move.

A second approach Tesauro and Sejnowski took to improving the network was to alter or augment the hand-crafted board features that were used as the inputs to the network in order to help the network more clearly distinguish between various positions and strategies.

Tesauro and Sejnowski concluded that *Neurogammon* had learned a number of key features of good backgammon play but had failed to learn others, yielding good average-case performance but very poor worst-case performance. Furthermore, although its performance was fairly strong compared to other programs of the time, further improvement of the network by this approach seemed unlikely (Tesauro and Sejnowski 1989):

> "Probably what is required [to eliminate the weaknesses of the program] is either an intractably large number of examples, or a major overhaul in either the pre-computed features or the training paradigm."

4.1.2 Temporal difference learning

TD-Gammon and related programs

Shortly after the formalization of the TD(λ) algorithm, Tesauro (1992) applied it to train a neural network to play backgammon. In this approach, the training positions for the network were generated purely by self-play by the network itself. Tesauro found that optimal performance was achieved after about 50,000 training games for a network with 10 hidden nodes, 100,000 with 20 hidden nodes, and 200,000 with 40 hidden nodes. In later research using a more complex set of inputs representing the game state, a network with 160 hidden units required more than 6,000,000 training games to achieve optimal performance (Tesauro 2002).

Although this approach is obviously rather time-intensive, requiring a tremendous number of games played in order to achieve optimal performance, it has the dual advantages of relative simplicity and effectiveness. Maximum performance was achieved by using hand-designed inputs in

combination with a naïve encoding of the raw board state, but even the early versions of *TD-Gammon* that used only raw features became extremely strong players.

Nearly all of the best publicly available modern backgammon programs, including *Snowie 3*, *Jellyfish*, and *gnubg* among others, were developed with approaches patterned after *TD-Gammon* (SnowieGroup SA 2006; JellyFish AS 2004; Heled 2003).

MULTI-NEURAL-NETWORK APPROACHES

Several researchers have investigated whether Tesauro's results could be improved upon by structuring the learner as a set of neural networks instead of a single network, and training each network to handle a specific set of situations. Obviously, the primary drawbacks of this approach are the additional expert knowledge required (to determine a reasonable decomposition of the game into distinct situation categories) and the increased complexity of the resulting program.

Heinze et al. (2005) created a learner consisting of two neural networks, used to represent strategies for the "contact" and "race" stages of the game. Each network had 80 hidden nodes, and the learner was trained for 400,000 games. This was not the primary focus of that work; rather, it was created solely for comparison to the effectiveness of their program *Fuzzeval*, described in a later section. We have thus elected to refer to this program as *Fuzzeval-TDNN*.

Wiering et al. (2005) used a learner consisting of nine neural networks. Simple rules corresponding to various types of game positions (early endgame, late endgame, positions where one player is at a substantial disadvantage, etc.) were used to select which network learned from and made decisions in any given position. Several different input representations of various sizes were used, corresponding to the varying complexity of different types of positions (for example, some board spaces are by definition not occupied during the late endgame, in which case, no input representation is needed for those points). This learner (unnamed by Wiering et al.; we shall call it *WPM-9*) was trained by self-play for more than 1,000,000 games.

Wiering et al. also evaluated other, simpler neural network based learners. These had only three neural networks instead of nine, corresponding to contact positions, early race positions (no

more contact, but pieces not yet all in the home board), and late race positions (all pieces in their respective home boards). These learners, which we will call *WPM-3* collectively, were trained by playing against *WPM-9* for up to 200,000 games.

4.1.3 *HC-Gammon*

Pollack and Blair (1998) used simple hill climbing to train a neural network to play backgammon. Beginning with initial network weights of zero, the network was trained as follows. A "mutant" network was generated by adding gaussian noise to each of the network weights, and the two networks were compared by playing a few games of backgammon against one another. If the mutant won a majority of the games, it would replace the current network, becoming the new "champion."

After some experimentation, Pollack and Blair made three modifications to the hill climbing algorithm, seeking to improve their results by reducing the likelihood that a streak of luck by a series of inferior mutants would cause the network to worsen in performance. First, games were played in parallel; that is, two games were played using the same sequence of dice rolls but with the player roles reversed, so that the effect of good luck with the dice would be decreased. Second, only partial replacement was used — if the mutant was superior to the champion, the champion would not be fully replaced by the mutant, but would instead change to a weighted average of the two individuals (95%/5% champion/challenger). Third, a form of annealing was used, such that when the champion reached a certain level of performance, the mutant had to win a greater number of games in order to cause any change to the champion. Pollack and Blair used this approach to train the network for up to 100,000 generations (at 2-8 games played per generation).

The most obvious drawback of this approach is the usual weakness of hill-climbing, that of potentially becoming caught in a local optimum that is substantially inferior to the global optimum strategy. Nonetheless, the elegant simplicity of this approach is quite compelling.

4.1.4 *ACT-R-Gammon*

Sanner et al. (2000) used a learning approach based on their ACT-R theory of human learning and cognition. In this approach, the learner, *ACT-R-Gammon*, consisted of a set of observed board features and the learned value of each. The learner stored newly encountered features only if they were sufficiently distinct from previously stored features, as defined by a partial mismatch penalty function. The value of a feature was based on a naïve Bayesian classifier approach (based on the number of winning and losing games in which that feature appeared), and the overall value of any observed set of features was calculated by the weighted nearest neighbor algorithm.

ACT-R-Gammon was trained by playing 1000 games against the program *pubeval* (Tesauro 1993). The performance of *ACT-R-Gammon* did not necessarily reach a maximum at that number of games; rather, the authors chose to stop learning at that point, justified as being the approximate number of games after which a human player could be expected to develop a fairly high level of skill (Sanner et al. 2000). This is probably the most impressive aspect of this program: its ability to achieve fairly strong play in several orders of magnitude less training time than other approaches.

4.1.5 GA-BASED MULTI-AGENT REINFORCEMENT LEARNING BIDDING

Qi and Sun (2003) used a complex approach combining genetic algorithms (GAs), reinforcement learning, and multi-agent bidding approaches, which they called GA-based multi-agent reinforcement learning bidding (GMARLB). More specifically, in their approach, a learner team consisted of a number of agents, each of which consisted of a pair of neural networks. One of these networks represented move selection (which move was best from a given position) while the other represented the agent's knowledge of which positions it was good at handling, versus which positions it should open up to bidding from the other agents for the opportunity to take charge in its place.

15 such teams were initially created. Each team played 200 games, using Q-learning (Mitchell 1997) to train the agents as it played. At that point, tournament selection was used to select the 5 best teams, and GA-style crossover and mutation was used to replace the other 10 teams with

descendants of the survivors. Each of these new 15 teams was trained for another 200 games, then the GA was applied again, and so forth, for a total of 400,000 games.[1]

Although this approach is novel, it seems perhaps overly complex, especially considering that simpler approaches by others have achieved performance comparable or superior to *GMARLB-Gammon*.

4.1.6 GENETIC PROGRAMMING

Azaria and Sipper (2005) applied a variant of genetic programming (GP) (Koza 1992), called strongly typed genetic programming (Montana 1995), to evolve backgammon strategies. The population consisted of 128 individual strategies, each represented by a LISP program that could use basic logical and arithmetic operators in combination with a number of provided "sensor" functions that read various aspects of the board state.

Azaria and Sipper ran the genetic programming algorithm for up to 500 generations (more than 2 million games played) to create a backgammon strategy called *GP-Gammon*. Azaria and Sipper take pains to point out that this is the number of games played by the system as a whole, and therefore the winning strategy itself had only played several thousand games. Nonetheless, this approach, while capable of producing very strong players, is obviously very time-intensive.

4.1.7 FUZZY LOGIC

Heinze et al. (2005) trained a fuzzy controller system (Jang et al. 1996) as a backgammon player. Given a set of numeric inputs derived from board features, the system would map those inputs to membership in a number of fuzzy sets along the lines of "few" versus "many" or "short" versus "long". A set of rules, of the form "if [input] belongs to [set] then board position is [good or bad]," was used to convert the fuzzy set memberships into a number representing the perceived strength of the board position.

[1] It is unclear from the paper whether this is 400,000 games for each team (2,000 generations of the GA) or 400,000 games total (133 generations of the GA).

The *Fuzzeval* controller was trained by playing 100 games against *pubeval*; learning occurred by adjusting the set membership functions for each input based on average value of that input that was observed in games that it won. That is, the membership in sets representing a good board position would be maximized around that average value, and membership in sets representing a bad board position would be minimized around that same value.

Given that Heinze et al. provided the derived board inputs for the learner to use, the sets to classify the inputs into, and the rules used to label sets as good or bad, it is arguable that very little actual machine learning occurred in this approach. In addition, Heinze et al. only provide performance estimates for *FuzzEval* in its untrained and fully trained states, with no information about the intermediate development given. Therefore, some skepticism as to the worth of this research seems warranted, but it is included here for the sake of completeness.

4.1.8 Relational reinforcement learning

Sanner (2005) applied a model-free relational reinforcement learning algorithm with feature attribute augmentation, dubbed FAA-SVRRL, to train a Bayes net as a backgammon player. In essence, the learner, beginning with a Bayes net with one node for each possible value of each of the three derived board features used, learned the worth of each node (based on games won and lost) and combined related or correlated nodes in order to reduce the complexity and improve the quality of the net as a whole, following the minimum description length principle.

As with Sanner's earlier work on *ACT-R-Gammon*, the greatest strength of this approach seems to be the speed with which it learned, requiring more games than *ACT-R-Gammon* but still far less than most other approaches. However, it was a complex approach to the problem of learning backgammon and was dependent upon the availability of intelligently designed board features in order to achieve its success.

4.2 LEARNING ENVIRONMENTS

These past researchers have differed as to the best environment for learning a game playing strategy, specifically whether learning is most effective in the context of learner-versus-learner self-play, learner-versus-expert play, or observation of games played by others without input from the learner.

Tesauro and Sejnowski (1989) used a training set for *Neurogammon* that consisted of a mixture of moves of all sorts — moves taken from expert games, moves from games where an expert played the partially trained network, and moves where the network played itself. However, these moves were not randomly selected, nor did they provide a complete snapshot of a game; rather, the self-play moves used for training were selected specifically because, in the opinion of a human expert, they represented mistakes made by the partially trained network.

Tesauro (1992) compared *TD-Gammon* (trained by TD(λ) and self-play) to neural networks trained by backpropagation on a expert-labeled set of moves (likely descendants of *Neurogammon*); the finding that *TD-Gammon* was able to equal and surpass the performance of the other networks was one of the more remarkable results of this research.

The hill-climbing algorithm in *HC-Gammon* (Pollack and Blair 1998) can be seen as a variant of learner-versus-learner self-play; learner-versus-expert or expert-versus-expert play would make no sense in this context.

Sanner et al. (2000) used a learner-versus-expert approach in which *ACT-R-Gammon* played against *pubeval* to learn the game . No justification was given for this design decision. Qi and Sun (2003) used a learner-versus-learner self-play approach for *GMARLB-Gammon*, again with no justification given.

GP-Gammon learned by the recombination, mutation, and selection of the GP algorithm, rather than through playing the game directly, but the selection operators of the algorithm were based on playing one evolved strategy against another for comparison (Azaria and Sipper 2005), which can be seen as somewhat similar to learning by self-play.

The application of the *FAA-SVRRL* algorithm to backgammon was only described very briefly by Sanner (2005), and it is not clear what sort of learning environment was used for that research.

Wiering et al. (2005) compared learner-versus-learner, learner-versus-expert, and expert-versus-expert approaches for *WPM-3*, with their trained modular network *WPM-9* as the expert player. This seems to have been the only reasonably thorough comparison of these approaches as applied to backgammon thus far; their conclusion was that learner-versus-learner and learner-versus-expert approaches produced nearly indistinguishable results, while observation of expert-versus-expert play was only slightly inferior.

4.3 Board representations, or input encodings

4.3.1 Objective and subjective representations

Some board representations used have been objective, that is, representing the board state as seen by a neutral observer at all times (Tesauro 1992, 2002), but most representations have been subjective, that is, representing the board state as seen by the player whose turn it currently is (Tesauro and Sejnowski 1989; Tesauro 1993; Pollack and Blair 1998; Sanner et al. 2000; Qi and Sun 2003; Azaria and Sipper 2005; Heinze et al. 2005; Sanner 2005; Wiering et al. 2005).

Conceptually, if position x has value k for player 1, then the mirror image of x (swapping piece colors and mirroring positions) should have equal value k for player 2. With a subjective representation, both players would observe the exact same subjective game state, and so this equality would be automatic, but to an objective board representation, the two positions may appear completely different and therefore must be learned separately, requiring additional training. This is probably why most research has used subjective board representations.

However, the downside of a subjective representation is that to a learner observing both sides of the game, the game state would appear to vary wildly from one move to the next as the perspective of the board representation changed, which would presumably cause difficulties for a learning algorithm, such as TD(λ), that bases its learning in part on the change from one observation to the

next. Wiering et al. (2005) managed to work around this issue by modifying TD(λ) to operate on the assumption that the predicted outcome would invert at each step instead of remaining constant.

4.3.2 Truncated unary board encoding

TD-Gammon

The initial version of *TD-Gammon* used a truncated partial unary encoding of the objective board state (Tesauro 1992), and much research since has followed at least partly in its footsteps in this regard (Pollack and Blair 1998; Tesauro 2002; Qi and Sun 2003; Azaria and Sipper 2005). In Tesauro's representation, for each space on the main board and for each player, three unary inputs (each with value 0 or 1) represented the absence or presence of at least 1, 2, or 3 of that player's pieces on that space, and a fourth integer input represented the number of the player's pieces present if greater than three.

Although Tesauro (2002) described this representation as "'knowledge-free' in the sense that no knowledge of expert concepts or strategies was built in," it is worth noting that in backgammon, having one, two, or three pieces on a space all have significantly different strategic value, while there are rapidly diminishing returns as a result of having any more than three pieces on a space.

Furthermore, for each player, the number of pieces on the bar and borne off were represented by a single integer each. Once again, "knowledge-free" this representation may be, but the absence of a unary encoding of these values "coincidentally" parallels the fact that for a player, more pieces on the bar is bad, and more pieces borne off is good, but there is no special strategic significance to having a particular number of pieces in either situation.

Finally, two units represented which player's turn it was to play, presumably by being set to 1 for the player whose turn it was and 0 for the other player. This gave a total of 198 distinct elements in the board representation, that is to say, 198 inputs to the *TD-Gammon* neural network.

OTHER PROGRAMS

Tesauro's benchmark backgammon player *pubeval* used an asymmetric encoding of the subjective board state, with four unary inputs and one numeric input per board space, corresponding to one enemy piece, one player piece, two or more player pieces, exactly three player pieces, and the number of player pieces greater than three present (Tesauro 1993). Two additional numeric inputs indicated the number of enemy pieces on the bar and the number of player pieces borne off (but not the enemy's pieces borne off or player pieces on the bar), for a total of 122 inputs.

HC-Gammon used a subjective representation based on *TD-Gammon*'s, omitting the two turn indicator inputs (unnecessary for a subjective board representation) and adding a binary input representing whether or not the game had entered the endgame "race" stage, for a total of 197 inputs (Pollack and Blair 1998).

GMARLB-Gammon used a subjective truncated unary encoding as well (Qi and Sun 2003). The representation was asymmetric, as while the opponent's pieces were represented using Tesauro's encoding, the player's pieces were represented by a custom encoding that apparently used five inputs per piece instead of assigning inputs to each space (Qi and Sun are somewhat unclear on this point). Their representation also included 12 inputs for the die roll (one binary input per possible value of each die) and an additional 16 units (one per piece plus one "no piece" input) apparently indicating which piece (if any) had been moved so far this turn by the player.

GP-Gammon also used a customized subjective truncated unary encoding (Azaria and Sipper 2005). Their representation was more truncated than Tesauro's, having only two unary inputs per space per player (representing 1 or more and 2 or more pieces on a space) instead of three as in Tesauro's work. Second, the input representing "more than two pieces present" only existed for pieces belonging to the player currently acting, and not for pieces belonging to the opponent. In addition to this encoding, the board representation for *GP-Gammon* included several derived features, described in the next section.

4.3.3 DERIVED FEATURES

In *Neurogammon*, Tesauro and Sejnowski (1989) used a set of eight derived board features to facilitate learning and make up for observed weaknesses of the trained program. A form of these same features was also added to a later version of *TD-Gammon* to enhance its performance (Tesauro 2002). The features not been precisely defined, but can be approximately summarized as follows:

1. Pip count, the total distance a player's pieces must move in order to reach their goal of leaving the board.

2. Degree of contact, the extent to which each player's pieces can interact with the other's.

3. Number of spaces occupied in the player's home board.

4. Number of spaces occupied in the opponent's home board.

5. Number of pieces in the opponent's home board.

6. Presence of a "prime;" blockading 6 consecutive spaces forms a prime, which is impossible for the other player to move pieces past.

7. Blot exposure, the probability that any given lone piece on a space can be hit by an opponent's piece on the opponent's next move.

8. Strength of any blockades present, the likelihood that a piece trapped behind a blockade by the opponent will be able to escape from it.

Features 1, 2, 3, 4, and 5 are simple linear functions of the board state, feature 6 represents the presence of a complex structural feature, and features 7 and 8 represent probabilities of certain events based on the overall board structure. It is worth noting that Tesauro (2002) argued that the use of hand-designed, derived board features in *Neurogammon* was to some extent:

> "...an effort to [cover] up the flaws of existing learning algorithms ...the ultimate goal ...should be to develop better learning algorithms that have no such flaws in

> the first place. . . . The supervised learning procedure used in *Neurogammon* was seriously flawed . . . Much effort was expended to try to compensate for these deficiencies through clever feature design. . . . [T]he vastly superior TD learning method was found to have no such deficiencies . . . Several of the features in the *Neurogammon* feature set could probably be deleted from *TD-Gammon* without harming its performance."

Nonetheless, *TD-Gammon* continued to use those derived features, and several other researchers have followed this lead.

In addition to its previously described unary encoding of the raw board position, *GP-Gammon* included derived features for the pip count of each player, the blot exposure of the current player, and the blockade strength of each player (Azaria and Sipper 2005).

Both *ACT-R-Gammon* and *FAA-SVRRL* used no explicit board representation, instead using three types of derived features (Sanner et al. 2000; Sanner 2005):

1. Position of enemy pieces captured as a result of a move.

2. Position of player pieces exposed (as lone pieces) after a move, and the number of enemy pieces that could potentially attack each one.

3. Position of player blockades existing after a move, their size, and the number of enemy pieces potentially trapped behind them.

Fuzzeval used a set of 15 derived features, which have not been completely described but apparently included features similar to those used in *Neurogammon* (Heinze et al. 2005).

4.4 Decision making methods, or output encodings

The mapping of learner outputs to game playing decisions has not been as clearly described in previous work as the mapping of game states to learner inputs, but here we endeavor to describe past approaches to the best of our understanding. Most researchers seem to have agreed that it is far more straightforward to train a learner to select the best move given a list of all possible legal

moves than it is to train it to propose a legal move on its own; they have, however, disagreed as to how to represent that move selection.

HC-Gammon and *Neurogammon* each had a single network output which corresponded to the perceived value of a given move or board position to the player making it (Tesauro and Sejnowski 1989; Pollack and Blair 1998). Although *GP-Gammon*, *Fuzzeval*, and *FAA-SVRRL* used vastly different learner representations, they too produced single output values of similar significance (Azaria and Sipper 2005; Heinze et al. 2005; Sanner 2005). All of these learners thus played the game by simply selecting out of all possible moves the move with the highest perceived value.

ACT-R-Gammon also produced a single output value, but as part of the ongoing goal of that project to model human cognitive processes, applied some amount of stochastic noise to these values before using them to to select the "best" move in any given situation (Sanner et al. 2000).

TD-Gammon used a slightly different output representation with four outputs, corresponding to four possible projected outcomes of the game: a win by white, a win by black, and "gammons" (wins by a large margin, worth two points) by each side (Tesauro 2002).[2] These four outputs were used together to select the preferred move for the appropriate player, though it is not entirely clear how they were thus combined.

Wiering et al. (2005) used a variant output encoding to try and maximize the overall learning. In addition to one output representing the overall value of a position in the range -3 to $+3$, there were six additional outputs representing the probability of each possible outcome of the game. The weighted sum of the six outputs was averaged with the value of the single output in order to obtain an overall predicted quality for any given position. However, Wiering et al. found that the sum of six and the single output always closely agreed, suggesting that this method was perhaps redundant.

The neural networks used in *GMARLB-Gammon* had 16 outputs, one per piece controlled by a player and one additional output meaning "no piece" (Qi and Sun 2003). The network would

[2]There is also the possibility of a "backgammon," a total rout worth three points, but this is very rare even in unskilled play, so Tesauro decided it was not worth including in the representation. Presumably backgammons were therefore counted as gammons while training the network.

Table 4.1: Training required and resulting performance of various backgammon programs.

Learner	Training	Best average performance
Neurogammon	3202 moves × 20 cycles	59% vs. *Gammontool*
TD-Gammon, 1992	200,000 games	66.2% vs. *Gammontool*
pubeval	unknown	57% vs. *Gammontool*
HC-Gammon	200,000–800,000 games	40% vs. *pubeval*
ACT-R-Gammon	1,000 games	45.23% vs. *pubeval*
TD-Gammon, 2002	6,000,000+ games	not provided
GMARLB-Gammon	400,000 games	51.2% vs. *pubeval*
GP-Gammon	500,000–2,000,000 games	56.8% vs. *pubeval*
Fuzzeval	100 games	42% vs. *pubeval*
Fuzzeval-TDNN	400,000 games	59% vs. *pubeval*
FAA-SVRRL	5,000 games	51.2% vs. *pubeval*
WPM-9	1,000,000+ games	not provided
WPM-3	200,000 games	51% vs. *WPM-9*

presumably opt to move whichever piece received the highest ranking. It is not clearly stated how the system dealt with a proposal to move a piece that could not in fact be legally moved (or for that matter an attempt to pass when a move was required), but it seems reasonable to assume that it would simply fall back to the highest-rated legal move.

4.5 Performance comparison of various approaches

Table 4.1 compares the amount of learning required by these various approaches and the resulting performance levels that were obtained. Most of Tesauro's work compared his players against a program called *Gammontool*, while most backgammon research since has focused on comparison to Tesauro's freely available program *pubeval* (Tesauro 1993). While it is not entirely clear how winnings versus each program compare, based on the performance of *pubeval* itself against *Gammontool*, it can at least be reasonably argued that *Neurogammon* would win slightly more than 50% against *pubeval* and the 1992 version of *TD-Gammon* was clearly significantly superior to *pubeval*.

Table 4.2: Comparison of neural networks used in various past research.

Learner	Inputs	Hidden nodes	Outputs
Neurogammon	393–459	0, 12, or 24; 1 or 2 layers	1
TD-Gammon, 1992	198	0, 10, 20, or 40	4
pubeval	122	0	2
HC-Gammon	197	20	1
TD-Gammon, 2002	approx. 300	160	4
GMARLB-Gammon	201	40 / 16	16
Fuzzeval-TDNN	196	80×2	5×2
WPM-9	68 / 277 / 393×7	20×2 / 40×7	7×9
WPM-3	68 / 277 / 393	20 / 20 / (40 or 80)	7×3

It should be noted that since past research has differed not only in the machine learning approach applied but also in the board representation and other parameters (such as neural network structure) used with each approach, a direct comparison of the results obtained may not be truly reflective of the difference between the machine learning methods alone.

Even if one considers only the NN-based approaches, the different input and output encodings and variety of hidden layers have resulted in a wide range of neural network structures used. Table 4.2 compares the structures used by the NN-based approaches previously discussed, insofar as such information is available.

Two notes are worth making about the contents of Table 4.2. First, *pubeval* is actually a pair of linear evaluation functions, only one of which is used to evaluate any given position. Conceptually, this could be viewed as a linear-weighted neural network with two outputs and a helper program used to decide which of these two outputs is relevant, so we have decided to present it as such here for purposes of comparison. Second, in addition to being vague about the specific representational features used in *Neurogammon* and later versions of *TD-Gammon*, Tesauro did not clearly state the number of inputs that were added to the *TD-Gammon* network to accomodate those features. The

given number for the inputs of the 2002 version of *TD-Gammon* is an inference based on Tesauro's statement that the network contains a total of "about 50,000 . . . weights" (Tesauro 2002).

CHAPTER 5

RESEARCH APPROACH

In this research, we have used TD(λ) to train neural networks to play backgammon, hypergammon, pachisi, and Parcheesi. A primary goal was to determine what generally works well for games of this type rather than what works optimally for each specific game, and therefore we have endeavored to keep the approach as generic as possible, with minimal tailoring for each specific game.

5.1 NEURAL NETWORK DESIGN

To save training time and allow for a wide range of experiments, we opted to use relatively small networks with only 10 hidden units each. A greater number of hidden units would presumably allow for increased performance at the cost of increased training time; however, we expected that 10 hidden nodes would be enough to allow for significant learning and clear differentiation between the effectiveness of various approaches. All network weights were initially selected randomly in the range $[-1.0, 1.0]$.

Since this research included games with more than two players, one network output would not suffice to clearly describe all possible outcomes of a given game or board state. Therefore, each network was given one output per player (thus, 2 for backgammon and hypergammon, and 4 for pachisi and Parcheesi), corresponding to the value of that state to each player. At the end of a game, each output was given a value of 0.1 if the indicated player lost the game and 0.9 if the player won. These values were chosen instead of 0.0 and 1.0, with the justification being that the sigmoid activation function of the network output units can only approach 0.0 and 1.0 asymptotically, and so using 0.1 and 0.9 instead would help avoid overfitting in the network training. For

backgammon and hypergammon, the value of 0.9 was reserved for the rare "backgammon" win condition; ordinary wins and "gammon" wins were assigned values of 0.6 and 0.75 respectively.

5.2 TD(λ) PARAMETERS AND APPLICATION

In selecting the parameters for TD(λ), we elected to follow the lead of Tesauro's early work (Tesauro 1992) by setting $\lambda = 0.7$ as the exponential weighting factor. To save on training time, however, we opted to use $\alpha = 0.3$ as the learning rate instead of the value of 0.1 used by Tesauro. Although this higher learning rate might potentially lead to overfitting or oscillating performance by adapting the network in response to improbable events, we did not generally find this to be the case.

Each network was trained for 50,000 games (the number of games that Tesauro (1992) found to be adequate for a 10-hidden-unit backgammon network), applying TD(λ) after each move. For backgammon, we elected to train for 100,000 games instead of 50,000, in the hopes that any additional improvement resulting from such additional training might allow our networks to compare that much more favorably to the many other extant backgammon programs.

Even with such simple networks and such relatively brief training time, it was possible to observe substantial learning and to clearly distinguish between the effectiveness of various approaches.

5.3 LEARNING ENVIRONMENTS

In order to apply TD(λ) to develop a game-playing strategy, it is necessary to have a body of games played (and their component board positions) to learn from. Since Wiering et al. (2005) found that allowing the learner to play a role in the generation of these games was generally more successful for backgammon than using games generated by other players alone, we elected to focus only on learner-involved learning environments in this research.

Given that one of the players involved in developing this body of games was to be the learner itself, the question was then what its opponents should be. The two obvious answers to this question

are “also the learner, playing all sides of the game at once” and “some other player.” The latter answer leads to another question, “which other player?”

The obvious difficulty of learning paradigms that involve an existing “expert” player is that an “expert” player is not necessarily an optimal player. As Tesauro (2002) put it, “when doing knowledge engineering of human expert judgement, some of the expertise being emulated may be erroneous.” The learner may learn how to play like this suboptimal player, making the same mistakes as the “expert”, or may learn a style of play that performs well against this particular opponent but performs poorly in general.

Conversely, there is a certain purity to the idea of learning by self-play. Ideally, as the learner learns, it will discover the flaws in its own playing style and seek to exploit them when playing against itself, creating pressure to revise its playing style to eliminate those flaws. However, there is also the danger that the learner will train itself into a local optimum, learning a playing style that is effective against itself but inferior against other opponents. Conceptually, the randomness present in the games studied here will help to prevent this, but it is not obvious that this is the case.

Therefore, we performed a number of experiments to compare the effectiveness of these sorts of learning approaches or learning environments. We compared learning by self-play to learning by playing against a skilled opponent and learning by playing against an *unskilled* opponent. We argue that while it is of course not feasible to examine *every* possible opponent player, this covers the two extremes, providing a reasonable sampling of possible outcomes. For pachisi and Parcheesi, which have four players instead of two, some intermediate learning environments (combining self-play and play against another opponent) were also possible, and were also considered.

5.3.1 Unskilled opponent

To serve as an unskilled opponent, we implemented a random playing strategy, *RandomPlayer*, which simply chooses randomly among all legal moves in a given position with equal probability. This *RandomPlayer* was therefore capable of playing any board game with equal (poor) skill, and so was used as the unskilled opponent for the experiments with all four games.

5.3.2 SKILLED OPPONENTS

For a skilled backgammon opponent, we selected the program *pubeval*, which has been widely used as an opponent in past research precisely because of its relatively high level of skill (Pollack and Blair 1998; Sanner et al. 2000; Qi and Sun 2003; Azaria and Sipper 2005, et al.). This program is a pair of linear evaluation functions, one for positions in the "contact" stage of the game, and one for the "race" endgame. Distinguishing between these two situations is trivial, and so *pubeval* applies the appropriate set of function weights to its 122-variable input encoding, previously described, in order to calculate a perceived value for any given board position.

Since hypergammon is essentially a much-simplified subset of backgammon, we decided that *pubeval* could be expected to perform reasonably well at hypergammon, or at any rate well enough to be used for the same purpose for that game as well.

Since there has not been the same level of study of pachisi and Parcheesi, no standard benchmark player exists. We designed and implemented some fairly simple heuristic players, called *HeuristicPachisiPlayer* and *HeuristicParcheesiPlayer* respectively, to fill this role.

HeuristicPachisiPlayer considers all possible moves from a given state and selects the move that, in order of decreasing priority:

1. Advances pieces closest to the goal (forcing the player to move rather than pass if possible)

2. Keeps the greatest number of pieces safe (a "safe" piece is one that is in the entry or return path, on a castle space, or home, but not remaining at start)

3. Keeps pieces most closely grouped together (minimizing the distance between the most and least advanced pieces).

If multiple possible moves are of equal value under this heuristic, the move first observed is the one selected.

HeuristicParcheesiPlayer is a similar heuristic for Parcheesi, selecting the set of moves that, in order of decreasing priority:

1. Advances pieces closest to the goal (encouraging moves that hit enemy pieces or bring pieces home, since both result in additional bonus moves that lead to further advancement)

2. Maximizes the number of safe pieces (a "safe" piece is one that is on the home path, at home, or on a castle space, but not remaining at start or forming a blockade on an otherwise vulnerable space)

As above, if multiple options are of equal value, the first one encountered will be selected by this player.

5.4 Board representations

The modification of TD(λ) by Wiering et al. (learning based on the expected outcome reversing after each move, allowing TD(λ) to be used with a subjective board representation) is obviously only applicable to two-player games, so it would not be of use for pachisi or Parcheesi. Since part of the intent of this research is to look for commonalities among the four games being studied, we have chosen to follow the lead of Tesauro (1992, 2002) and use standard TD(λ) with an objective board representation for this work.

Three different board representations were considered: a simple unary encoding of the board state, patterned after the representation used in early versions of *TD-Gammon* (Tesauro 1992), a "smart" representation consisting of derived board features that were relevant to all four games, and a hybrid representation combining both of these.

5.4.1 Unary representations

For backgammon and hypergammon, the truncated unary board representation we selected was nearly identical to that of Tesauro (1992) (as previously described in section 4.3.2), save that it omitted the two turn indicator units, thus having a total of 196 inputs. We expected that while this omission obviously would weaken the maximal performance of the player, since knowing who is to play next is an important piece of strategic information in evaluating a board position, it is of

less importance in a racing game like backgammon than in a highly positional game like chess. This omission also would force the learner to focus more on positional features that are of value regardless of whose turn it is, which may in fact be beneficial.

We used a similar board representation for the investigations into pachisi and Parcheesi.

In pachisi, a player can potentially place all four of his pieces on a single space, but there is no strategic benefit to doing so. Furthermore, the number of possible positions for each piece is much higher than in backgammon (83 "active" spaces versus 24). Therefore, we opted to truncate the unary representation to two unary inputs per space per player, representing 1 or more and 2 or more pieces present respectively, with no additional numeric input to quantify the number of additional pieces present. Combined with the possibility of up to four pieces at start or at finish for each player, the pachisi representation had $((83 \times 2 + 4 + 4) \times 4 =)$ 696 inputs.

In Parcheesi, no more than two pieces may share a space except for the start and finish spaces, so it is unnecessary to truncate the unary representation. Since each player may have up to 2 pieces on any of 75 active spaces or up to four at the start or finish, the unary Parcheesi representation produced $((75 \times 2 + 4 + 4) \times 4 =)$ 632 inputs.

5.4.2 Derived features

After considering the derived board features that had been used with backgammon (Tesauro and Sejnowski 1989; Azaria and Sipper 2005), we selected eight derived "smart" features that we expected to be relevant to all four games being studied. These features were defined as follows:

1. Turn indicators: one binary value per player, with that of the next player to move being set to 1.

2. Pip count: for each player, the total number of spaces that player's pieces would need to move in order to win the game.

3. Worst piece: for each player, the distance of its least advanced piece from the goal.

4. Best piece: for each player, the distance of its most advanced piece from the goal.

5. Contact: for each player, the sum of the number of enemy pieces each of its own pieces might potentially interact with on its way to the goal. For example, if a player had two pieces, and one must pass three enemy pieces to reach the goal, while the other must pass one enemy piece, this feature would have a value of four.

6. Exposure: for each player, the number of pieces that are potentially vulnerable to being attacked, independent of the ability of the enemy to actually endanger them.

7. Hit probability: for each player, the probability that any enemy might, on their next turn, make a roll that would allow them to hit one of the player's vulnerable pieces.

8. Blockade probability: for each player, the probability that they would make a roll which they are unable to use to make a complete legal move from the current position.

These features include simple linear functions of the board state (2, 3, 4, 5, and 6), more complex probabilistic attributes (7 and 8), and a game feature that is relevant but could not be determined by simply looking at the board (1).

Features 1, 2, 5, 7, and 8 are a direct imitation of features from *Neurogammon* and *GP-Gammon*. Features 3 and 4 are a generalization of the "pieces in home board" features present in *Neurogammon*; since positional considerations in pachisi and Parcheesi are quite different from those in backgammon, it is hoped that this interpretation of those features provides a more general view of the strength of each player's position.

Feature 6 is an alternate expression of feature 7, with the intent being that a player should probably learn that vulnerable pieces are a risk even if the opponent is not immediately in position to threaten them, as they might soon move into position to do so and the player might not then make a roll allowing them to move the pieces to safety at that point.

No complex positional feature analogous to *Neurogammon*'s "prime" feature is present in this feature set, since there is no one such feature that would be relevant to all four games.

Clearly, the learner must develop some contextual knowledge in order to make the most use of these features, especially the turn indicators. Being in a vulnerable position at the beginning of

one's turn is quite different than being in a vulnerable position at the beginning of one's opponent's turn! However, to minimize the observed change in features between successive board positions, all of the above features were calculated and made available with every board position, even those features which were not relevant at that time, such as the probability of a player being hit by an opponent when it was not the opponent's turn.

5.4.3 Game-specific details of derived features

In backgammon and hypergammon, since the players' pieces move in opposite directions, the contact value for each player is always identical: three black pieces needing to pass one white piece is the same as one white piece needing to pass three black pieces. Therefore, only one value for this feature was needed, and so the derived feature representation for backgammon and hypergammon consisted of 15 values. Since there are 36 possible die rolls in these games (ignoring the symmetry of, for example, a roll of (5,6) and a roll of (6,5)), each occurring with equal frequency, the hit probability and blockade probability each corresponded to the number of such rolls that could result in a hit or a block, and were easily computed.

In pachisi, the contact feature included, but did not distinguish between, pieces belonging to both players on the opposing team. Similarly, the hit probability feature combines the probability of being hit by either opponent, so that if (for example) one opponent had a 50% chance of being able to hit a player's vulnerable pieces and the other opponent had a 0% chance, a single probability of 25% would be given for this value. Thus, each player received one value for each feature, for a total of 32 values.

There are only seven different possible rolls in pachisi, and each results in only a single piece being moved, so the probabilistic features were easily calculated, but it is important to remember that these rolls fall on a bell curve and do not occur with equal probability — a roll of 0 or 6 occurs only $1/20^{\text{th}}$ as often as a roll of 3, so the values of these derived features were established accordingly.

In Parcheesi, there is no team play, so the contact and hit probability features each combined the interaction of each player with all three of his opponents. Parcheesi's "smart" feature representation, like that of pachisi, thus contained 32 values.

Because of the possibility of "bonus" moves in Parcheesi (awarded as a result of landing on an enemy piece or bringing a piece home) the derived probabilistic features are quite complex to compute for that game. It is possible (albeit unlikely) that a player might make a roll that could not hit a particular opponent directly, but could in fact use that roll to hit a different opponent, then use the 10-space bonus move from that to bring a piece home, then use the 20-space bonus move from that in order to hit the opponent in question. Moves of this complexity were deemed too computationally expensive to consider in the construction of the probabilistic features (recall that to construct the "smart" feature representation, every possible move of every player with every possible roll must be considered) and so instead two simplifying approximations were made for Parcheesi's smart features:

1. Bonus moves were not considered in the calculation of either probability — the hit probability did not include hits made possible only by using a bonus move, and the blockade probability did not consider the inability to use a bonus move as being blocked.

2. The special rules surrounding a roll of doubles (if all of a player's pieces are out of the start area and the player rolls doubles, the player may use the opposite sides of the dice as well, but *must* use all four such moves or else cannot move at all) were disregarded in the calculation of the blockade probability; instead, doubles were treated as normal rolls in this case.

5.4.4 COMBINED FEATURES

The combination of the unary board representation and the derived features was accomplished by simply appending one set of numbers to the other. Thus, for backgammon and hypergammon, the combined feature set consisted of 211 values, for pachisi, 728 values, and for Parcheesi, 664

Table 5.1: Size of (number of values in) each board representation for each game.

Game	Unary representation	"Smart" features	Combined representation
Backgammon	196	15	211
Hypergammon	196	15	211
Pachisi	696	32	728
Parcheesi	632	32	664

values. The size of all three types of representations for each game are collected in Table 5.1 for easy reference and comparison.

5.5 Experimental design

Our experiments thus would investigate the effect of two variables on the quality of the developed players: the learning environment used and the input representation selected. We hypothesized, based on the results that others (especially Tesauro (1992, 2002); Wiering et al. (2005)) obtained for backgammon, that self-play and play-versus-skilled learning environments would both work well, but play-versus-unskilled learning would be markedly inferior. We also hypothesized that the combination of unary and derived board representations would, logically enough, produce players stronger than either representation alone.

Due to time constraints, rather than perform a full cross-comparison of all possible combinations of these two variables, when the learning environment was to serve as an experimental variable, we used only the unary input encoding, and when studying the effect of the board representation, we used only the self-play environment.

To provide some certainty that any difference in our observed results were due to our experimental variables and not mere chance, each experiment (game plus learning environment plus board representation) was repeated five times, starting with different random network weights each time.

As learning proceeded (again, a total of 50,000 games per experiment for hypergammon, pachisi, and Parcheesi, or 100,000 games per experiment for backgammon), the current network weights were saved to disk at regular intervals. After the training was complete, each saved network was loaded in turn and used to play against a benchmark opponent in order to evaluate its strength.

Since these are nondeterministic games, we could not simply assume that the winner of a single match would necessarily be the stronger player; to do so would be to ignore the significant impact of chance upon the results of these games. A somewhat larger number of matches must be played in order to truly understand the relative strength of two players. In order to determine a reasonable value for this number, an experiment was performed.

In addition to the previously described *RandomPlayer*, we implemented a second simple strategy, *DeterministicPlayer*, which always picked the first possible move presented to it. A program was implemented to have four identical *RandomPlayer*s play Parcheesi repeatedly, tracking the total number of games won by each player after each game played. The same experiment was then repeated for four *DeterministicPlayer*s.

Since all four players in each experiment were identical, the difference between the highest and lowest winning rates after each game can be taken as an estimate of the error, or influence of chance on the winning rates. The results of both experiments in this regard are shown in Figure 5.1. For the *DeterministicPlayer* experiment, the error appears to have reached a near-constant value of slightly under 2% after about 2,500 games played, and did not further decrease after that. For the *RandomPlayer* experiment, the error decreased continually to the end of the experiment, dropping below 3% after about 5,000 games and below 2% after about 8,000.

We argue that the *RandomPlayer* results represent a rough upper bound on the error, as noise was present both in the game itself and in the selection of each move as well. We argue that the *DeterministicPlayer* results are a more realistic error estimate for our trained networks, as while *DeterministicPlayer* is in no way a *good* player, its move selection in any given situation, like that made by the trained networks, was at least self-consistent.

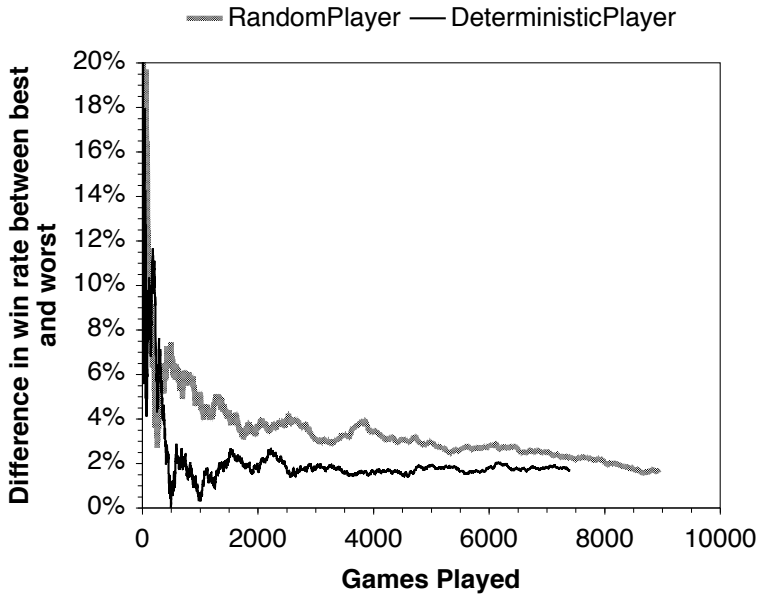

Figure 5.1: Error estimate in calculated winning rates of two Parcheesi players.

Furthermore, we argue that since backgammon and hypergammon each have only two players, compared to the four of Parcheesi, and therefore have a smaller number of possible outcomes to each game played, the error estimates of Figure 5.1 also represent a reasonable upper bound on the error in performance estimates for these other games. Since pachisi has fewer different moves than Parcheesi (7 possible throws of six cowries as opposed to 21 possible rolls of a pair of dice) we argue that these results also represent an upper bound on the error in pachisi performance estimates.

After considering these results, we determined that we would compare the strength of any two players by having them play one another for 5,000 games. Given the above arguments, we claim that this approach should yield a measure of their relative strength accurate to within 3% for each of the four games being considered.

Thus, each trained network in turn had its performance evaluated by using it to play 5,000 games against the appropriate "skilled" player for the relevant game. Thus, for backgammon and hypergammon, the networks played against *pubeval*, for pachisi, two copies of the network being tested (acting as a team) played against a team of *HeuristicPachisiPlayer*s, and for Parcheesi, the network played against three *HeuristicParcheesiPlayer*s. The percentage of games won by each network would serve as a measure of its strength; for this purpose, for backgammon and hypergammon, no distinction was drawn between ordinary wins, "gammon" wins (normally worth two ordinary wins) or "backgammon" wins (normally worth three ordinary wins).

5.6 Software design and implementation

All software used in this research was designed and implemented using the Java 2 Platform Standard Edition 5.0 (J2SE 5.0). The class structure used abstract classes, composition, and polymorphism in order to make modifications and extensions of the code more feasible (Figure 5.2). A game to be played was broken down into three abstract components:

- One *Board*, representing the current state of the game board (piece locations, the current player, the most recent roll of the dice) and the rules associated with that state (whether any player had won yet, which moves were legal for a given player in that state).
- A number of *Player*s, each with internal logic used to select a move in any given situation.
- One *Game* to handle actually playing the game until it ended, including rolling the dice or cowries, telling a particular *Player* that it was their turn to move, and handling special cases independent of the raw board state, such as the "no more than three doubles in a row" rule of Parcheesi.

Subclasses of *Game* and *Board* were implemented for each of the four games tested; *BackgammonBoard* was implemented as a subclass of *HypergammonBoard* and *BackgammonGame* as a subclass of *HypergammonGame* since the only differences between the two boards and games are the number of pieces per player and the initial setup of the board.

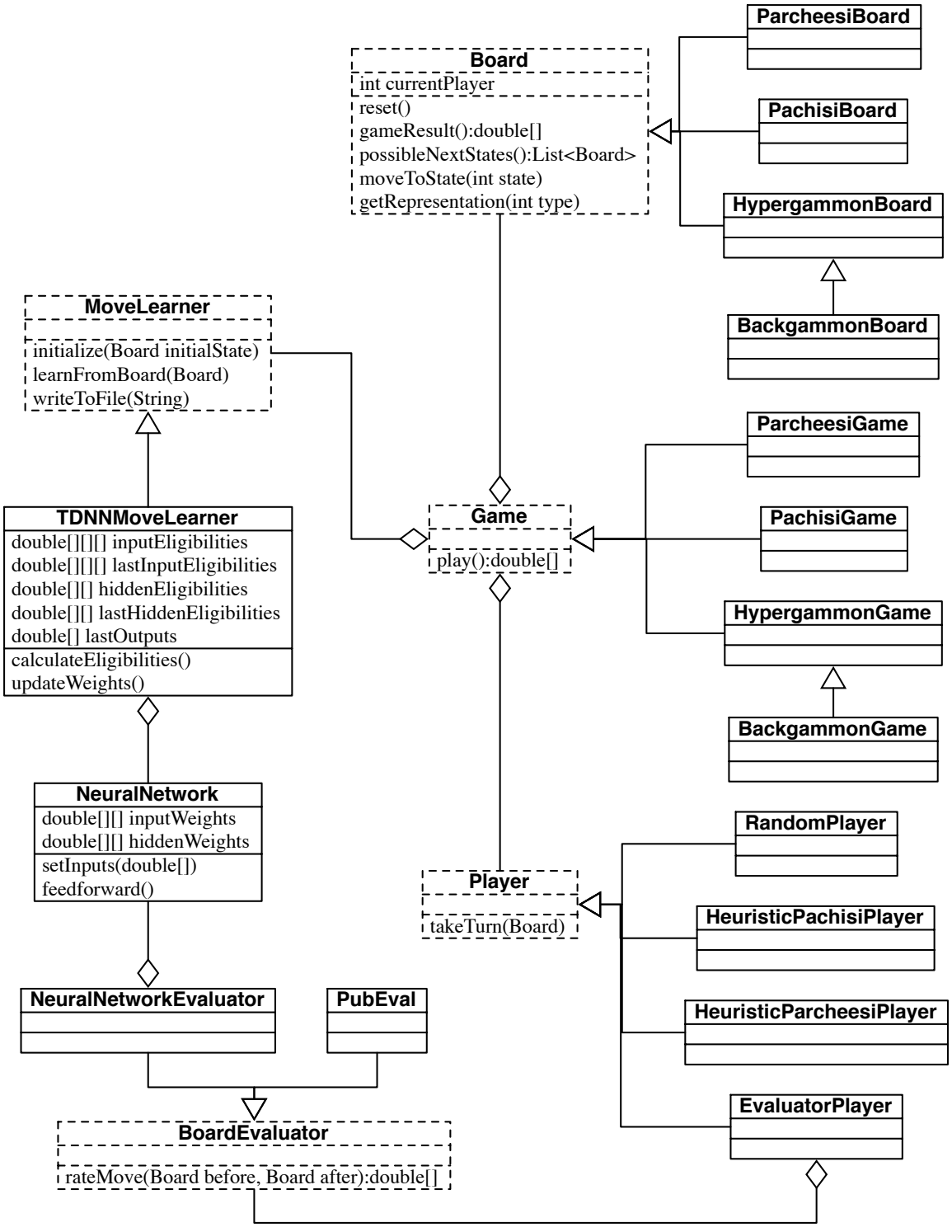

Figure 5.2: Overview of the class structure used for this research.

In order to use neural networks (or other knowledge representations) as learners and players, two additional abstract classes were defined:

- *BoardEvaluator*, for any code that could look at a *Board* and return a set of values representing its worth for each player in the game. An *EvaluatorPlayer* is a type of *Player* that simply feeds the board state resulting from every possible move into a *BoardEvaluator* and selects the move that was most highly rated by the evaluator.

- *MoveLearner*, for any machine learner that could learn in some fashion by looking at a succession of *Board*s.

The code structure reflects the distinction between the neural network (a *NeuralNetwork*), the TD(λ) algorithm used to train it (*TDNNMoveLearner*), and the use of such a network as a game board evaluator (*NeuralNetworkEvaluator*). This made it possible to easily use a trained network as a game player without further learning (for evaluation of its play strength) or to train a network even when it was not directly in control of the game (such as learning from its opponent's moves when using a learning environment other than self-play). Also, with this structure, future research could conceivably use another learning method (or even a mixture of learning methods) to train the same network.

Two controller applications were implemented in order to actually perform the necessary training and evaluation: *PlayGames* and *EvaluatePlayers*.

PlayGames was used for learning; given a *Game*, a set of *Player*s, and one or more *MoveLearner*s, it would play a number of games, writing the current state of the *MoveLearner* (in this case, the weights of the *NeuralNetwork*) to disk at specified intervals.

EvaluatePlayers was used for evaluation of these stored learners; given a *Game*, a path to load the learners (the *NeuralNetwork*s) from, and a set of opponent *Player*s, it would load one such network from disk, and using it with a *BoardEvaluator* and *EvaluatorPlayer*, play it against the specified opponents for a number of games (here, 5,000), and output the number of games won by this network, then repeat for each successive stored network.

5.7 SOFTWARE EXECUTION

Taking advantage of Java's broad cross-platform compatibility, these programs were run on whatever systems became available as experimentation proceeded, including a dual-processor Apple PowerMac G4 workstation and an Apple PowerBook G4 laptop running Mac OS X 10.4, two Dell Pentium 4 workstations running Windows XP, a 4-processor Xeon server running Windows Server 2003, and fourteen Sun Blade workstations running Solaris 5.8.

A single 50,000-game training or evaluation run took anywhere from several hours to over a week, depending on the system in use (the G4 workstation was nearly obsolete and quite slow, while the Xeon server was much faster), the game being studied (hypergammon was quite fast, backgammon and pachisi somewhat slower, and Parcheesi very slow), and the network in use (the smart-feature networks had fewer inputs and so fewer weights to repeatedly update than the unary or combined-feature networks).

CHAPTER 6

INFLUENCE OF THE LEARNING ENVIRONMENT

As previously stated, for each of the four games being studied, we ran experiments for a number of different learning environments, or sources of the game positions being used as training, generally corresponding to self-play by the learner, play versus a skilled opponent (*pubeval*, *HeuristiPachisiPlayer*, or *HeuristicParcheesiPlayer*), or play versus an unskilled opponent (*RandomPlayer*). Each experiment was repeated five times in order to establish some certainty that differences in the results from one experiment to the next were not due to chance alone.

6.1 BACKGAMMON

For backgammon, we compared three learning environments for TD(λ): learner versus self (Figure 6.1), learner versus *RandomPlayer* (Figure 6.2), and learner versus *pubeval* (Figure 6.3). The self-play and play-versus-*pubeval* learners performed similarly, achieving best performances of 48.6% and 48.1% wins respectively versus *pubeval*, while the play-versus-*RandomPlayer* learner performed much more poorly, achieving a best performance of only 7.3% wins versus *pubeval*.

A direct comparison of these results (Figure 6.4) illustrates that the play-versus-*pubeval* learners demonstrated faster initial learning, achieving for example an average winning rate of 39.2% after 10,000 training games as compared to 27.5% for the self-play learners after the same amount of training. However, the asymptotic performance of both kinds of learner was nearly identical. By contrast, the play-versus-*RandomPlayer* learners also learned very quickly (showing very little further improvement after less than 10,000 games of training) but reached much lower asymptotic average performance.

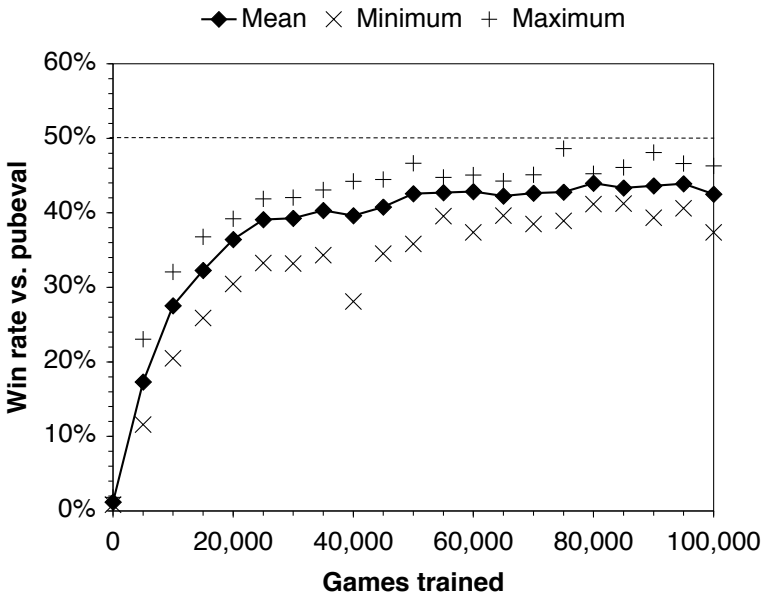

Figure 6.1: Performance of backgammon networks trained by self-play.

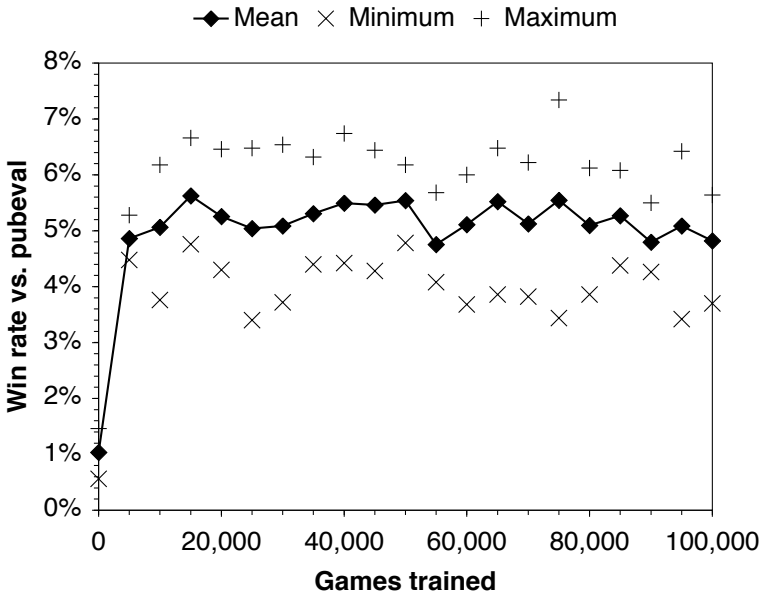

Figure 6.2: Performance of backgammon networks trained by playing against *RandomPlayer*.

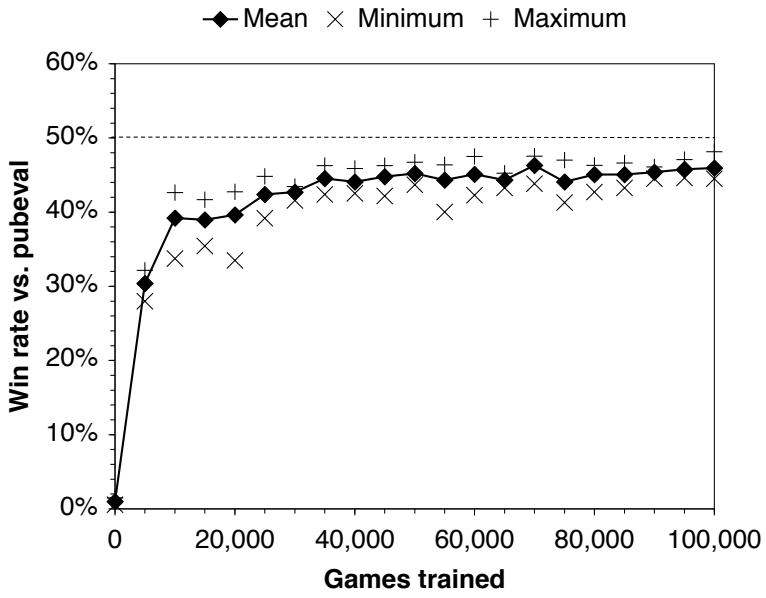

Figure 6.3: Performance of backgammon networks trained by playing against *pubeval*.

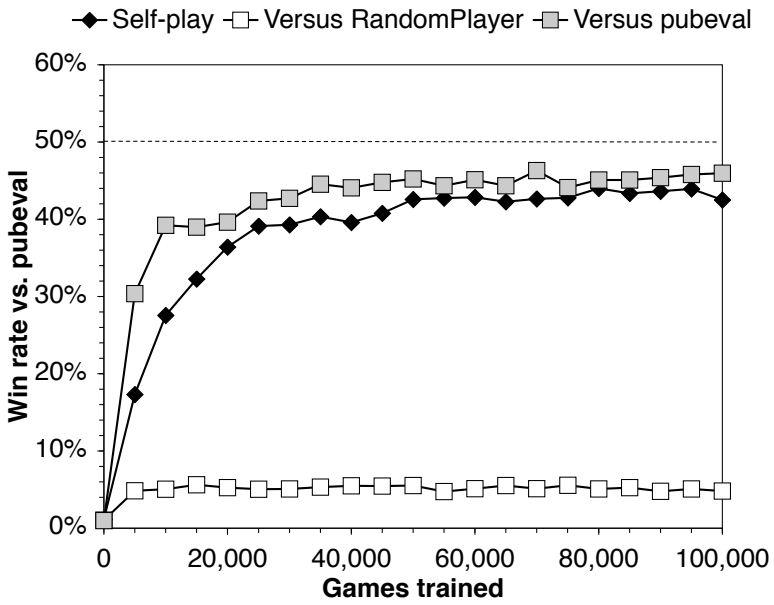

Figure 6.4: Mean performance of backgammon networks produced by each learning environment.

It is worth noting that since *pubeval* was used as the designated opponent for evaluation, it would seem that the networks trained by the play-versus-*pubeval* environment had something of an unfair advantage in that they learned to play well against that specific opponent through their training. One must wonder about the possibility of overfitting in such a case; that is, the possibility that the play-versus-*pubeval* learners might be overly specialized for playing against *pubeval* and would in fact be of inferior skill against other possible opponents.

Given that the play-versus-*pubeval* learners had an unfair advantage in its competition with *pubeval*, it is all the more noteworthy that the self-play learners managed to do very nearly as well against *pubeval* with no such advantage themselves. These learners cannot be accused of being overspecialized to play against *pubeval*, since they never played against it while learning, and so their winning rate against *pubeval* must be assumed to be an honest measure of their overall skill.

6.2 HYPERGAMMON

For hypergammon, we used the same three learning environments as were used for backgammon (Figures 6.5, 6.6, 6.7). The self-play and play-versus-*pubeval* learners performed similarly, achieving best performances of 59.1% and 61.7% wins respectively versus *pubeval*, while the play-versus-*RandomPlayer* learner performed somewhat more poorly, achieving a best performance of 42.6% wins versus *pubeval*.

A direct comparison of these results (Figure 6.8) shows features similar to those already observed for backgammon. Again, the self-play learners demonstrated slower initial learning than the play-versus-*pubeval* learners, but achieved similar asymptotic performance after 50,000 games. As with backgammon, the play-versus-*RandomPlayer* learner again achieved its asymptotic level of performance after a relatively short number of training games, but that level, while clearly lower than that of the other two learners, was not as substantially inferior as the equivalent performance achieved for backgammon.

This seems eminently reasonable; since hypergammon is a far simpler game than backgammon, it stands to reason that there are fewer strategic and tactical skills to learn. Thus, a learner that

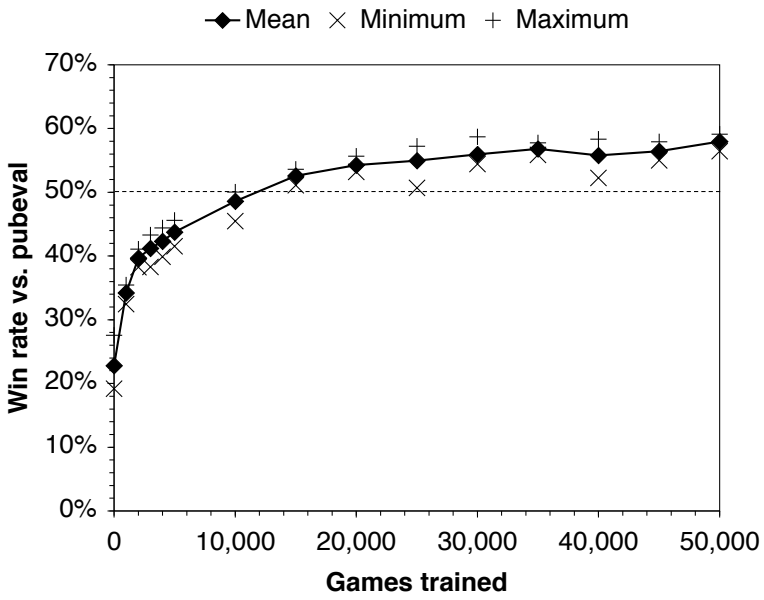

Figure 6.5: Performance of hypergammon networks trained by self-play.

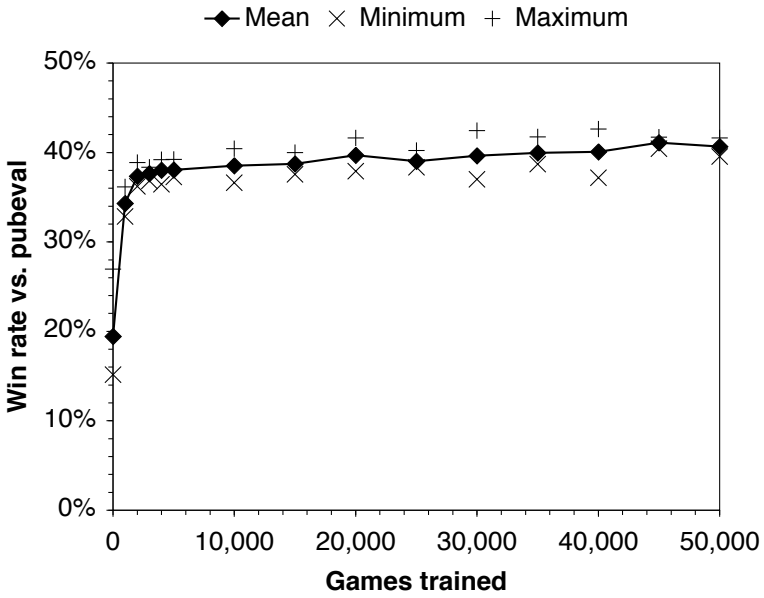

Figure 6.6: Performance of hypergammon networks trained by playing against *RandomPlayer*.

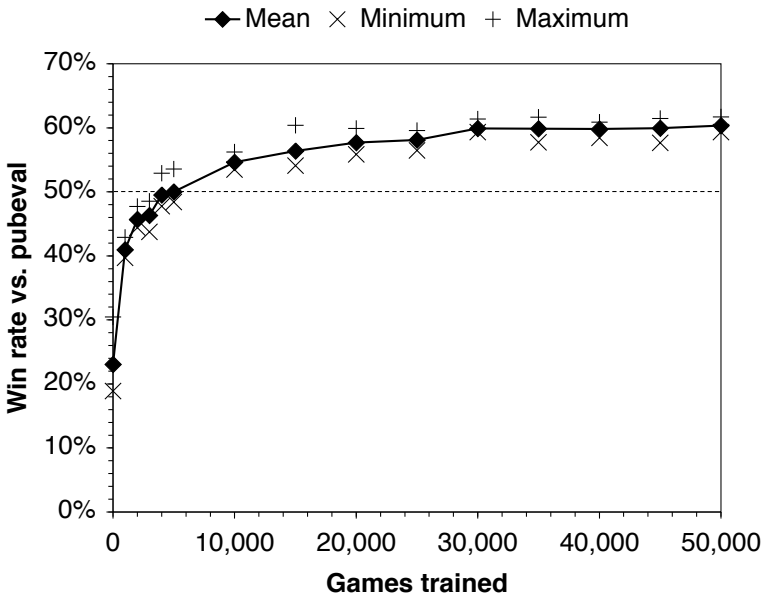

Figure 6.7: Performance of hypergammon networks trained by playing against *pubeval*.

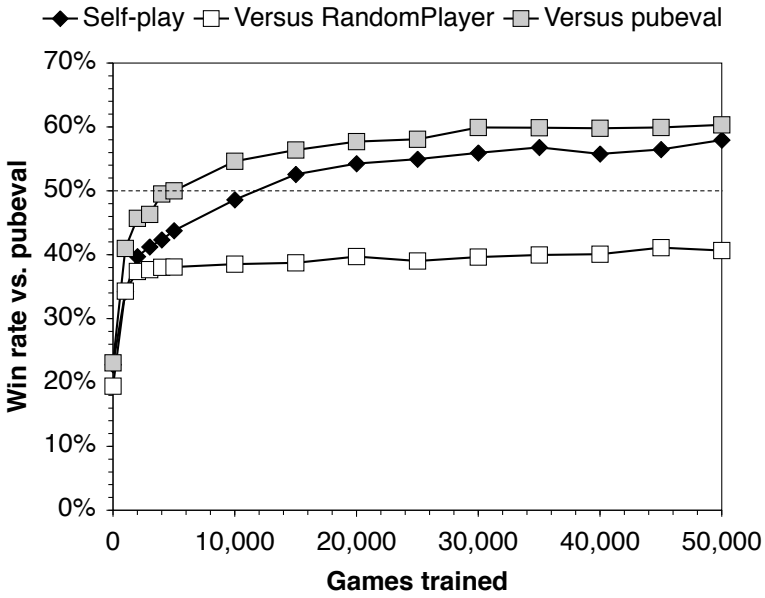

Figure 6.8: Mean performance of hypergammon networks produced by each learning environment.

failed to develop such skills (due to never needing them to prevail against the inferior opponent it was learning against) would not be at as much of a disadvantage, when playing against a skilled opponent, as would be a learner who failed to learn the details of backgammon, playing in a similar situation.

As previously discussed with backgammon, it is again worth mentioning the possibility that the play-versus-*pubeval* learners showed some degree of overfitting to that particular opponent. Therefore, it is also again noteworthy that the self-play learners managed to achieve similar performance with no such risk of overfitting.

6.3 PACHISI

Since pachisi is a team game, the learner-versus-other learning environment can be interpreted as either a single learner playing, with other players serving as both opponents and teammates, or a team of learners playing against a team of other players. Thus, in addition to the pure self-play learning environment (Figure 6.9), we performed both of these sorts of experiments: the learner playing with three *RandomPlayer*s or three *HeuristicPachisiPlayer*s (Figures 6.10 and 6.11), and a team of learners playing against a team of *RandomPlayer*s or *HeuristicPachisiPlayer*s (Figures 6.12 and 6.13).

The self-play learner was most successful, achieving a best performance of 27.6% in team play against a team of *HeuristicPachisiPlayer*s. The play-with-3-heuristic learner achieved 24.0%, the play-with-3-random learner achieved 17.3%, the play-versus-heuristic-team learner achieved 13.2%, and the play-versus-random-team learner achieved 11.5% best win rate against a team of *HeuristicPachisiPlayer*s. A win rate of 50% would indicate a playing level equal to that of *HeuristicPachisiPlayer*, so it appears that none of these learners were very successful given that they were unable to match the performance of such a simplistic heuristic strategy, even after 50,000 games of training.

A comparison of the results of these experiments (Figure 6.14) shows that a few patterns appear to be clear. The play-with-3-others learners consistently performed better than the play-versus-

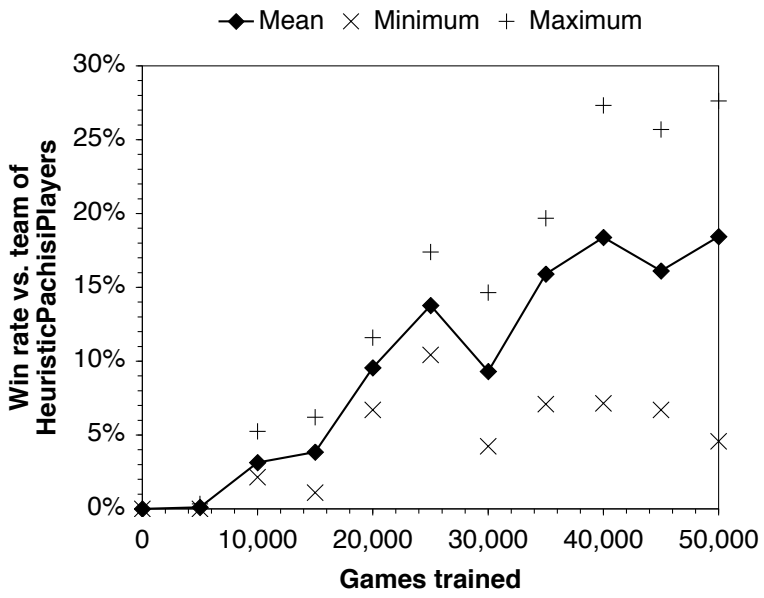

Figure 6.9: Performance of pachisi networks trained by self-play.

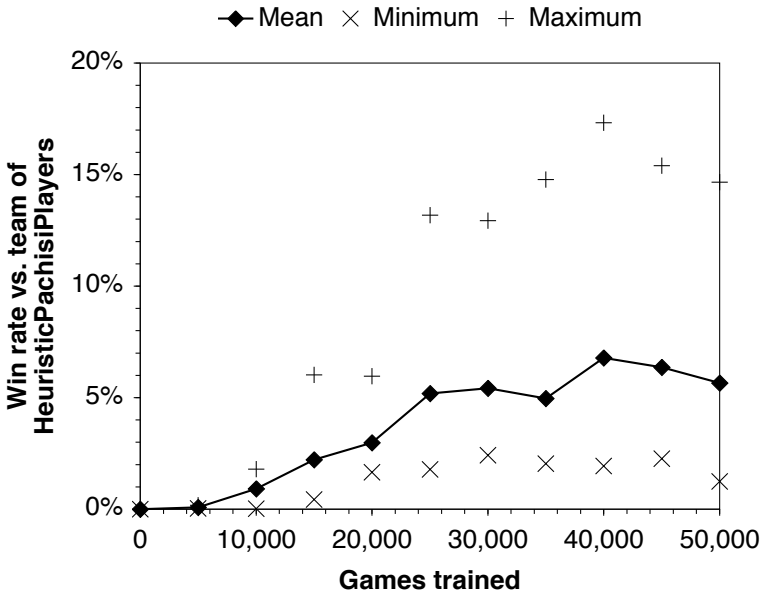

Figure 6.10: Performance of pachisi networks trained by playing with 3 *RandomPlayer*s.

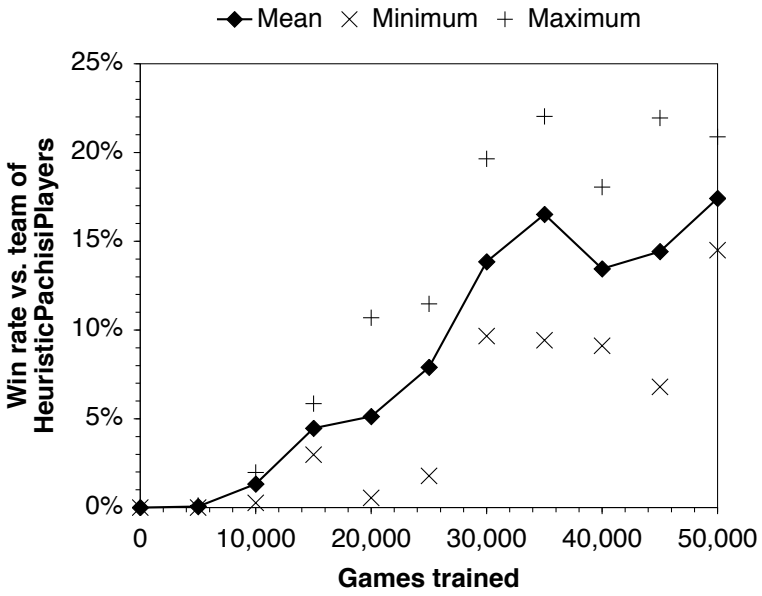

Figure 6.11: Performance of pachisi networks trained by playing with 3 *HeuristicPachisiPlayer*s.

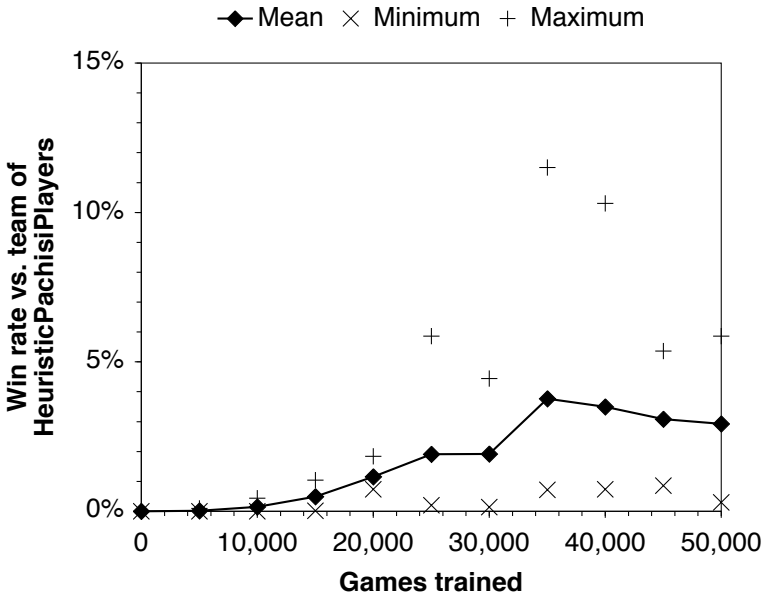

Figure 6.12: Performance of pachisi networks trained by team play against *RandomPlayer*s.

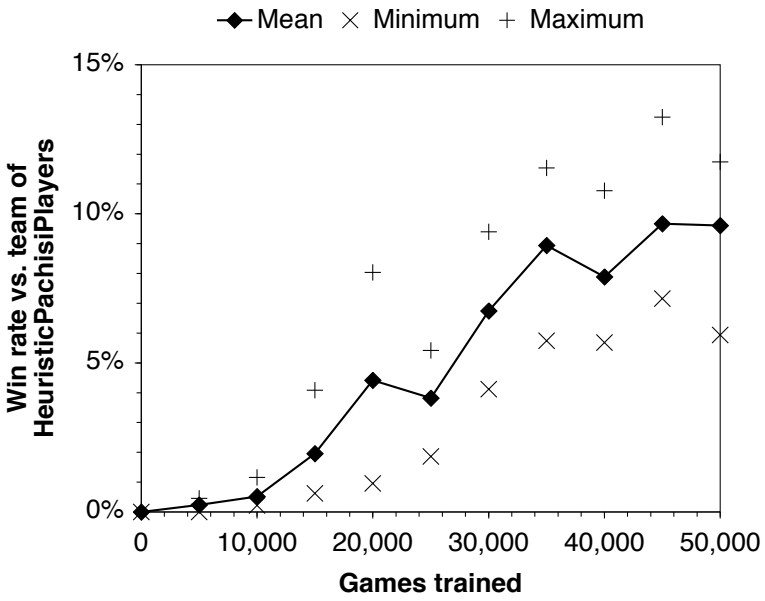

Figure 6.13: Performance of pachisi networks trained by team play against *HeuristicPachisiPlayer*s.

team learners that used the same opponent. It also seems reasonably clear that the learners using *HeuristicPachisiPlayer*s as other players consistently became stronger players than those using *RandomPlayer*s.

These pachisi results are quite different from those observed with backgammon and hypergammon, given that they are much noisier and do not seem to demonstrate the usual learning curve of rapid initial improvement followed by deceleration and approach to an asymptotic level of performance. The performance of each individual learner was also much more variable — it was quite common for a single learner to drop substantially in performance after a few thousand games of training, then rebound to a high level of performance again after a few thousand games more.

We argue that, given that these issues were observed for all learning environments, it would seem that the learning environment is not the cause of these issues. Instead, a likely explanation

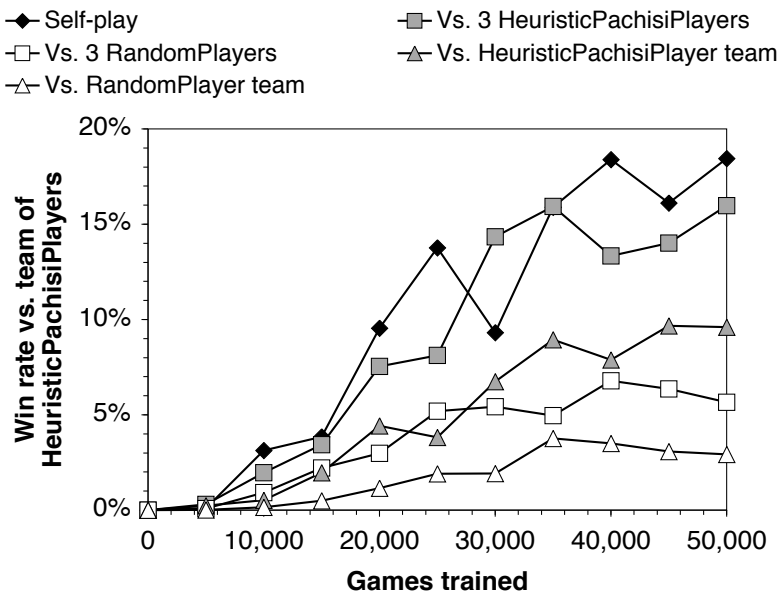

Figure 6.14: Mean performance of pachisi networks produced by each learning environment.

is that our unary board representation for this game does not adequately represent key features of the game. (The results in the following chapter support this argument somewhat.) In particular, it seems likely in retrospect that the truncation of the representation, which for this game draws no distinction whatsoever between a player having two, three, or all four pieces on a single space, might lead to the development of poor strategy.

That is, while the learner will probably learn that there is no safety in numbers (multiple pieces on a space may be captured as easily as a lone piece), it might be deceived by the truncated representation into selecting a move that did not reflect that wisdom. For example, when presented with two pieces on a highly vulnerable space and a third piece on a less vulnerable space nearby, it might opt to move that third piece onto the other two, seeing this as a move from "two vulnerable

here and one vulnerable here" to "two vulnerable here (due to the truncation) and no vulnerable here — fewer vulnerable pieces is better!" which could in fact possibly be a very bad move.

Unlike the backgammon and hypergammon experiments, for pachisi it appears that self-play was not significantly inferior to training against the opponent that would later be used for evaluation; indeed, on average, the latter learning approach seemed to be slightly inferior. It is possible that, given the relative simplicity of *HeuristicPachisiPlayer*, playing against it resulted in exploration of only a small subset of the game's state space, resulting in a not particularly strong trained player.

6.4 PARCHEESI

Since Parcheesi (as studied in this research) is a four-player game with no team aspect, some hybrid learning environments were possible between the extremes of pure self-play and one learner versus a single opponent (or several copies thereof). These possibilities included mixing self-play and play-versus-opponent environments (by using several learners and several other opponents together in one game) and mixing several opponents into a single learning environment.

We selected six learning environments to study for parcheesi: pure self-play (Figure 6.15); pure learner-vs-single-opponent, specifically in the form of one learner versus either three *RandomPlayer*s (Figure 6.16) or three *HeuristicParcheesiPlayer*s (Figure 6.17); and hybrid approaches in the form of two copies of the learner playing with either two *RandomPlayer*s (Figure 6.18), two *HeuristicParcheesiPlayer*s (Figure 6.19), or one of each of these (Figure 6.20).

For the sake of curiosity, we also performed for Parcheesi a set of experiments in which the learner did not itself play while learning, but instead learned by observing games played by others, specifically, games of Parcheesi played by two *HeuristicParcheesiPlayer*s and two *RandomPlayer*s (Figure 6.21).

Unlike the other games, for Parcheesi there was little difference between the best performance of players trained by most of these learning environments: with the exception of the play-versus-3-*RandomPlayer*s learners, which achieved a maximum performance of 27.1% versus three *Heu-*

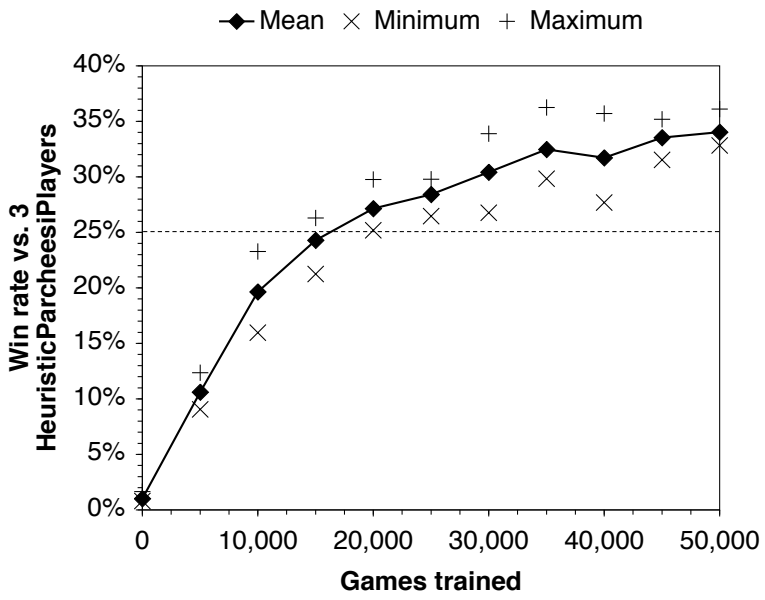

Figure 6.15: Performance of Parcheesi networks trained by self-play.

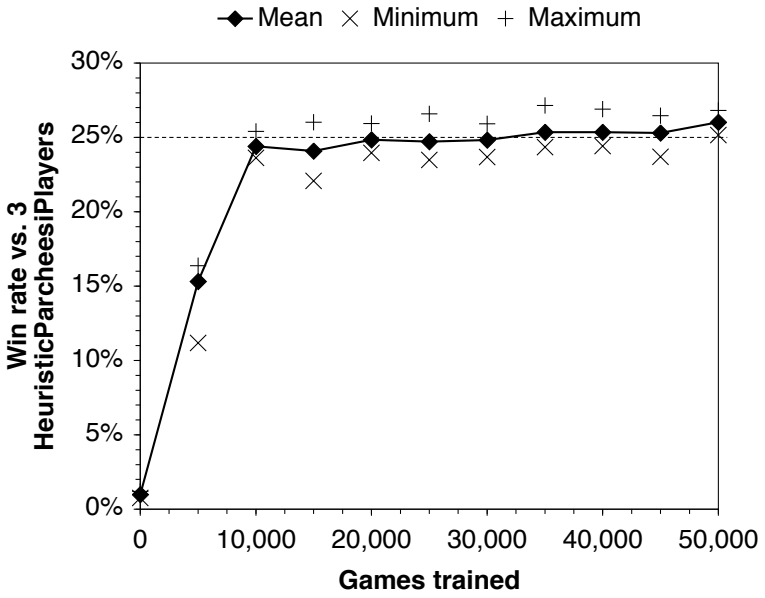

Figure 6.16: Performance of Parcheesi networks trained by playing against 3 *RandomPlayer*s.

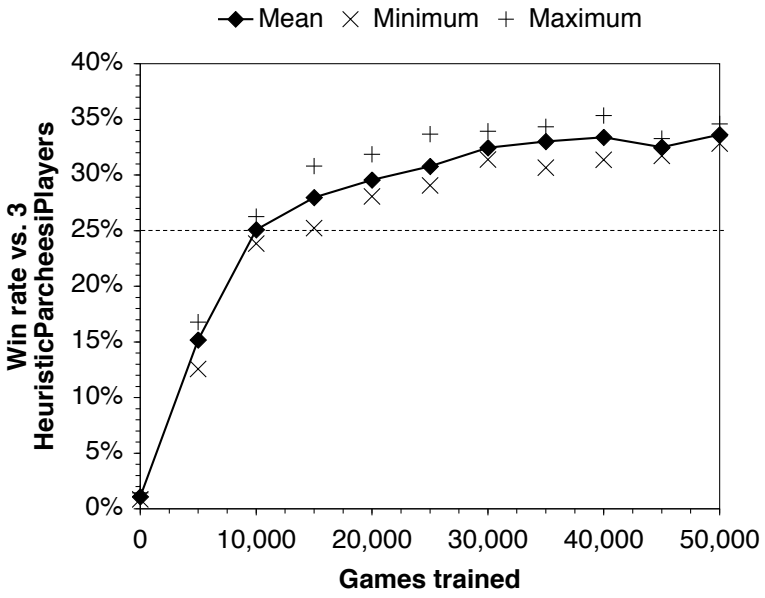

Figure 6.17: Performance of Parcheesi networks trained by playing against 3 *HeuristicParcheesi-Player*s.

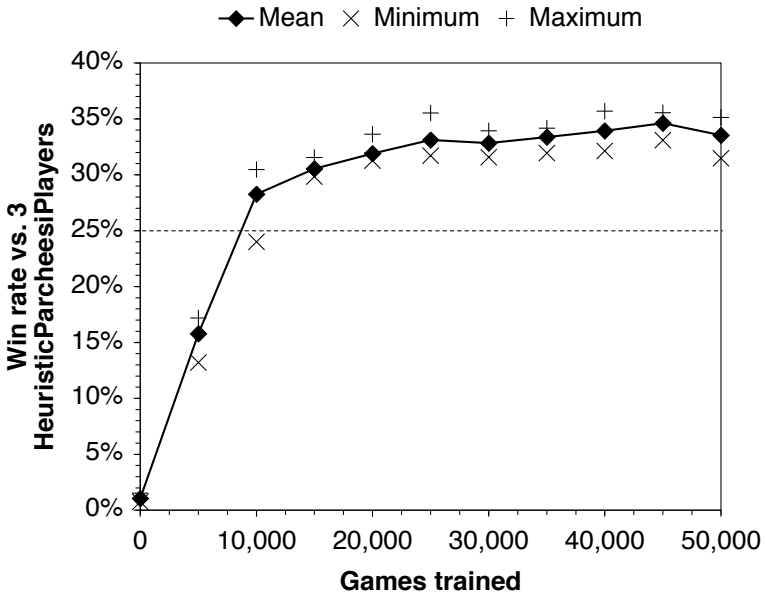

Figure 6.18: Performance of Parcheesi networks trained by playing against self and 2 *Random-Player*s.

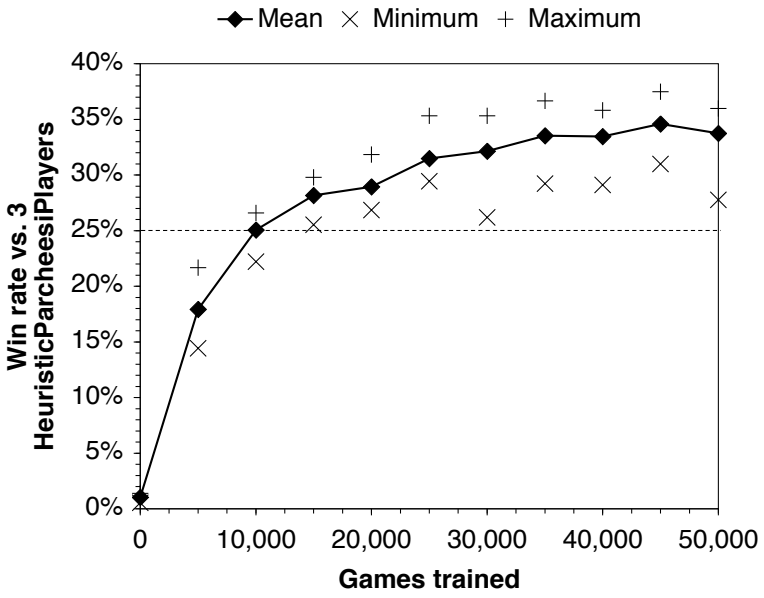

Figure 6.19: Performance of Parcheesi networks trained by playing against self and 2 *HeuristicParcheesiPlayer*s.

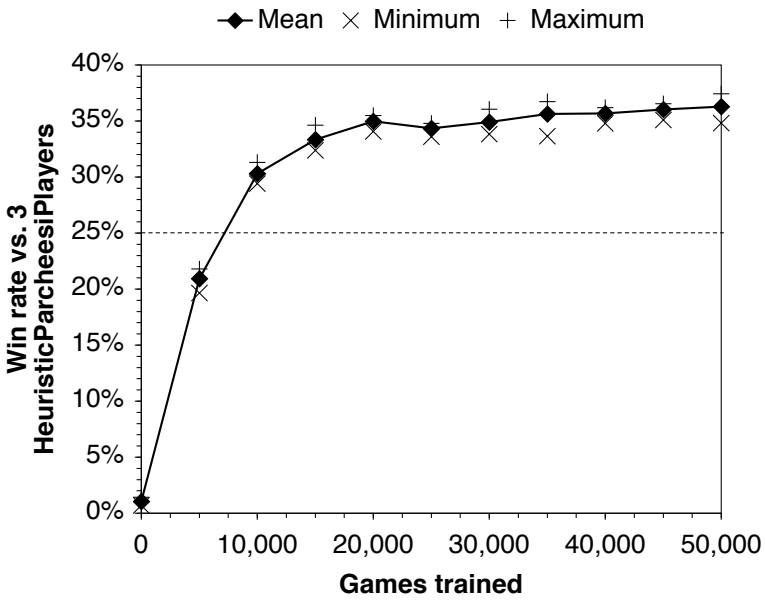

Figure 6.20: Performance of Parcheesi networks trained by playing against self, 1 *RandomPlayer*, and 1 *HeuristicParcheesiPlayer*.

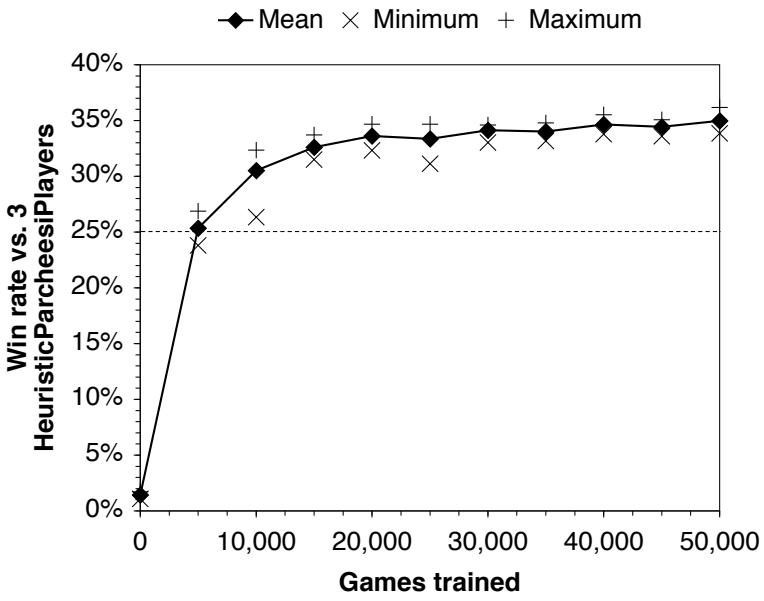

Figure 6.21: Performance of Parcheesi networks trained by observing 2 *RandomPlayer*s playing against 2 *HeuristicParcheesiPlayer*s.

*risticParcheesiPlayer*s, all of the other approaches yielded a best performance between 35.3% and 37.5% against three HeuristicParcheesiPlayers.

Since these numbers are based on a single player against three *HeuristicParcheesiPlayer* opponents for evaluation, 25% wins would indicate equal playing skill, so it is clear that even the inferior play-versus-3-*RandomPlayer*s learner managed to develop skill somewhat superior to that of the simple HeuristicParcheesiPlayer.

A comparison of the learning curves for each experiment (Figure 6.22) shows that although the maximum performance of most approaches was roughly the same, the various approaches differed somewhat in their rates of learning in approaching that maximum performance. The self-play learner was the slowest to learn throughout the experiment, and the observation-only learner was the fastest to learn initially, but was surpassed (albeit only slightly) after about 10,000 games

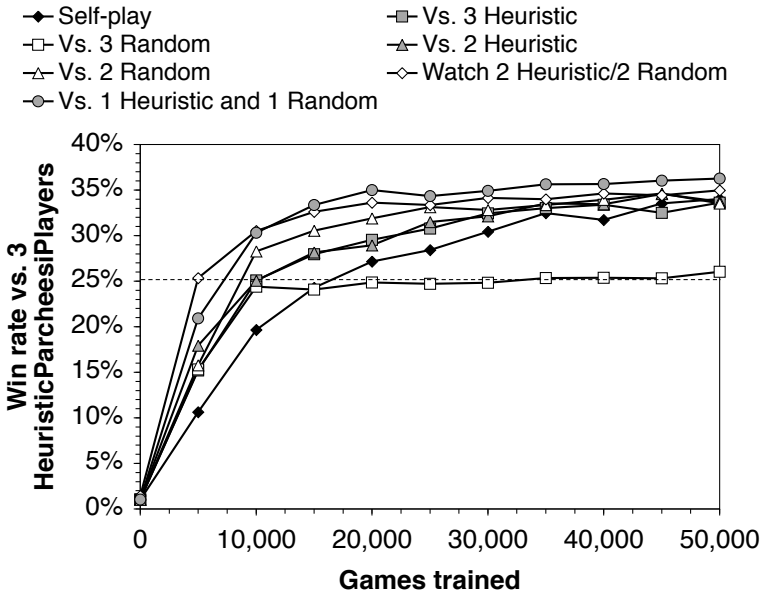

Figure 6.22: Mean performance of Parcheesi networks produced by each learning environment.

by the hybrid learner that combined self-play with play versus both other player types as well. This suggests that for this game and that particular learning environment, at least, the absence of learner involvement in the training games was neither particularly an asset nor a hindrance.

As with the other games, it is possible that the learning environments that included the skilled *HeuristicParcheesiPlayer* could have led to some degree of overfitting by the learner. However, training against the *HeuristicParcheesiPlayer* apparently did not lead to much significant improvement compared to other training environments, suggesting that there is little potential benefit to such an approach in this case.

It is potentially interesting that in this game, the hybrid approaches that included *RandomPlayer*s generally produced slightly more skilled players than the hybrid approaches including *HeuristicParcheesiPlayer*, yet of the pure learner-versus-opponent approaches (those lacking

self-play), the play-versus-3-*RandomPlayer*s learner was clearly inferior to the play-versus-3-*HeuristicParcheesiPlayer*s learner. Perhaps when self-play is present, a little extra bit of randomness in the game allows better exploration of the game, and thus better learning, than the presence of a different sort of skilled player can.

6.5 Conclusions

For all four of these games, several general observations can be stated about the various learning environments.

Learning by playing against a skilled opponent generally produces a highly skilled player, but raises the possibility of overfitting to that player, especially given that the training opponent is the same player later used for performance evaluation of the trained network. Furthermore, this approach leads to only a slight (if any) edge in performance as compared to the best networks trained by other methods, so it is probably best avoided unless achieving the absolute highest possible performance against a given opponent is necessary.

Learning by playing against an unskilled opponent, on the other hand, consistently produces a much poorer player than those produced by self-play or playing against a skilled opponent. This approach is obviously best avoided, but it should also serve as a caution against the careless use of *any* non-learning opponent for training, as there is always the possibility that if the learner manages to surpass the playing level of its opponent, its subsequent learning will be similarly hampered by the inability of its opponent to mount a serious challenge to its newfound skill.

Therefore, these results seem to support our hypothesis, that in the context of nondeterministic games and temporal difference learning, pure self-play does not generally produce players significantly inferior to those produced by training against a skilled opponent, and generally does produce players significantly superior to those produced by training against an unskilled opponent. When in doubt, it seems, choose self-play.

CHAPTER 7

INFLUENCE OF THE BOARD REPRESENTATION

7.1 EXPERIMENTAL PROCEDURE

The experimental results for a unary representation network trained by self-play had already been generated as part of the experiments in the previous chapter, so only additional experiments using the smart and combined representations were performed at this stage.

Initial experiments with the smart and combined representations for backgammon produced unexpectedly poor results, to the extent that the combined (unary plus smart) representation produced players far inferior to those produced for the unary representation alone. Further experimentation demonstrated that the apparent cause of this poor performance was the large magnitude of some of the inputs produced by the smart representation — while some inputs such as the turn indicators had values of zero or one, on par with the inputs of the unary representation, others, such as the pip count and the contact, could potentially be as large as 100 or more. It appeared that this discrepancy made it more difficult for the network to learn, as simply altering the representation to normalize these values (by dividing each derived input by its theoretical maximum value) resulted in vastly improved performance (Figures 7.1, 7.2). All subsequent experiments for backgammon and the other three games therefore used similar scaling for their smart features.

7.2 BACKGAMMON

The results of training networks using the unary backgammon representation have previously been presented (Figure 6.1) and so are not repeated here. While that approach yielded a best performance of 48.6% wins against *pubeval*, networks using the smart representation alone achieved a maximum

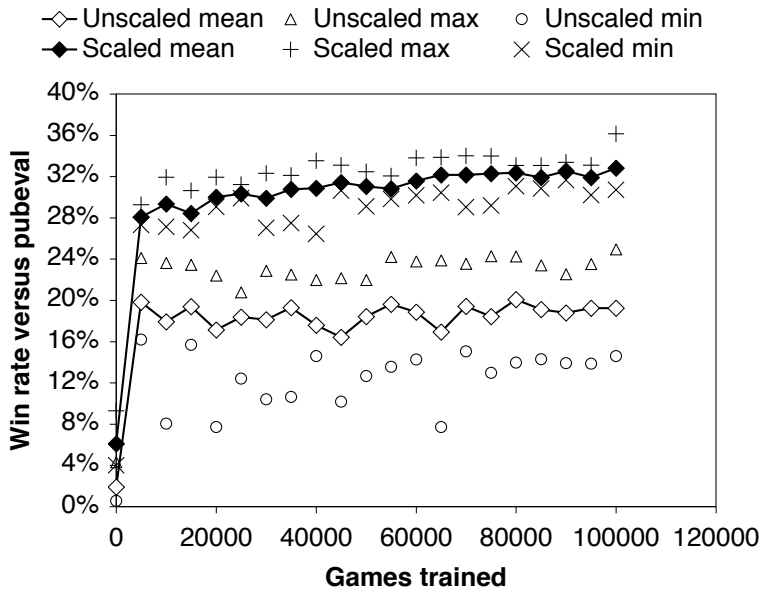

Figure 7.1: Effect of input scaling on the performance of backgammon networks using the smart representation.

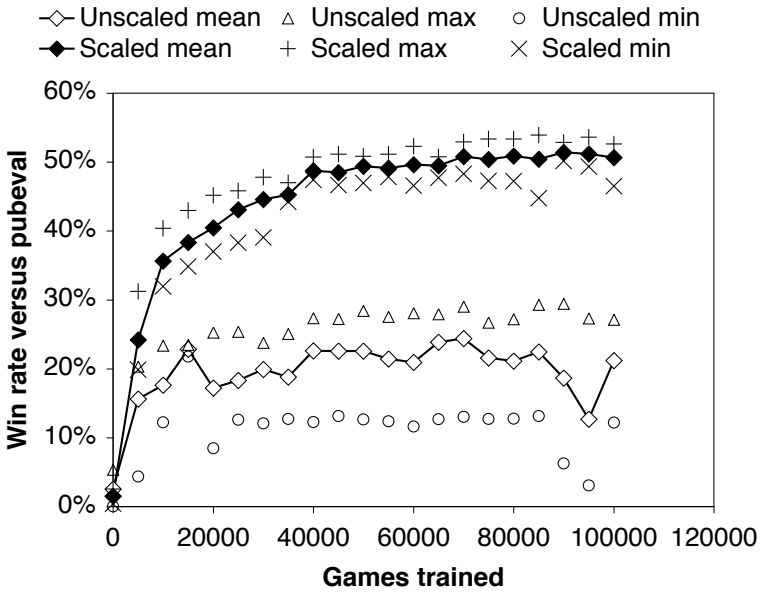

Figure 7.2: Effect of input scaling on the performance of backgammon networks using a combination of smart and unary representations.

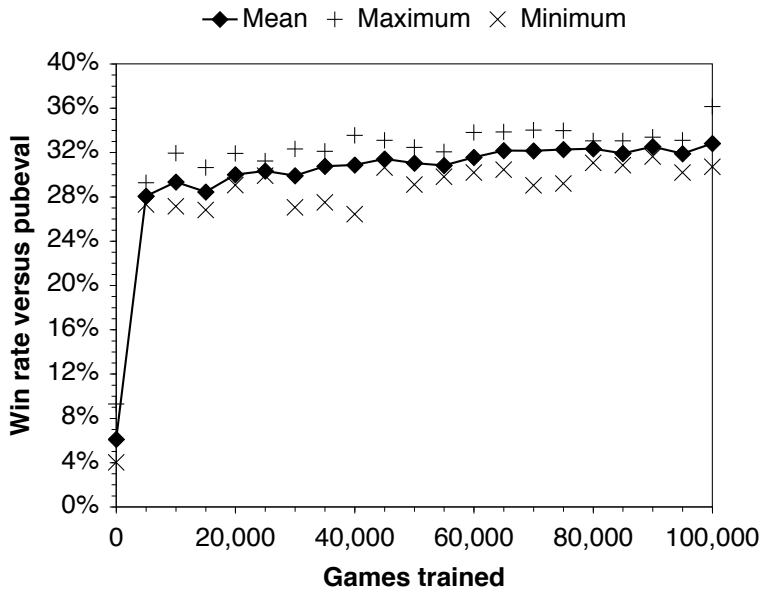

Figure 7.3: Performance of backgammon networks using smart representation alone.

winning rate of 36.1% (Figure 7.3), and networks using a combination of these two representations achieved 53.9% (Figure 7.4).

A comparison of these three experiments (Figure 7.5) shows that the smart representation networks initially led in performance but soon stagnated and were surpassed by the other networks. The combined representation was superior to the unary representation alone at all stages of training, suggesting that while the smart representation by itself was inferior to the unary representation, it was by no means redundant and contained some informational content that could be of benefit above and beyond the information provided by the unary representation.

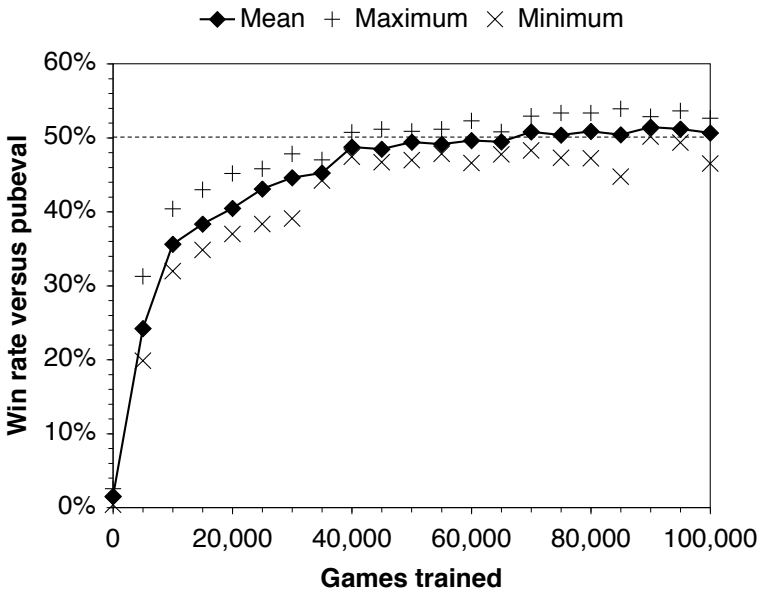

Figure 7.4: Performance of backgammon networks using combined representation.

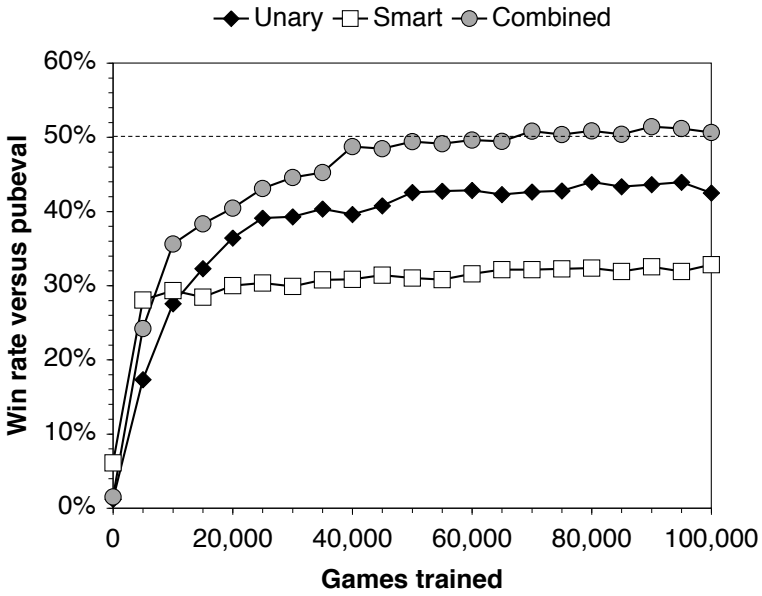

Figure 7.5: Mean performance of backgammon networks using each representation.

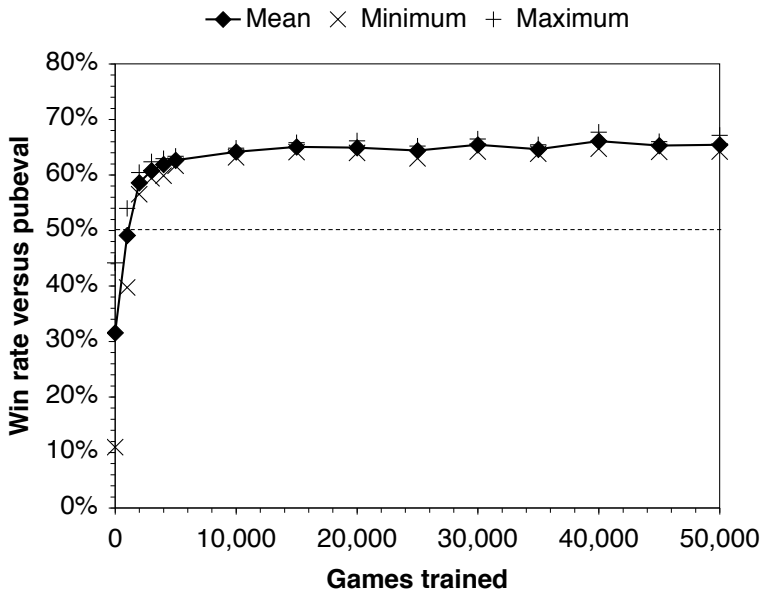

Figure 7.6: Performance of hypergammon networks using smart representation alone.

7.3 HYPERGAMMON

Networks using the unary representation achieved a best performance of 59.1% wins versus *pubeval*, as previously described (Figure 6.5), while networks using the smart representation or a combination of the two representations achieved best performances of 67.7% and 68.7% wins respectively (Figures 7.6, 7.7).

A comparison of the results of these three experiments (Figure 7.8) shows that unlike backgammon, for hypergammon it appears that the smart representation was essential to optimal performance, while the unary representation was almost entirely unnecessary. The combined representation learned more slowly at first than the smart representation and achieved only slightly better asymptotic mean performance after 50,000 training games.

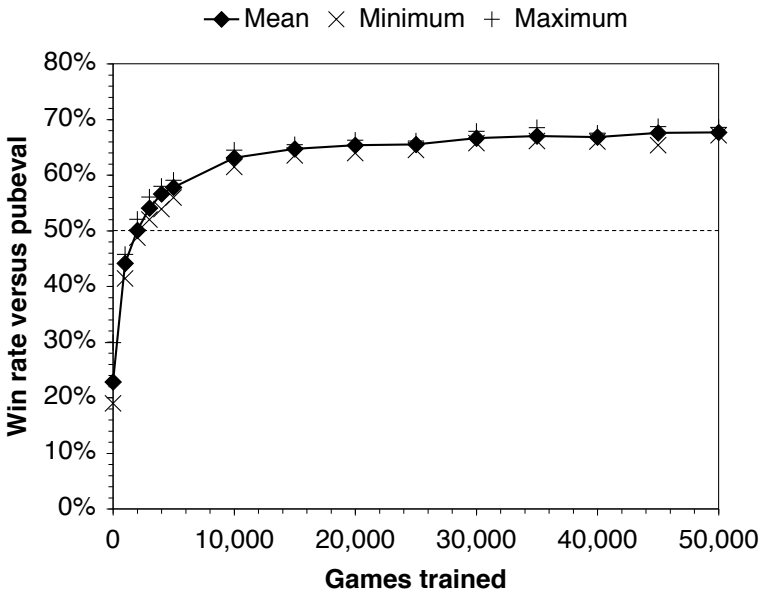

Figure 7.7: Performance of hypergammon networks using combined representation.

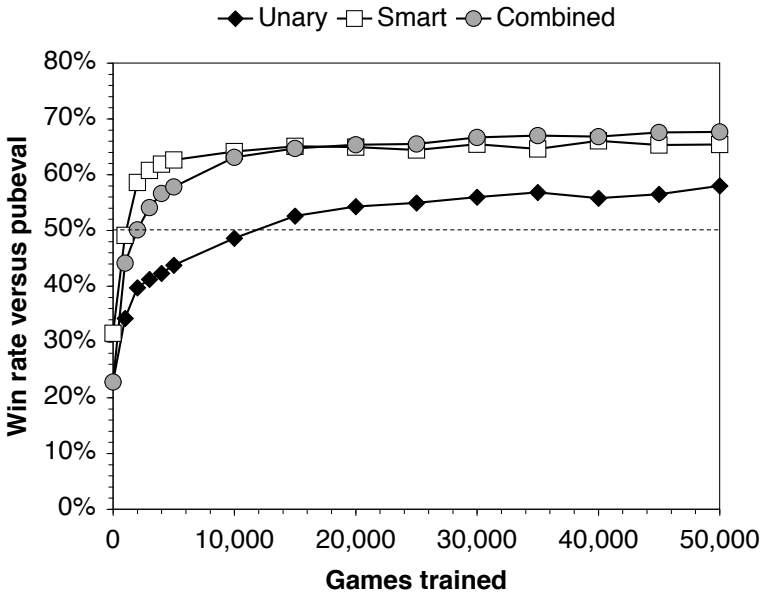

Figure 7.8: Mean performance of hypergammon networks using each representation.

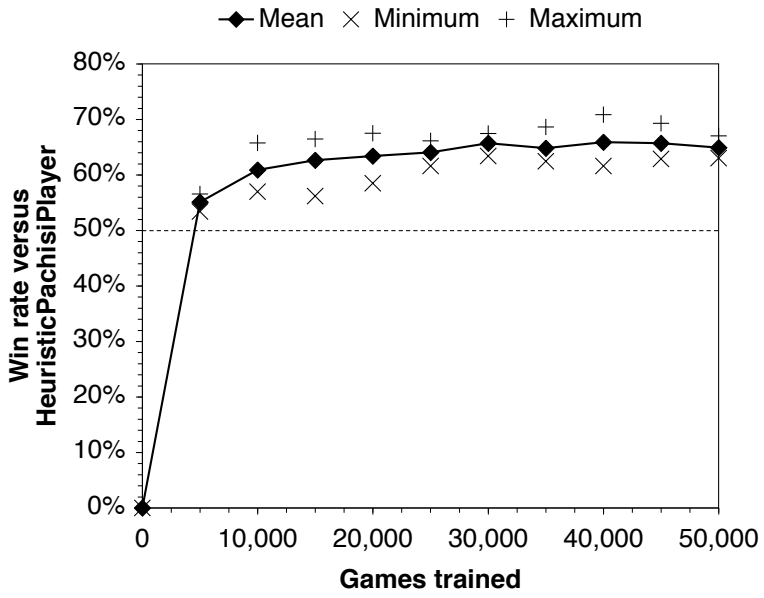

Figure 7.9: Performance of pachisi networks using smart representation alone.

This suggests that for hypergammon, the given set of derived features captures nearly every relevant aspect of the board, and that no particularly significant tactical or strategic features are discarded in the construction of the "smart" board representation.

7.4 PACHISI

Networks using the unary representation achieved a best performance of only 27.6% wins versus a team of *HeuristicPachisiPlayer*s, as previously described (Figure 6.9). By contrast, networks using the smart representation or a combination of the two representations achieved best performances of 70.8% and 77.2% wins respectively (Figures 7.9, 7.10).

A comparison of the results of these three experiments (Figure 7.11) shows that for pachisi, the derived features were essential to the development of a player more skilled than *HeuristicPachisi-*

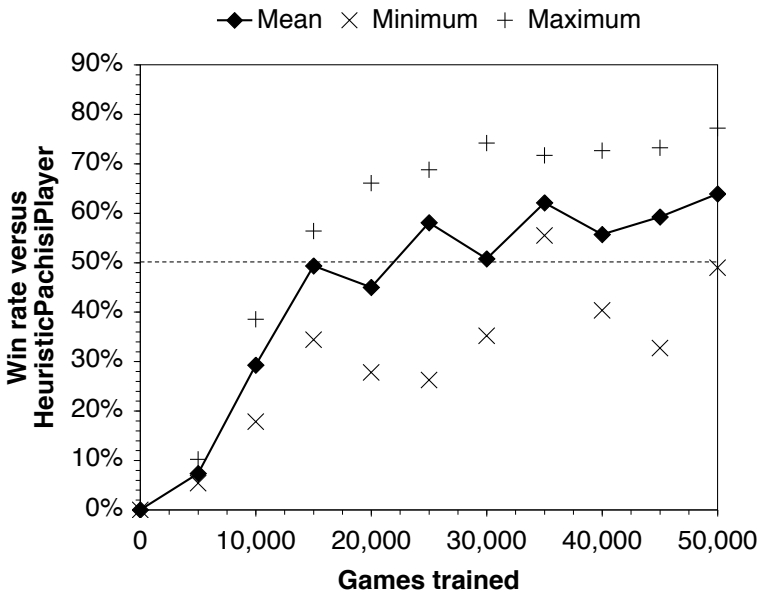

Figure 7.10: Performance of pachisi networks using combined representation.

Player. Although the learners using the combined representation achieved slightly higher best performance than those using the smart representation alone, on average, the combined-representation learners were consistently inferior and much slower to learn, achieving average performance equivalent to that of the smart-representation learners only at the very end of the training period.

Given the vastly superior performance of players developed using the smart feature representation, it seems clear that this representation captured many key aspects of this game in ways that the truncated unary representation did not. The fact that adding in the unary representation to the smart representation generally hindered and added noise to the development of the player supports the argument made in Section 6.3, that our truncated unary representation for this game apparently either failed to represent key aspects of the game or perhaps was even misleading to the learner in some fashion.

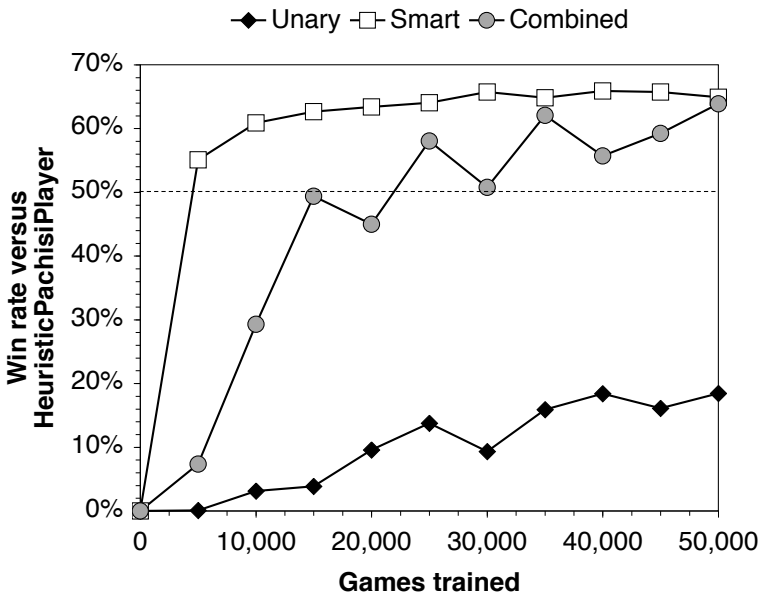

Figure 7.11: Mean performance of pachisi networks using each representation.

However, the fact that the highest-performing player using the combined representation was substantially better than the highest-performing player using the smart representation alone suggests that the unary representation is not *entirely* without merit, merely significantly flawed. It seems reasonable to conjecture that a learner using an improved (perhaps merely untruncated) unary representation in combination with the given smart features would be able to achieve even greater performance.

7.5 PARCHEESI

Networks using the unary representation achieved a best performance of 36.2% wins versus 3 *HeuristicParcheesiPlayer*s, as previously described (Figure 6.15), while networks using the smart

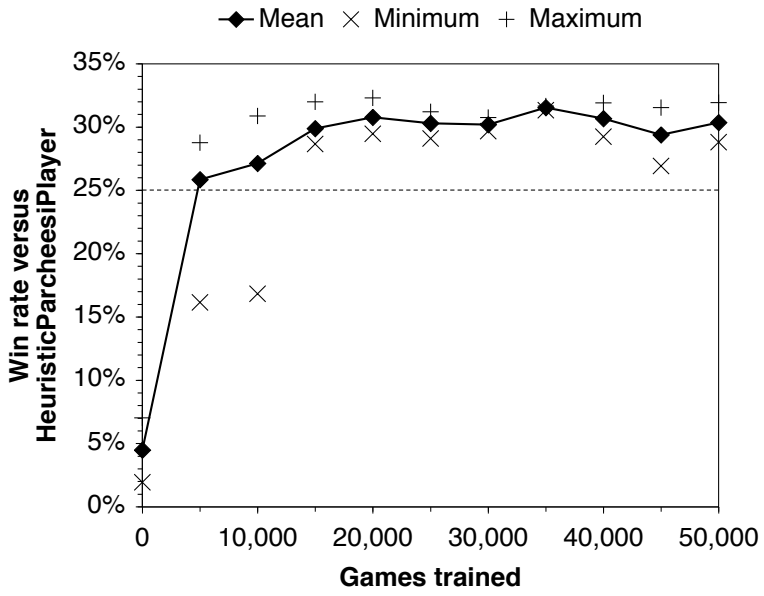

Figure 7.12: Performance of Parcheesi networks using smart representation alone.

representation or a combination of the two representations achieved best performances of 32.3% and 44.9% wins respectively (Figures 7.12, 7.13).

A comparison of these three experiments (Figure 7.14) shows that the smart representation learners initially led in performance but soon stagnated and were surpassed, first by the combined representation learners, then eventually by the unary representation learners as well. The combined representation was superior to the unary representation alone at all stages of training, suggesting that while the smart representation and unary representation performed somewhat similarly on their own, they were by no means redundant with one another, and each contained some informational content that could be beneficial above and beyond the information provided by the other representation.

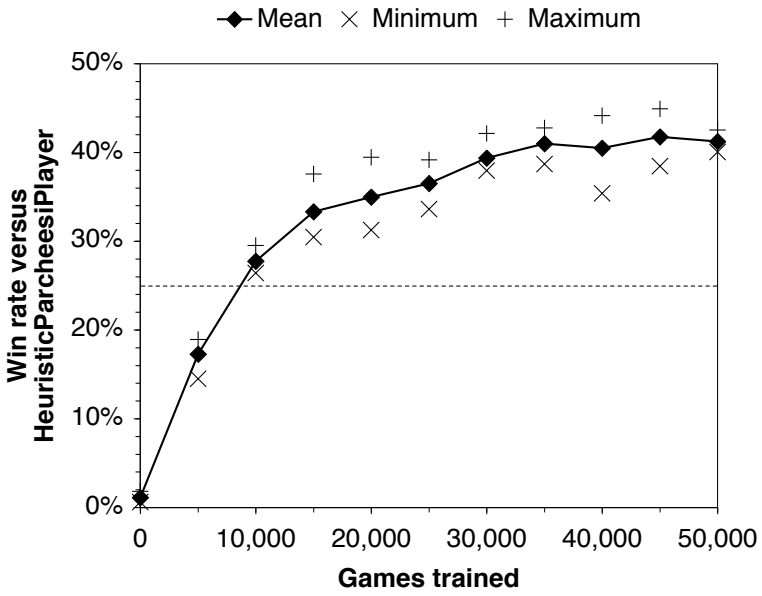

Figure 7.13: Performance of Parcheesi networks using combined representation.

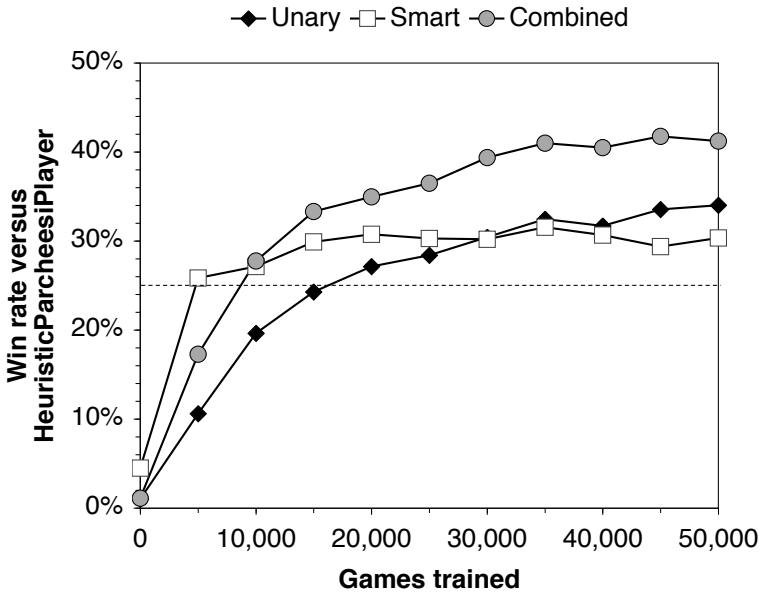

Figure 7.14: Mean performance of Parcheesi networks using each representation.

7.6 Conclusions

These results do not provide conclusions as clear-cut as those of the previous chapter. For Parcheesi and backgammon alike, the smart feature representation was insufficient to capture all relevant aspects of the game by itself, but it could be combined with the unary representation to yield a representation that was greater than the sum of its parts. By contrast, for hypergammon, the smart representation was sufficiently powerful on its own as to make the unary representation almost totally redundant, and for pachisi, the smart representation was very powerful, but learners benefitted very slightly (in terms of their best performance) from the inclusion of the unary representation along with it.

At the very least, it appears that the use of a combined representation does not generally harm the resulting performance of the trained network, given sufficient training time. When only a short training period is feasible, the use of smart features alone may be the best approach, but in the long run, the combined representation seems to be the strongest approach.

CHAPTER 8

CONCLUSIONS AND POSSIBLE DIRECTIONS FOR FUTURE WORK

The experimental results presented in this thesis clearly demonstrate the following:

- TD(λ) can be successfully used to train neural networks as players of many nondeterministic board games. The basic TD(λ) algorithm suffices for these games, without any of the modifications to the learning algorithm that have been necessary for success with games such as Chess and Othello.

- A self-play learning environment is generally adequate for the development of very strong players by this approach; no expert player or data from expert games is required.

- When used as inputs to these neural networks, both a naïve unary representation of the board state and a set of derived, or "smart," features can contribute significantly to the strength of the player; in general, therefore, using a combination of these results in the strongest players.

These results demonstrate that the selection of learning environment and board representation have clear influence on the strength of the resulting TD(λ)-trained neural networks, but it should be emphasized that the goal of this research was to demonstrate these differences, not necessarily to produce the best possible player for each game. Therefore, one possible followup to this work would be attempting to develop stronger players.

While it is possible that better players could in fact be produced by other learning approaches (in fact, hypergammon in particular is a simple enough game that Sconyers (2006) has calculated all possible outcomes of it and developed a program that can play the game optimally), it also seems quite likely that significantly stronger players could be developed merely by optimizing the TD(λ) and neural network approach. Two obvious routes to further improvement would be longer

training and the use of more hidden units in the neural networks. We recall that the latest version of *TD-Gammon* (Tesauro 2002) used 160 hidden units and 2,000,000 training games — many more than the mere 10 hidden units and 50,000–100,000 training games that we used in this research.

As a quick test of this hypothesis, we ran a small number of experiments using networks with 50 hidden units each, the combined board representation, and the training environment we found to be most effective for each game.

In backgammon, we performed a single experiment in which a network of this sort was trained against *pubeval*. After 150,000 games of training, this network achieved a best performance of 59.6% wins against *pubeval*, clearly superior to the best-of-5-networks performance of 53.9% that we previously achieved by training 10-hidden-unit, combined-representation networks by self-play for 100,000 games each. This performance, in fact, compares favorably with that of every other backgammon player described in Table 4.1, suggesting that for backgammon, at least, the TD(λ)-trained neural network approach remains the king of the hill.

Given that hypergammon is a solved game, as previously mentioned, attempting to optimize our approach to the game may seem foolish, but we attempted it anyway just to see. By training a 50-hidden-unit, combined-representation network against *pubeval*, we achieved after 175,000 training games a performance of 70.1% wins against *pubeval*, a slight improvement over our previous best value of 68.7%. However, after 50,000 training games, this network had only achieved a best performance of 68.1%, suggesting that this slight improvement was due to the greatly increased training time rather than the increased number of hidden nodes.

Another possible source of improvement to the strength of these networks could be augmenting the "smart" feature set, either with additional general features or with some specialized game-specific features. For example, the "presence of a prime" feature from *Neurogammon* and *TD-Gammon* would probably benefit the development of a backgammon player, and features similar to those in *ACT-R-Gammon* that would indicate board features that have changed as the result of a move might be useful for most games.

BIBLIOGRAPHY

Y. Azaria and M. Sipper. GP-Gammon: Using genetic programming to evolve backgammon players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of 8th European Conference on Genetic Programming*, pages 132–141, 2005.

J. Baxter, A. Tridgell, and L. Weaver. KnightCap: A chess program that learns by combining TD(λ) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning*, 1998.

M. Heinze, D. Ortiz-Arroyo, H. L. Larsen, and F. Rodriquez-Henriquez. Fuzzeval: A fuzzy controller-based approach in adaptive learning for backgammon game. In A. Gelbukh, A. Albornoz, and H. Terashima-Marín, editors, *MICAI 2005: Advances in Artificial Intelligence*, pages 224–233. Springer-Verlag, 2005.

J. Heled. The GNUbg training program. `http://pages.quicksilver.net.nz/pepe/ngb/index.html`, 2003.

J.-S. R. Jang, C.-T. Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, 1996.

JellyFish AS. What is JellyFish? `http://www.jellyfish-backgammon.com/whatis.htm`, 2004.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

T. Keith. Standard rules of backgammon. `http://www.bkgm.com/rules.html`, 2006.

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

A. Leouski. Learning of position evaluation in the game of Othello. Master's thesis, University of Massachusetts, 1995.

Masters Games Ltd. The rules of pachisi. `http://www.mastersgames.com/rules/pachisi-rules.htm`, 1999.

T. M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

J. B. Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32(1):225–240, 1998.

D. Qi and R. Sun. Integrating reinforcement learning, bidding, and genetic algorithms. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT-2003)*, pages 53–59, Los Alamitos, CA, 2003. IEEE Computer Society Press.

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

S. Sanner. Simultaneous learning of structure and value in relational reinforcement learning. In *Proceedings of the ICML'05 Workshop on Rich Representations for Reinforcement Learning*, 2005.

S. Sanner, J. R. Anderson, C. Lebiere, and M. Lovett. Achieving efficient and cognitively plausible learning in backgammon. In P. Langley, editor, *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 823–830, Stanford, CA, 2000. Morgan Kaufmann.

N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Temporal difference learning of position evaluation in the game of Go. *Advances in Neural Information Processing*, 6, 1994.

H. Sconyers. Backgammon variants: Hyper-backgammon. `http://www.bkgm.com/variants/HyperBackgammon.html`, 2006.

Selchow & Righter Company. Parcheesi®: Royal game of india: Playing instructions. Amsterdam, NY, 1987.

SnowieGroup SA. What is snowie 4. `http://www.bgsnowie.com/snowie/snowie.dhtml`, 2006.

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

R. S. Sutton. Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation. Technical Note TN87-509.1, GTE Laboratories, 1989. Published 1987, corrected 1989.

G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.

G. Tesauro. FTPable benchmark evaluation function. Usenet post to `rec.games.backgammon`, available online at `http://www.bkgm.com/rgb/rgb.cgi?view+610`, February 1993.

G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

S. Thrun. Learning to play the game of chess. *Advances in Neural Information Processing Systems*, 7, 1995.

M. A. Wiering, J. P. Patist, and H. Mannen. Learning to play board games using temporal difference methods. Technical Report UU-CS-2005-048, Institute of Information and Computing Sciences, Utrecht University, 2005.

APPENDIX

FULL EXPERIMENTAL RESULTS

The following tables include the performance estimates for each partially trained network generated throughout each experiment performed in this research. As described in Section 5.5, these performance estimates are based on each network playing 5,000 games against the appropriate "skilled" opponent for its game, so performance of a network ranges from 0 games won (0% wins) to 5,000 games won (100% wins).

Table A.1: Backgammon results for unary representation and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	92	57	47	42	59
5,000	1,104	615	1,152	877	580
10,000	1,463	1,232	1,603	1,565	1,024
15,000	1,661	1,483	1,839	1,794	1,295
20,000	1,944	1,746	1,960	1,931	1,522
25,000	2,065	2,094	1,930	2,025	1,664
30,000	2,060	2,103	2,027	1,970	1,660
35,000	2,037	2,154	2,029	2,146	1,716
40,000	1,924	2,191	2,169	2,212	1,406
45,000	2,144	2,224	2,005	2,095	1,727
50,000	2,141	2,272	2,109	2,332	1,791
55,000	2,166	2,237	2,209	2,094	1,977
60,000	2,158	2,253	2,227	2,206	1,868
65,000	2,132	2,213	2,089	2,154	1,981
70,000	2,117	2,179	2,255	2,186	1,925
75,000	2,230	2,074	2,431	2,016	1,946
80,000	2,191	2,262	2,246	2,058	2,241
85,000	2,191	2,305	2,216	2,062	2,065
90,000	2,134	2,225	2,405	2,174	1,968
95,000	2,177	2,331	2,301	2,142	2,031
100,000	2,098	2,187	2,315	2,156	1,868

Table A.2: Backgammon results for unary representation and learning by play against *Random-Player*.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	73	55	42	28	61
5,000	228	237	262	224	264
10,000	236	282	309	188	251
15,000	256	307	333	238	272
20,000	323	269	270	215	237
25,000	324	272	267	170	227
30,000	327	249	309	186	201
35,000	305	230	316	220	256
40,000	281	297	337	221	238
45,000	322	282	318	214	229
50,000	309	284	281	239	272
55,000	220	274	284	206	204
60,000	271	287	300	184	235
65,000	308	299	324	257	193
70,000	311	260	306	213	191
75,000	298	367	303	246	172
80,000	306	303	249	193	223
85,000	304	299	242	253	219
90,000	263	275	230	218	213
95,000	321	318	256	206	171
100,000	282	265	264	209	185

Table A.3: Backgammon results for unary representation and learning by play against *pubeval*.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	75	28	47	49	50
5,000	1,490	1,399	1,556	1,544	1,608
10,000	1,932	1,939	2,115	2,132	1,687
15,000	1,772	2,084	1,914	1,992	1,983
20,000	1,674	2,081	2,137	2,025	1,993
25,000	2,129	2,164	2,102	2,241	1,958
30,000	2,079	2,162	2,172	2,163	2,101
35,000	2,258	2,314	2,259	2,119	2,187
40,000	2,127	2,294	2,218	2,156	2,224
45,000	2,300	2,314	2,300	2,175	2,109
50,000	2,337	2,304	2,184	2,214	2,265
55,000	2,318	2,277	2,248	2,002	2,239
60,000	2,198	2,341	2,114	2,375	2,252
65,000	2,258	2,263	2,164	2,206	2,190
70,000	2,192	2,294	2,377	2,367	2,348
75,000	2,065	2,229	2,154	2,350	2,225
80,000	2,135	2,204	2,315	2,299	2,316
85,000	2,281	2,163	2,242	2,250	2,331
90,000	2,292	2,262	2,225	2,304	2,262
95,000	2,230	2,339	2,256	2,268	2,354
100,000	2,226	2,287	2,306	2,266	2,407

Table A.4: Backgammon results for "smart" representation (non-normalized) and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	28	121	41	216	73
5,000	1,206	982	984	975	810
10,000	1,181	973	402	1,056	869
15,000	1,018	980	1,173	892	784
20,000	1,121	1,057	961	386	761
25,000	620	1,039	1,001	1,034	903
30,000	520	1,143	1,079	984	805
35,000	1,126	1,067	1,061	1,036	532
40,000	1,099	1,049	762	730	755
45,000	1,108	777	1,005	509	707
50,000	1,099	891	892	1,092	633
55,000	1,093	1,211	895	1,028	678
60,000	1,097	772	941	1,189	713
65,000	1,146	810	386	1,194	697
70,000	1,075	1,019	826	1,178	753
75,000	1,079	649	970	1,215	696
80,000	1,131	1,098	882	1,213	698
85,000	1,064	900	928	1,170	714
90,000	1,090	1,077	696	1,127	704
95,000	1,099	922	920	1,177	693
100,000	1,109	991	733	1,248	730

Table A.5: Backgammon results for "smart" representation (normalized) and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	201	237	465	349	273
5,000	1,365	1,398	1,369	1,463	1,420
10,000	1,523	1,427	1,357	1,597	1,431
15,000	1,470	1,340	1,395	1,532	1,372
20,000	1,471	1,453	1,496	1,596	1,482
25,000	1,495	1,516	1,503	1,561	1,509
30,000	1,612	1,509	1,388	1,616	1,352
35,000	1,587	1,566	1,375	1,605	1,559
40,000	1,490	1,647	1,322	1,580	1,677
45,000	1,573	1,532	1,561	1,536	1,655
50,000	1,600	1,554	1,455	1,527	1,623
55,000	1,491	1,576	1,520	1,515	1,603
60,000	1,509	1,690	1,529	1,525	1,640
65,000	1,522	1,527	1,682	1,617	1,693
70,000	1,701	1,683	1,596	1,452	1,605
75,000	1,696	1,699	1,460	1,623	1,591
80,000	1,623	1,642	1,553	1,653	1,617
85,000	1,653	1,617	1,542	1,567	1,599
90,000	1,609	1,669	1,582	1,617	1,657
95,000	1,561	1,655	1,510	1,648	1,600
100,000	1,591	1,807	1,536	1,638	1,631

Table A.6: Backgammon results for combined representation (non-normalized) and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	208	69	2	92	267
5,000	974	849	849	1,017	219
10,000	915	1,168	611	950	760
15,000	1,148	1,088	1,128	1,174	1,175
20,000	856	784	1,264	976	424
25,000	779	630	806	1,270	1,097
30,000	1,028	604	1,033	1,123	1,190
35,000	636	653	937	1,225	1,254
40,000	1,219	614	1,189	1,258	1,369
45,000	1,362	658	1,186	1,248	1,187
50,000	1,421	632	1,230	1,080	1,284
55,000	1,245	620	1,378	1,174	939
60,000	1,299	581	640	1,308	1,405
65,000	1,396	634	1,329	1,356	1,255
70,000	1,312	652	1,452	1,451	1,234
75,000	1,158	637	1,214	1,335	1,052
80,000	1,236	639	1,361	710	1,334
85,000	1,464	657	959	1,411	1,128
90,000	807	609	314	1,472	1,467
95,000	204	645	1,366	153	801
100,000	1,192	609	1,356	858	1,290

Table A.7: Backgammon results for combined representation (normalized) and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	20	106	74	50	128
5,000	993	1,057	1,366	1,562	1,073
10,000	1,759	1,598	1,889	2,020	1,641
15,000	2,047	1,755	1,892	2,148	1,742
20,000	2,259	1,850	2,072	2,065	1,868
25,000	2,288	1,916	2,108	2,291	2,167
30,000	2,247	1,954	2,264	2,391	2,291
35,000	2,282	2,351	2,211	2,216	2,252
40,000	2,410	2,372	2,537	2,420	2,449
45,000	2,359	2,334	2,447	2,557	2,420
50,000	2,494	2,348	2,543	2,433	2,532
55,000	2,525	2,396	2,392	2,411	2,558
60,000	2,415	2,329	2,555	2,614	2,492
65,000	2,539	2,387	2,421	2,510	2,507
70,000	2,486	2,415	2,646	2,596	2,559
75,000	2,598	2,364	2,668	2,479	2,486
80,000	2,425	2,361	2,667	2,632	2,632
85,000	2,622	2,238	2,541	2,505	2,696
90,000	2,508	2,549	2,549	2,643	2,605
95,000	2,681	2,468	2,510	2,542	2,592
100,000	2,575	2,326	2,631	2,557	2,574

Table A.8: Hypergammon results for unary representation and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	1,162	1,226	975	1,378	959
1,000	1,773	1,623	1,769	1,742	1,645
2,000	1,976	1,992	2,053	1,990	1,913
3,000	1,914	2,078	2,083	2,164	2,066
4,000	1,995	2,163	2,057	2,144	2,221
5,000	2,217	2,258	2,112	2,076	2,279
10,000	2,273	2,417	2,483	2,501	2,475
15,000	2,556	2,642	2,680	2,656	2,613
20,000	2,784	2,737	2,659	2,665	2,724
25,000	2,739	2,787	2,816	2,535	2,861
30,000	2,766	2,800	2,766	2,722	2,935
35,000	2,838	2,802	2,888	2,794	2,879
40,000	2,868	2,823	2,916	2,612	2,726
45,000	2,751	2,873	2,897	2,758	2,835
50,000	2,852	2,927	2,956	2,822	2,935

Table A.9: Hypergammon results for unary representation and learning by play against *RandomPlayer*.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	936	1,016	758	799	1,348
1,000	1,724	1,642	1,807	1,701	1,702
2,000	1,940	1,825	1,944	1,811	1,820
3,000	1,841	1,910	1,917	1,853	1,893
4,000	1,960	1,935	1,823	1,884	1,908
5,000	1,881	1,961	1,939	1,863	1,871
10,000	1,897	1,975	1,831	2,022	1,908
15,000	1,922	1,934	1,878	1,945	2,000
20,000	2,081	1,895	1,962	1,998	1,988
25,000	1,918	1,974	1,921	1,933	2,011
30,000	2,045	1,897	1,850	2,123	1,996
35,000	2,087	1,998	1,963	1,935	2,004
40,000	2,131	2,018	1,929	2,081	1,859
45,000	2,040	2,043	2,083	2,021	2,086
50,000	2,082	2,028	2,068	2,007	1,978

Table A.10: Hypergammon results for unary representation and learning by play against *pubeval*.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	1,160	944	993	1,147	1,523
1,000	2,145	2,123	1,986	2,007	1,986
2,000	2,220	2,386	2,281	2,244	2,295
3,000	2,235	2,427	2,188	2,308	2,424
4,000	2,424	2,436	2,488	2,387	2,645
5,000	2,422	2,424	2,560	2,420	2,678
10,000	2,715	2,706	2,751	2,674	2,812
15,000	2,854	2,777	2,744	2,705	3,019
20,000	2,920	2,996	2,792	2,834	2,881
25,000	2,979	2,937	2,898	2,823	2,885
30,000	3,069	3,005	2,970	2,966	2,970
35,000	3,083	2,972	3,049	2,980	2,886
40,000	3,045	2,921	3,043	2,957	2,987
45,000	3,072	3,049	3,003	2,883	2,974
50,000	3,085	3,015	2,997	3,020	2,966

Table A.11: Hypergammon results for “smart” representation and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	546	1,407	2,208	2,150	1,579
1,000	2,580	2,699	2,540	2,469	1,985
2,000	3,021	2,859	2,923	3,020	2,826
3,000	3,029	3,001	3,069	3,117	2,969
4,000	3,147	3,113	3,128	3,092	2,994
5,000	3,138	3,155	3,082	3,168	3,113
10,000	3,229	3,240	3,168	3,158	3,241
15,000	3,239	3,269	3,265	3,205	3,289
20,000	3,223	3,306	3,216	3,286	3,199
25,000	3,148	3,231	3,217	3,246	3,259
30,000	3,209	3,304	3,248	3,277	3,322
35,000	3,247	3,190	3,237	3,209	3,271
40,000	3,384	3,294	3,283	3,237	3,321
45,000	3,207	3,295	3,289	3,236	3,299
50,000	3,230	3,306	3,355	3,211	3,253

Table A.12: Hypergammon results for combined representation and learning by self-play.

	Wins vs. *pubeval*				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	1,495	1,007	1,194	948	1,063
1,000	2,071	2,206	2,242	2,286	2,230
2,000	2,438	2,445	2,530	2,603	2,502
3,000	2,643	2,601	2,685	2,802	2,791
4,000	2,695	2,828	2,859	2,900	2,869
5,000	2,870	2,797	2,912	2,918	2,952
10,000	3,127	3,073	3,224	3,195	3,161
15,000	3,176	3,195	3,273	3,272	3,262
20,000	3,302	3,198	3,274	3,255	3,314
25,000	3,290	3,275	3,276	3,306	3,226
30,000	3,286	3,329	3,299	3,393	3,355
35,000	3,306	3,426	3,362	3,348	3,309
40,000	3,357	3,301	3,377	3,347	3,320
45,000	3,407	3,269	3,435	3,402	3,379
50,000	3,386	3,373	3,374	3,429	3,357

Table A.13: Pachisi results for unary representation and learning by self-play.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	1	3	17	0	4
10,000	107	121	262	117	177
15,000	264	55	219	310	115
20,000	473	335	492	508	580
25,000	521	870	812	653	586
30,000	298	583	212	732	504
35,000	921	984	355	842	872
40,000	1,366	933	357	1,015	927
45,000	1,285	1,063	335	776	569
50,000	1,381	1,068	229	1,354	580

Table A.14: Pachisi results for unary representation and learning by playing against three *RandomPlayer*s.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	2	9	3	4	3
10,000	90	80	44	15	1
15,000	301	52	76	103	22
20,000	298	120	108	135	83
25,000	659	219	218	112	89
30,000	647	155	186	246	121
35,000	739	116	102	182	102
40,000	866	266	131	335	97
45,000	770	212	165	329	113
50,000	733	229	62	280	111

Table A.15: Pachisi results for unary representation and learning by playing against three *HeuristicPachisiPlayer*s.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	35	1	18	2	21
10,000	72	49	206	53	112
15,000	63	156	221	223	194
20,000	153	609	406	405	312
25,000	286	374	498	583	290
30,000	447	862	1,003	891	381
35,000	793	915	818	1,004	452
40,000	805	464	664	1,056	344
45,000	551	710	813	890	538
50,000	580	910	907	1,199	399

Table A.16: Pachisi results for unary representation and learning by playing against a team of *RandomPlayer*s.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	0	0	0	5	0
10,000	0	22	0	5	11
15,000	19	52	1	14	37
20,000	71	92	51	37	39
25,000	69	293	10	94	12
30,000	156	222	7	70	25
35,000	126	575	36	71	134
40,000	113	515	37	66	144
45,000	192	268	71	43	196
50,000	81	293	15	55	287

Table A.17: Pachisi results for unary representation and learning by playing against a team of *HeuristicPachisiPlayer*s.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	8	15	13	23	0
10,000	58	12	11	35	13
15,000	35	112	108	204	31
20,000	48	402	393	159	103
25,000	188	256	271	146	93
30,000	206	470	458	216	336
35,000	500	577	545	327	287
40,000	341	539	510	298	284
45,000	662	449	480	358	469
50,000	556	587	580	297	382

Table A.18: Pachisi results for "smart" representation and learning by self-play.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	1	0	0	0
5,000	2,750	2,778	2,669	2,829	2,751
10,000	2,866	3,125	3,288	2,849	3,093
15,000	3,208	3,019	3,304	2,810	3,322
20,000	2,924	3,041	3,344	3,166	3,374
25,000	3,237	3,081	3,306	3,097	3,288
30,000	3,170	3,294	3,342	3,373	3,250
35,000	3,149	3,123	3,144	3,360	3,431
40,000	3,354	3,172	3,080	3,327	3,542
45,000	3,356	3,145	3,147	3,315	3,465
50,000	3,242	3,352	3,164	3,322	3,151

Table A.19: Pachisi results for combined representation and learning by self-play.

	Wins vs. *HeuristicPachisiPlayer* team				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	0	0	0	0	0
5,000	511	281	271	487	290
10,000	1,176	892	1,900	1,926	1,429
15,000	2,373	2,747	2,818	2,682	1,720
20,000	1,389	2,811	3,304	2,103	1,638
25,000	3,375	3,138	3,439	1,313	3,250
30,000	2,371	2,664	3,709	2,188	1,759
35,000	2,869	2,773	3,584	3,120	3,175
40,000	2,018	2,950	3,633	3,057	2,267
45,000	2,519	1,636	3,661	3,481	3,517
50,000	2,874	2,448	3,860	3,002	3,787

Table A.20: Parcheesi results for unary representation and learning by self-play.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	45	40	42	82	46
5,000	536	539	618	510	453
10,000	798	988	1,164	1,010	952
15,000	1,062	1,193	1,307	1,315	1,194
20,000	1,264	1,259	1,362	1,415	1,488
25,000	1,323	1,419	1,456	1,489	1,416
30,000	1,400	1,338	1,613	1,694	1,564
35,000	1,492	1,571	1,648	1,812	1,597
40,000	1,385	1,606	1,560	1,785	1,594
45,000	1,577	1,680	1,653	1,759	1,718
50,000	1,672	1,714	1,642	1,806	1,675

Table A.21: Parcheesi results for unary representation and learning by play against 3 *RandomPlayer*s.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	49	41	60	38	60
5,000	819	813	559	819	819
10,000	1,181	1,270	1,186	1,201	1,261
15,000	1,229	1,301	1,104	1,121	1,264
20,000	1,198	1,277	1,213	1,226	1,297
25,000	1,193	1,329	1,174	1,201	1,281
30,000	1,244	1,227	1,296	1,257	1,184
35,000	1,234	1,357	1,240	1,217	1,288
40,000	1,345	1,224	1,221	1,311	1,239
45,000	1,185	1,279	1,323	1,266	1,274
50,000	1,341	1,258	1,309	1,321	1,277

Table A.22: Parcheesi results for unary representation and learning by play against 3 *HeuristicParcheesiPlayer*s.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	59	53	42	42	72
5,000	628	706	786	839	838
10,000	1,192	1,207	1,265	1,291	1,313
15,000	1,261	1,383	1,321	1,487	1,540
20,000	1,444	1,593	1,420	1,403	1,527
25,000	1,503	1,573	1,453	1,481	1,684
30,000	1,679	1,585	1,570	1,583	1,697
35,000	1,715	1,704	1,533	1,584	1,716
40,000	1,671	1,708	1,568	1,631	1,767
45,000	1,586	1,640	1,617	1,616	1,663
50,000	1,702	1,689	1,643	1,641	1,729

Table A.23: Parcheesi results for unary representation and learning by play with self and 2 *RandomPlayer*s.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	76	55	38	44	49
5,000	661	829	860	785	810
10,000	1,523	1,413	1,492	1,200	1,441
15,000	1,523	1,492	1,541	1,501	1,577
20,000	1,566	1,583	1,563	1,582	1,681
25,000	1,622	1,776	1,585	1,653	1,645
30,000	1,654	1,592	1,579	1,685	1,697
35,000	1,597	1,652	1,709	1,688	1,696
40,000	1,607	1,707	1,705	1,785	1,680
45,000	1,755	1,654	1,777	1,731	1,739
50,000	1,574	1,687	1,756	1,654	1,713

Table A.24: Parcheesi results for unary representation and learning by play with self and 2 *HeuristicParcheesiPlayer*s.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	69	45	59	27	58
5,000	984	831	1,084	721	864
10,000	1,284	1,291	1,330	1,111	1,248
15,000	1,457	1,396	1,490	1,276	1,421
20,000	1,437	1,463	1,592	1,342	1,399
25,000	1,610	1,528	1,766	1,472	1,497
30,000	1,687	1,646	1,766	1,309	1,627
35,000	1,736	1,833	1,742	1,461	1,610
40,000	1,783	1,791	1,758	1,455	1,577
45,000	1,778	1,816	1,874	1,550	1,627
50,000	1,783	1,799	1,758	1,388	1,705

Table A.25: Parcheesi results for unary representation and learning by play with self, *RandomPlayer* and *HeuristicParcheesiPlayer*.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	64	42	71	35	50
5,000	1,058	1,090	1,090	983	1,012
10,000	1,496	1,565	1,471	1,491	1,555
15,000	1,630	1,731	1,703	1,655	1,619
20,000	1,753	1,703	1,753	1,775	1,763
25,000	1,680	1,740	1,712	1,728	1,726
30,000	1,802	1,760	1,738	1,733	1,692
35,000	1,827	1,774	1,837	1,682	1,787
40,000	1,810	1,740	1,790	1,779	1,799
45,000	1,827	1,793	1,810	1,824	1,755
50,000	1,785	1,871	1,828	1,741	1,844

Table A.26: Parcheesi results for unary representation and learning by observation of 2 *RandomPlayer*s and 2 *HeuristicParcheesiPlayer*s.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	54	86	63	72	82
5,000	1,308	1,304	1,190	1,344	1,193
10,000	1,569	1,529	1,317	1,595	1,617
15,000	1,647	1,664	1,581	1,685	1,575
20,000	1,734	1,727	1,679	1,646	1,616
25,000	1,628	1,699	1,556	1,723	1,733
30,000	1,728	1,724	1,651	1,730	1,702
35,000	1,726	1,739	1,658	1,658	1,721
40,000	1,735	1,749	1,776	1,711	1,689
45,000	1,711	1,735	1,754	1,729	1,680
50,000	1,779	1,809	1,692	1,714	

Table A.27: Parcheesi results for "smart" representation and learning by self-play.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	203	132	352	97	339
5,000	1,380	1,422	808	1,413	1,439
10,000	1,408	1,544	842	1,533	1,459
15,000	1,442	1,565	1,433	1,435	1,600
20,000	1,616	1,506	1,549	1,474	1,549
25,000	1,503	1,537	1,455	1,521	1,561
30,000	1,484	1,524	1,538	1,488	1,517
35,000	1,578	1,583	1,577	1,567	1,582
40,000	1,508	1,596	1,519	1,582	1,462
45,000	1,347	1,565	1,425	1,577	1,432
50,000	1,441	1,597	1,486	1,560	1,506

Table A.28: Parcheesi results for combined representation and learning by self-play.

	Wins vs. 3 *HeuristicParcheesiPlayer*s				
Games trained	Run 1	Run 2	Run 3	Run 4	Run 5
0	59	51	31	45	92
5,000	834	907	726	906	947
10,000	1,360	1,410	1,322	1,375	1,476
15,000	1,529	1,684	1,715	1,879	1,523
20,000	1,755	1,593	1,563	1,974	1,861
25,000	1,836	1,704	1,682	1,959	1,946
30,000	1,898	1,974	1,915	1,952	2,108
35,000	1,935	2,139	1,959	2,087	2,132
40,000	1,910	2,083	1,770	2,152	2,208
45,000	1,924	2,110	2,033	2,129	2,247
50,000	2,029	2,094	2,003	2,054	2,127