A Reinforcement Learner for Point-Blank

Nathan Sitkoff

22 December 1995

1 Point-Blank

Point-Blank is a board game in which players take turns placing pieces on a square grid. The players begin by placing pieces at the opposing corners of the board. Each succeeding move must be made to an empty cell which is adjacent (horizontally, vertically, or diagonally) to the last occupied cell. In this way, each player "grows" a path, snakelike, through the grid. The game ends when neither player can make a move. The winner is the player who occupies the majority of the grid at the end of the game. Figure 1 shows a few opening moves on a 4x4 grid, and the endgame.

In general, this game can be played on any size gridworld, but only 4x4 and 8x8 worlds were examined for this project. In addition, games can be played either on "bounded" boards, in which the edges of the grid cannot be crossed, or on "torus" worlds, in which moves leading off the grid wraparound to the other side. Learning methods were applied to both types of environments.

The key to winning Point-Blank is to trap your opponent in a small section of the grid. The opponent quickly fills up this small space and then is forced to stop, with no more legal moves. This leaves you free to occupy all the remaining cells, which should constitute the majority. The more quickly you trap your opponent, the more cells you collect, and the greater the margin of victory.

2 Reinforcement Learning Methods

Phrasing Point-Blank as an RL problem is relatively straightforward. Each grid configuration is an exact representation of the state, and there are a finite number of possible states. Therefore it meets the requirements of a finite Markov Decision Process (MDP) [4]. An exact model of the environment is available, so for every possible action, the resulting state can be calculated. Therefore it is sufficient to learn a value for each state (ie a V-function), and choose actions which give rise to states with desirable values.

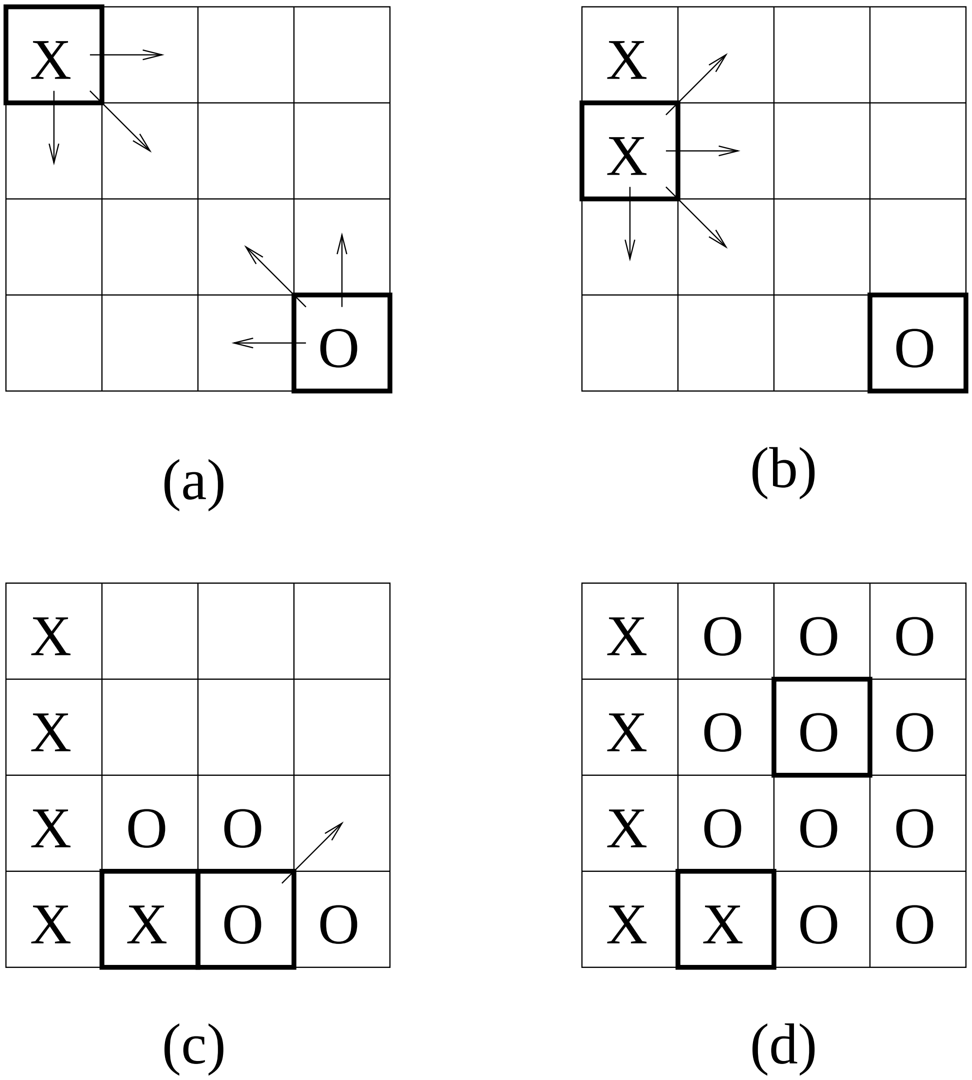

Figure 1: Point Blank: (a) Game begins. X's indicate a cell occupied by Player 1, O's by Player 2. Arrows indicate the legal moves available to each player. (b) After Player 1's first move. (c) Player 2 has "trapped" Player 1 in the lower left corner. (d) Player 2 fills up the remaining cells, winning the game. This endgame has fixed value of -6.

As in Tesauro's TD-Gammon [3], the reinforcement learner was trained against itself. Instead of creating a system of rewards, all endgames are assigned a fixed value, and these values are backed up using off-policy temporal difference learning (TD) [1]. No eligibility traces were used (ie TD(0)). Because all state transitions are deterministic, the step size parameter ("alpha") was fixed at 1. Since the goal is to arrive at the best solution (not the quickest one) no discounting was used. Exploration during training was controlled by softmax action selection, with a scheduled temperature.

Several different methods were used to encode the state information and store the value function, described below.

2.1 Bounded Boards

Bounded boards are encoded using a fixed frame of reference, aligned with the grid boundaries. Every cell can take on five values: empty, occupied by player 1, occupied by player 2, location of player 1, and location of player 2. The value of an endgame is calculated as follows:

$$V(B) = N(B,1) - N(B,2)$$

where N(B,1) is the number of cells occupied by player 1, and N(B,2) is the number of cells occupied by player 2.

The value function assigns positive values to states which will lead to wins by Player 1, and negative values to states leading to wins by player 2. Both players use the same value function to train and to play, but Player 1 is "positive-seeking," choosing actions that lead to states of highest value, while Player 2 is "negative-seeking," choosing actions leading to states of lowest value.

All state values are stored in a hash table. States not present in the table are assumed to have value zero (ie draw). This is a convenient assumption, because it means that zero-valued states do not require table entries.

2.2 Torus Boards

Because the boundaries of the grid have no significance on the torus board, an egocentric frame of reference is used to encode the state. Each player re-arranges the the board so as to place its current position in the upper left corner of the grid. In addition, Player 2 actually rotates the entire board 180 degrees, and then inverts the state of every cell, so that cells occupied by Player 1 appear to be occupied by Player 2 and vice versa.

The result of all these manipulations is that the state is no longer encoded as player 1 versus player 2, but rather as current player versus opponent. Every cell can take on one of four values: empty, occupied by current player, occupied by opponent, and location of opponent. Since the same cell always holds the location of the current player, that cell need not be represented.

Board type	Encoding	V-function
4x4 Bounded	fixed	hash table
4x4 Torus	mobility	backprop net (I=14;H=30;O=1)
4x4 Torus	egocentric	backprop net (I=60;H=30;O=1)
4x4 Torus	ego + mobility	backprop net (I=74;H=30;O=1)
8x8 Torus	mobility	backprop net (I=14;H=30;O=1)
8x8 Torus	ego + mobility	backprop net (I=266;H=40;O=1)

Table 1: V-function encoding methods

Instead of calculating endgame values based on the margin of victory, endgames were scored as either a win by the current player, a draw, or a win by the opponent. Each TD backup is performed by generating all the possible next states, and then evaluating each state *with respect to the current player on the next turn.* Typically, that means looking at things from the opponent's point of view. Obviously, states which are good for the opponent are bad for the current player, so state values for the opponent are negated. Finally, the maximum state value is backed up.

The state space for the torus board is so large as to make even a hash table infeasable. Therefore, state values were stored in a backprop network, with an expansize input representation of four binary input units per cell, representing the four possible values listed above. Exactly one out of every four input units are on for every board configuration. Testing was performed with this "raw" encoding, and also with units representing a special feature known as mobility, described below.

Table 1 summarizes the methods used to encode the states and store the V-function. The sizes of the backprop networks used (inputs; hidden; outputs) are also shown.

2.3 Mobility Heuristic

A simple heuristic strategy was devised for comparison with the learning methods described above. The heuristic is an evaluation function based on the mobility of each player. Mobility is defined as the number of legal moves available to a given player; it varies between 0 and 7. The heuristic value for a board configuration is given as:

$$H(B) = (M(B,1) - M(B,2))/(M(B,1) + M(B,2))$$

where M(B,1) is the mobility of player 1 on the given board, and M(B,2) is the mobility of player 2. These heuristic values were used to select actions in the same way as the RL values.

Mobility was also used as a handcrafted feature input to the backprop network. 14 binary units (7 per player) encode the mobilities. If a player has a mobility of zero, all 7 units are off. If the mobility is one, then only the first unit is on; if it is two, then both the first unit and the second unit are on. If the mobility is seven, all seven units are on.

3 Results

Results are discussed in the sections below. Performance is typically measured by playing games against a random player.

Note: As it turns out, there is a significant advantage to having the first move. In order to remove this bias, half of the testing games always begin with player 1, and the other half with player 2.

3.1 Bounded boards

Figure 2(a) shows the progress of RL for a 4x4 bounded board, over 200,000 training games. The softmax action selection temperature decays exponentially over this time. After every 1000 games, the RL player is tested against a random player, to measure performance. Two curves are shown: the wins scored by the RL player, and the wins scored by the random player.

The three horizontal lines which divide the graph represent the performance of the heuristic player against the random player, and the random player against itself. Note that the percent wins and percent losses always sum to less than unity; the balance of the games are draws. The RL player exceeds heuristic performance after about 20,000 training games. After 150,000 games it never loses to the random player.

Figure 2(b) compares the performance of the trained RL player to the heuristic and random players. As before, the RL player and the heuristic player are compared to the random player, and the random player is compared to itself. Each game is scored according to the value function in (2.1), where the player being tested is player 1, and player 2 is the random opponent. Positive values indicate wins, negative values losses, and zeros are draws. Note that fewer games have odd valued scores than even; in order to achieve an odd score, one or more cells must be unoccupied at the end of the game, which is somewhat unlikely on a 4x4 board.

Each player was tested for 10,000 games, which were then grouped by score and plotted. As expected, the random player shows a symmetric distribution when tested against itself, with a peak at zero. The heuristic player wins more than it loses, but usually manages a win only by a small margin (1 cell). The RL player never loses, and often wins by a large margin (6 cells). This is to be expected, since the V-function is based on the margin of victory in the endgame.

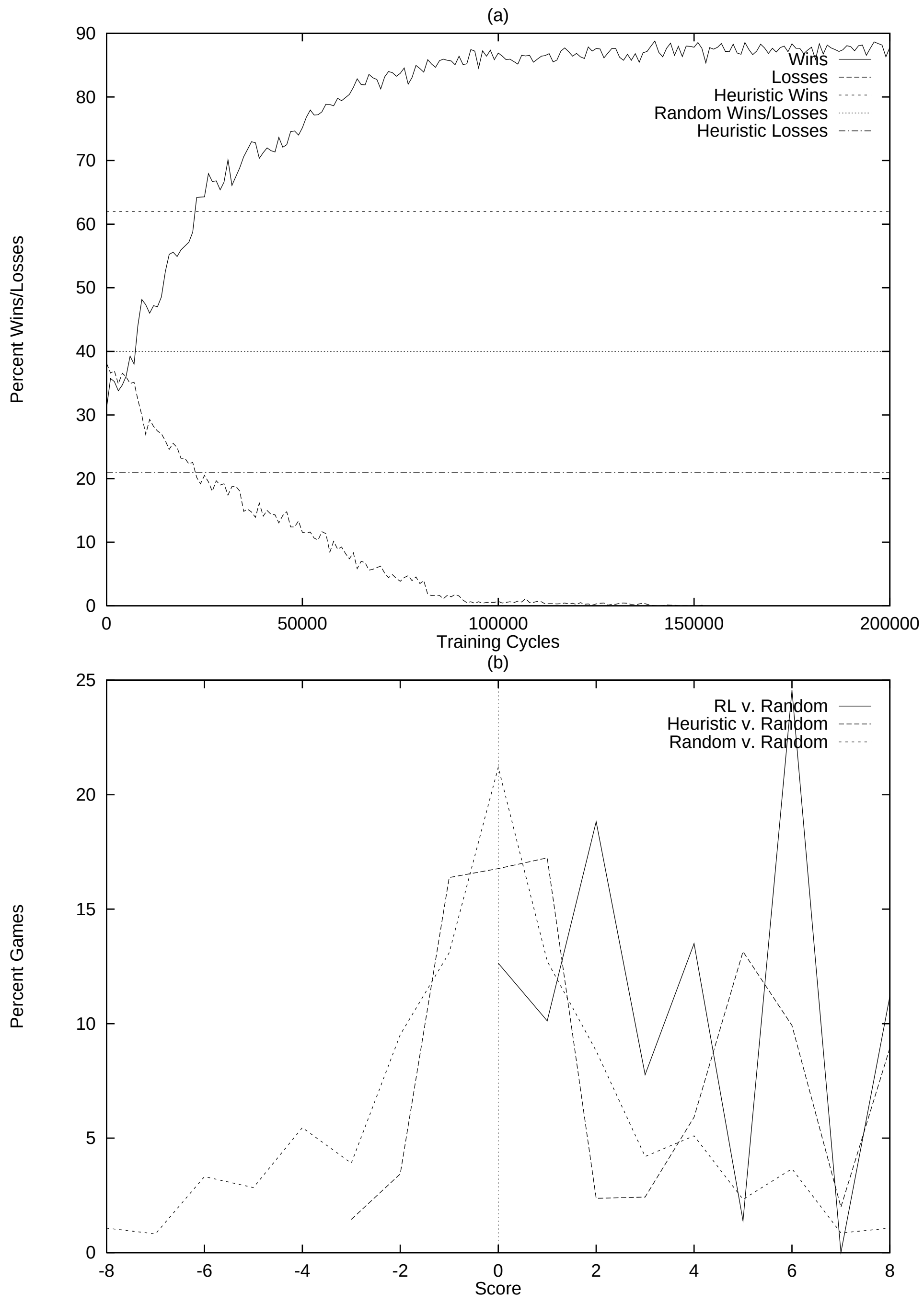

Figure 2: 4x4 bounded board: (a) training data, and (b) win/loss margins for random, heuristic and RL players

3.2 Torus boards

Torus boards turn out to be a far more difficult environment for training. The branching factor is much larger, and so is the state space. Efforts to use a hash table representation for a 4x4 torus board required hundreds of thousands of training games before the performance began to rise significantly better than random. A backprop net was chosen to replace the hash table, in order to speed up learning by generalization.

Figure 3 contains two plots of training data for the 4x4 torus board, over 100,000 training games. This is the same kind of graph as Figure 2, described above. Figure 3(a) plots the learning curves for a backprop net with a raw input encoding. The performance improves dramatically after the first 1000 training cycles, but there is no significant further improvement. The win percentage never reaches that of the heuristic.

Figure 3(b) plots learning curves for a backprop net with the 14 additional mobility units. Once again, the performance shoots up immediately; in this case it actually slightly exceeds heuristic levels. After the initial jump, the performance oscillates for the rest of the training set.

In both plots, the wins for the RL player actually seem to fall off slightly at the end of the training run. Why this should occur is not entirely clear, but it may be due to overfitting. As training progresses, the softmax action selection temperature cools, reducing the degree of exploration. If the temperature becomes low enough, the RL player might always make the exact same sequence of moves. Under conditions like these, backprop nets are known to overfit the data; that is, the net learns the repeated pattern very well, but "forgets" more general knowledge it learned earlier. It should be noted, however, that while the number of wins is decreasing, so is the number of losses; so in a sense, the performance is unchanged.

It is interesting that the addition of the 14 mobility input units results in performance so close to the level of the mobility heuristic. The obvious question to ask is: does the backprop net simply learn the mobility function? One way to answer this question is to perform the training run again, but this time using *only* the 14 mobility input units to represent the situation (rather than the state). The number of hidden units was held constant.

The results are shown in Figure 4(a). Performance seems to be almost exactly that of the mobility heuristic, and only slightly less than that of the combined encoding. Figure 4(b) shows training results for the same mobility-only net net, but this time for the 8x8 torus world. Due to the increased length of each game, training times are far longer, and so fewer training cycles were used. Performance actually seems to slightly exceed heuristic levels for the 8x8 torus.

Since these two nets have the same number of input and hidden units, their cross-performance can be analyzed: how well does the 4x4 trained net operate in the 8x8 environment, and vice-versa? It turns out that both nets do rather

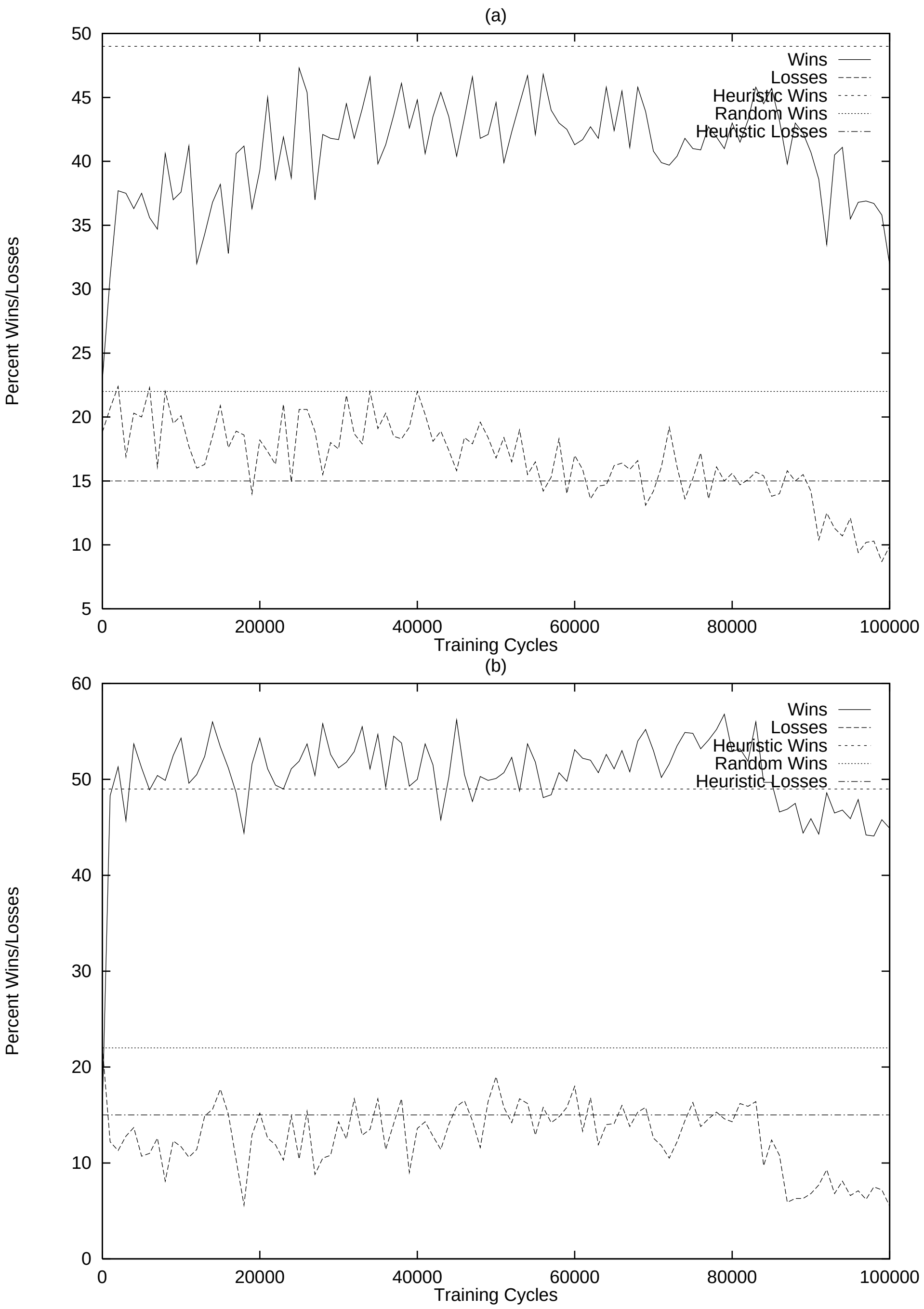

Figure 3: Training data for the 4x4 torus board: (a) With raw inputs (60 input units). (b) With raw inputs and mobility features (74 input units). 30 hidden units were used in both runs.

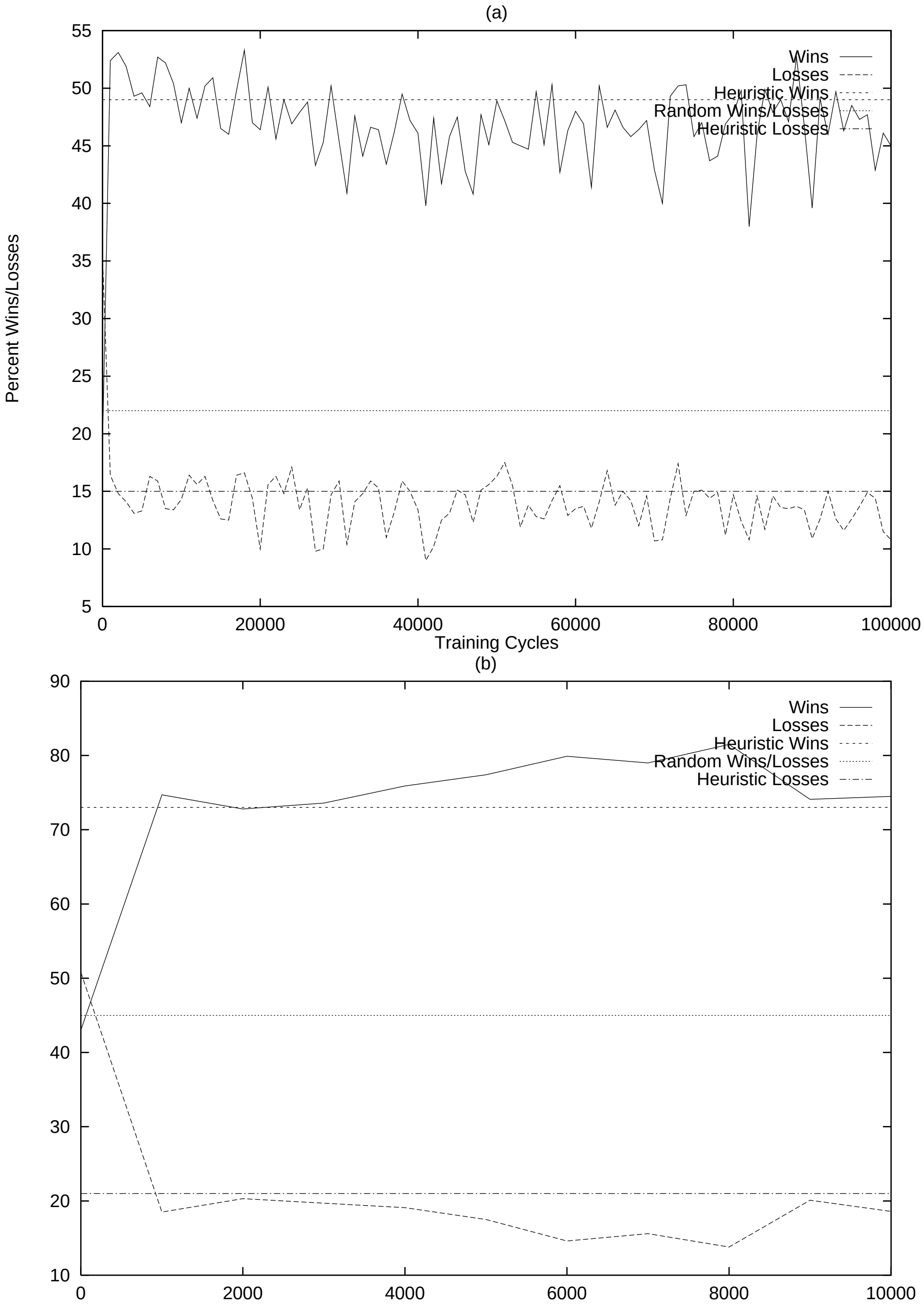

Figure 4: Training data with mobility features *only*, (a) for the 4x4 torus world, and (b) for the 8x8 torus world. 14 input units and 30 hidden units were used in both runs.

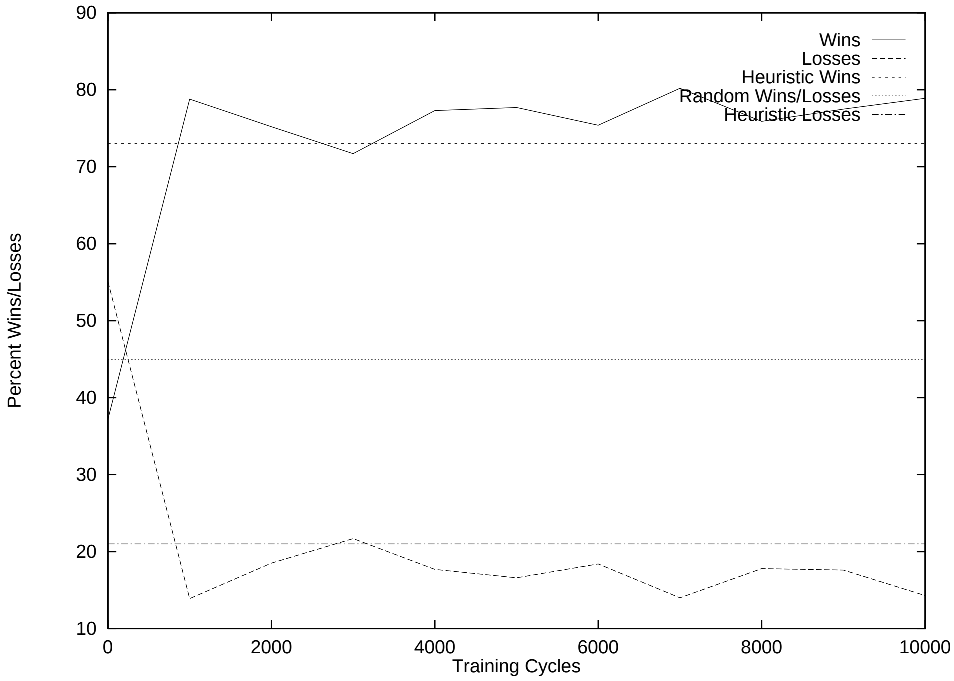

Figure 5: Training data for the 8x8 torus board, with raw inputs and mobility features (266 input units and 40 hidden units).

well, performing at about the heuristic level. This would suggest that the nets really are learning something very similar to the mobility heuristic function.

The last results are for the 8x8 torus board, using a full input representation, including both raw and mobility units. Unfortunately, this makes for an enormous net, with 266 input units and 40 hidden units. Backprop required so much time to run that the result obtained were fairly limited. Figure 5 shows the performance curves over a period of 10000 trials. The results are promising, but inconclusive.

4 Conclusions

The results show maximum performance for the 4x4 bounded world. This is not surprising, since the hash table represents the value function exactly, and the state space is fairly limited. The learning curve has the initial jump which is characteristic of RL, and performance continues to improve asymptotically.

The step from the 4x4 bounded world to the 4x4 torus world is a large one. It was very difficult just to get the RL player to perform better than random; many different state encodings were tried, with both hash tables and neural networks. One explanation for this difficulty is the large branching factor of the torus world. Another is the existence of a strategy; is there anything to be learned? For the 4x4 bounded board, a definite strategy exists; trap the opponent in a corner and fill the rest of the cells. On the 4x4 torus board, trapping the opponent is extremely difficult. Even when a human plays a random player, most games end in draws! Winning by a margin of more than two is all but impossible. Since a draw can be reached from almost every state, nearly all states have value zero, except for those just before an endgame. In a sense, no strategy exists.

For the 8x8 torus board, there is no doubt that a strategy exists. A human beats a random player virtually every game, usually by a very large margin. The difference is the room to maneuver on the 8x8 torus, which is lacking in the 4x4 torus. This is really the interesting domain, but unfortunately requires exponentially more time to obtain results, since the games are longer and the backprop net larger.

Another area that deserves more attention is trying to determine the relative importance of the data that make up the raw state encoding. If the size of the input vector is reduced, so is the size of the situation space (and that of the backprop net). The success of the mobility heuristic suggests that the cells in the immediate vicinity of each player location are far more important than any others. By constructing a situation based on these cells, an RL player could be trained which is indifferent to the exact size and nature of the board.

5 References

References

[1] Barto and Sutton. *Reinforcement Learning.* (unpublished)

[2] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers." *Computers and Thought.* McGraw-Hill: New York, 1963.

[3] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM,* March 1995 (pp. 58-68).

[4] Douglas J. White. "Real Applications of Markov Decision Processes." *Interfaces* 15: 6 November-December 1985 (pp. 73-83).