Over the Hill: An Evolutionary Approach to Checkers

Luke Horgan
horgan.l@husky.neu.edu

Abstract

TD-Gammon, a computer program developed by Gerald Tesauro at IBM in 1992, famously managed to play competitively against some of the world's top backgammon experts using a machine learning strategy known as temporal difference (TD) learning. Pollack and Blair demonstrated that much of TD-Gammon's success could be replicated using a somewhat simpler "hill-climbing" strategy, indicating that the favorable "dynamics" of backgammon, rather than the novelty of Tesauro's strategy, were at least partially responsible for the program's remarkable performance. In this paper, we test Pollack and Blair's conclusion by applying their methodology to another ancient game, checkers, which lacks many of backgammon's supposedly agreeable features. Although our program enjoys a measure of qualified success, its performance might (charitably) be characterized as mediocre, supporting the notion that there is indeed something special about backgammon.

Introduction

Checkers is familiar territory for computer scientists; it has been closely studied by some of the field's top minds for over half a century. Arthur Lee Samuel, who also coined the phrase "machine learning", created one of the first successful computer checkers programs in the late 1950s. Since then, checkers has even been "solved" using copious computing power and a bit of human elbow grease and ingenuity. A program known as Chinook, developed by (presumably freezing) researchers at the University of Alberta, will always make a provably optimal move against an opponent who is also playing optimally.

There are several properties of checkers which are typically viewed as beneficial for its use as a tool to study artificial intelligence algorithms. In particular, it is deterministic and zero-sum. A stochastic game is one in which there is an element of random chance which the players cannot influence, such as a pair of dice that must be rolled to determine a legal move. Monopoly is such a game, as is backgammon. Checkers, then, is deterministic in the sense that it is not stochastic. A zero-sum game is simply one in which net wins are exactly balanced by net losses. Monopoly is not a zero-sum game; one player wins and all the rest lose (and probably also stop talking to each other). Checkers and backgammon are zero-sum games; they end with a single winner and a single loser. Note that checkers may additionally end with a draw.

Problem Statement

Backgammon, as mentioned, was tackled handily by Tesauro's 1992 program. Pollack and Blair were able to achieve similar results by using a modified strategy, the success of which they credited to the structure of backgammon itself. We sought to adapt the core features of Pollack and Blair's approach for use in checkers, thereby offering some additional insight into just how favorable backgammon really is to machine learning methods.

Approach

We began by creating a generic checkers environment that would allow us to easily "drop in" various artificial intelligence techniques without substantially modifying the checkers logic. This starter code includes an easily configurable virtual board and checkers that are able to efficiently determine all of their legal moves. Absent is a hard-coded "evaluation function", such as the one employed by Arthur Samuel Lee's historic 1959 program. Instead of telling our program how to

evaluate a particular board state, we instead allowed it to learn an evaluation function through tens of thousands of rounds of self-play, in accord with the methods outlined by Pollack and Blair (and, in turn, Tesauro himself). Throughout the rest of this paper, we will refer to the two players in the game as green (player 1) and red (player 2).

To create this evaluation function, we used a *neural network* comprised of 512 binary input nodes, two *hidden layers* of 10 nodes each, and a single output node whose value was used directly to assess the strength of a particular board state.

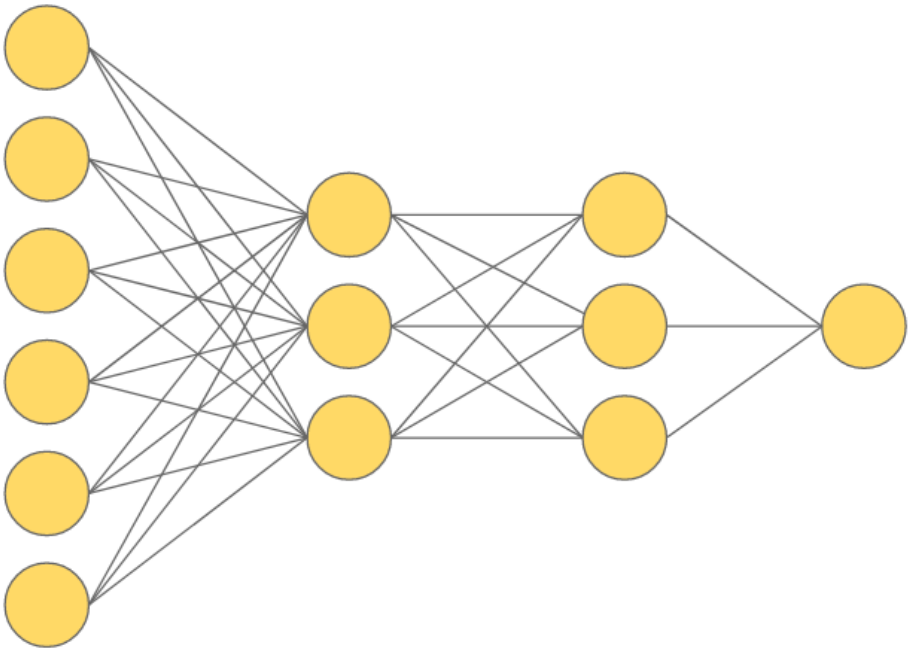

Figure 1: Simplified network structure. In reality, there are 512 input nodes, two layers of 10 hidden nodes, and a single output node (as shown).

The output value of a single node in the network is calculated as follows:

1) $x_o = \frac{1}{1+e^{-z}}$ (sigmoid function)

where z is defined as:

2) $z = \sum_i w_i x_i + b$

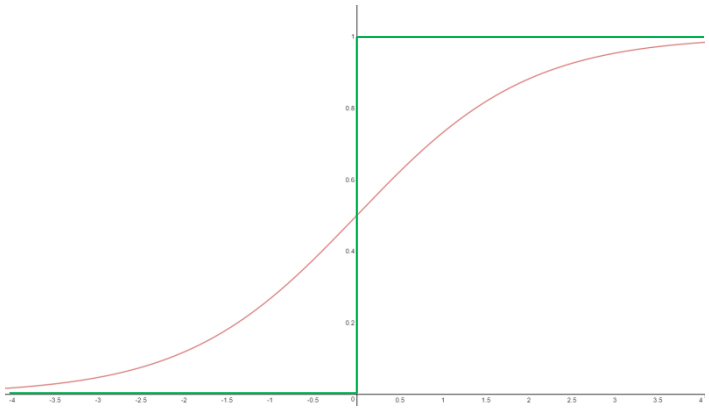

Figure 2: Unit step function plotted against the sigmoid function

This definition of x_o allows us to fine tune the network by adjusting the weights; no single change will drastically change the network's overall behavior. Indeed, x_o may be thought of as a smoothed out version of the unit step function that is a bit less given to spontaneous mood swings.

To calculate the value of the single output node, equation 1 is simply "fed-forward" through the network, node by node and layer by layer, until the output node is reached. Note that output node is a function of the 512 input nodes, which are calculated directly from the board state. The hope, then, is that we can find weights and biases for the network such that the encoding of a good board state yields a higher value for the output node than a bad one. Originally, the encoding used was a nearly direct translation of the board configuration. The board was flattened out into two 64-dimensional vectors. In the first, an entry was "on" (1) if there was a green checker in its corresponding square, and "off" otherwise. The second followed the same scheme, but entries were treated as on for red checkers instead. The input layer was the concatenation of these two vectors.

This approach is pure in the sense that the evaluation function really has no pre-programmed knowledge of the game of checkers outside the rules for movement. Unfortunately, it would have taken a very long time to determine if there was any promise in this approach because the network's performance improved very slowly, if at all. To move things along, three additional meta-features were added for each space on the board, resulting in two 256-dimensional vectors and 512 input nodes total. The second component for each space encodes whether or not the checker in that space is directly adjacent (on the diagonal) to an enemy checker that could capture it if it was undefended. Fittingly, then, the third component indicates whether or not the space is either defended by a colleague of the same color or is "safe" by virtue of being along an edge. The fourth and final component simply encodes whether or not the checker in the space is a king. Note that each of these three binary auxiliary components are off if their parent component is off (i.e. there is no checker of the requisite color in the corresponding board space).

To train the network, we settled on a slightly modified version of the hill-climbing algorithm used by Tesauro and Pollack. Initially, all weights in the network were specified as 0. This network was then taken and "perturbed" slightly by introducing Gaussian noise to each weight and bias. This so-called mutant network, which we call the rookie because we don't enjoy having nightmares, was then played against the reigning incumbent (initially the 0-weight network), which we call the champ. If the rookie (C) beats the champ (R), then the champ is modified by dotting its weights with the rookie's according to the following update:

3) $C' \leftarrow 0.9C \cdot 0.1R$

C' is then the base network for the subsequent iteration, and is played once more against a mutant version of itself. This process repeats over and over ad infinitum, the idea being that eventually the weights in the network will become optimal. Note that weights were normalized using an algorithm suggested by Oon and Henz (4). To see an implementation of this algorithm, refer to network.py in the source code.

Experimental Setup and Data

To test the efficacy of this approach, the network was first played against a random baseline at various stages of its evolution.

	Depth 1	Depth 2	Depth 3
Iteration 0	6	5	5
Iteration 1	5	6	6
Iteration 1000	6	10	10

Iteration 5000	10	10	10
Iteration 10000	9	10	9
Iteration 20000	9	9	10
Iteration 30000	9	9	10
Iteration 40000	10	10	10
Iteration 50000	10	10	10

Table 1: The number of wins each network scored against a random baseline in trials of 10.

Each entry in the table shows the number of wins that the network player scored against the random player in a round of 10 games (trials) for a given "look-ahead" depth. Look-ahead is a technique in which more than a single ply is factored into the calculation for a player's move. For depth 1 search, only the current player's immediate moves were considered; the move that resulted in the best board state as scored by our evaluation function was chosen. For depth 2 search, all of our potential moves were considered against all of our opponent's potential responses. For depth 3 search, each potential response was in turn considered against our own potential re-responses.

It is plain to see that the search tree grows rapidly as the depth increases, so a technique known as alpha-beta pruning was employed to cull search branches that cannot possibly be optimal (according to our suboptimal evaluation function). Note that although the code was written to allow search up to an arbitrary depth, it is not the well-established efficacy of search that we are questioning. Rather, search is a tool that we used to investigate the behavior of our networks, and there is limited additional illustrative value in doing a 10-ply look-ahead, but very substantial additional computation.

Next, we played the networks against themselves to test their improvement over time. The results of these trials are shown in the table below. Each cell indicates the result of 10 trials. We designate the row labels as Network 1 and the column labels at Network 2. Thus, the leftmost cell in the first row indicates that, when both Network 1 and Network 2 were the initial 0-weight network, Network 1 won 6 times out of 10. When Network 1 was iteration 0 and Network 2 was iteration 1000, Network 1 won 4 times out of 10.

	0	1000	5000	10000	20000	30000	40000	50000
0	6	4	0	1	1	1	0	0
1000	6	5	3	4	6	8	2	7
5000	10	7	6	5	7	5	6	7
10000	9	6	5	4	6	4	2	8
20000	9	4	3	4	5	6	5	6
30000	9	2	5	6	4	4	4	4
40000	10	8	4	8	5	6	6	7
50000	10	3	3	2	4	6	3	5

Table 2: The number of wins each network in the set scored against each other network in the set (including itself) in trials of 10. The transpose ${C_{ij}}^T$ of each cell C_{ij} is ${C_{ij}}^T = 10 - C_{ij}$.

In all trials for both experiments, the treatments (network 1, network 2, or random play) were randomly assigned to the red and green agents to account for any bias that might result from green having the first move.

Analysis and Discussion

At a glance, the results summarized in Table 1 are certainly encouraging. The initial Iteration 0 (I0) network is no better than random. In fact, its behavior *is* random, since it assigns equal weight (0) to all board states. Unsurprisingly, the I1 network is comparably bad. The I1000 network,

however, shows a marked improvement over the random baseline. With depth-1 look-ahead, it wins 6 out of 10 games against a random agent. However, with 2 and 3 look-ahead, it wins *every* match against the random agent. This is an especially exciting result because it indicates that there is a certain measure of generality to the evaluation function that is being learned. It was trained by agents playing each other with depth-1 look-ahead. However, because it is a general fitness function, it can be used in tandem with conventional techniques like depth-first search and alpha-beta pruning.

Beyond I1000, the network agent beats the random agent almost without exception, though the existence of any exceptions at all is certainly significant. A world-class checkers program, or even a prodigious toddler, will easily defeat a random agent without fail. Moreover, the fact that the random agent was still winning (occasionally) even after 30,000 iterations begs the question of just how quickly the networks are improving. Does the perfect performance of the I40000 and the I50000 networks against the random agent suggest that they are strictly superior to the I20000 and I30000 networks?

The answer, disappointingly, is no. Up to 5000 iterations, there is a clear trend of improvement. Facing every other network (I0, I1000, I5000, I10000, I20000, I30000, I40000, I50000) and itself, I5000 wins an average of 6.625 games out of 10, against I1000's 5.125 games and I0's 1.625 games. After that, however, the network stagnates, oscillating between slight improvements and slight declines. Performance peaks at I40000, with an average of 6.75 wins out of 10, but then bottoms out at I50000 (the most advanced network), with only 4.5 wins out of 10.

Iteration	Average Wins
0	1.625
1000	5.125
5000	6.625
10000	5.5
20000	5.25
30000	4.75
40000	6.75
50000	4.5

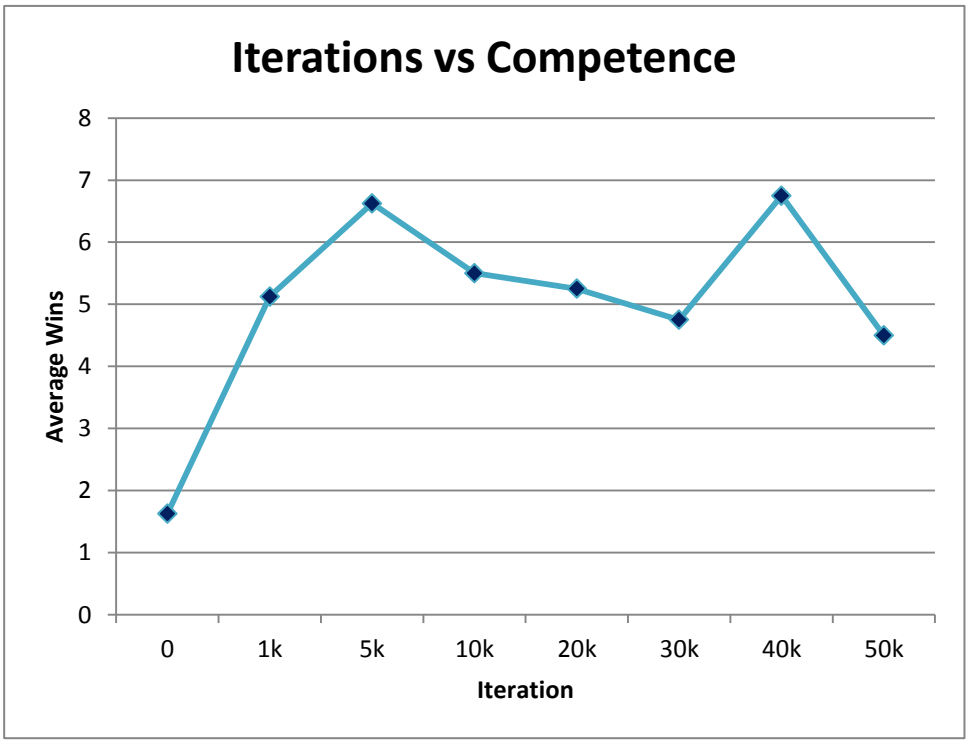

Table 3, Figure 3: The average number of wins for a given network (out of 10) when it was played against each of the other networks in the set (including itself).

So, what's going on here? Why do we stop improving? To see, let's start by looking at one of the ways in which our setup differs from Pollack and Blair's. They suggest conducting matches between champs and rookies in rounds of four. That is, to have its weights incorporated, a rookie must defeat the champ in three out of four rounds played between them. The naive thing to do would be to simply replace the champ with the rookie each time the rookie wins; taking the weighted average somewhat mitigates the risk of a lucky rookie that really isn't very good ruining

all of our progress. Additionally requiring three wins out of four serves as a further safeguard, but also presents a number of logistical headaches that begin to expose the reason this approach is so well suited to backgammon, and decidedly less well suited to checkers.

Backgammon never ends in a draw. Checkers can hypothetically end in a draw if both players agree that a decisive win is unlikely. If each player has only one piece remaining on the board, it is easy to imagine the game continuing indefinitely. This rarely happens in practice *between people*. Our checkers program ran into this situation *constantly*. The rookie is just a slight random perturbation of the champ, so both play a very similar game. An average checkers game lasts around 50 moves; ours frequently lasted hundreds or even thousands. To prevent them from continuing *forever*, an epsilon-greedy strategy was employed. During training, one in every twenty moves, on average, was random. If a board state was repeatedly detected, epsilon was increased to prompt exploration. Additionally, games that lasted more than 200-ply were declared to be over, assigning a win to the player with more remaining pieces, because past that point, most moves were either repetitive or random anyway. If both players were in the possession of the same number of pieces after 200-ply, then the game was counted as a draw (effectively a loss for the rookie).

Requiring 3 *decisive* wins out of four meant that the network only mutated about 20% of the time, forcing us to go through about 5 iterations of 4 games each to advance a single *generation.* Assuming a binomial distribution, where the rookie has a 50% chance of winning, one might figure that the rookie has roughly a 1 in 3 chance (31.25% to be precise) of emerging victorious from a 4-round match. In fact, after 5,000 iterations with a draw-cutoff of 2,000-ply, this was almost exactly the case, suggesting that random mutations do indeed yield random results. They are, on average, neither harmful nor beneficial. Declaring a draw after 2,000 moves, though, was prohibitively time-consuming; long games like that pass by very slowly, even though they usually do have a victor. Bringing the draw-threshold down to 200-moves made games faster, but still produced a *very* slow rate of evolution because so many games ended in draws. Indeed, we estimated that to match Pollack and Jackson's 100,000 generations with this approach, it would take roughly 40 days of continuous computation on a modest computer.

We settled, therefore, on simply taking the weighted average (equation 3) of the rookie and the champ every time the rookie won, under the assumption that churning through generations was more important than preventing bad mutations from incorporating a bit of their judgment. This deviation was practically motivated, not theoretically motivated, so any follow-up work should certainly involve trials with the 3-out-of-4 rule imposed. But is it responsible for the stagnation we witnessed? Again, the answer is most likely no. Because of the lack of stochasticity, each game is very similar the one that preceded it, meaning that only some limited portion of the actual state space of checkers is explored. This problem persists even if we are more selective about the mutations we permit. Whether it takes 5,000 iterations or 50,000, it seems we will eventually start repeating ourselves, though rigorous confirmation of this supposition will take further investigation.

Conclusion

Pollack and Blair's hill-climbing strategy for backgammon can be adapted, with reasonable success, to play checkers as well. Due, however, to the deterministic nature of the game, the networks are prohibitively time-consuming to train and eventually stagnate. In our experiments, the networks improved steadily up to about 5,000 iterations (roughly 2,000 generations) and stagnated thereafter. The results are nonetheless exciting because they demonstrate that, contrary to Pollack and Blair's warnings, deterministic games like checkers can indeed be learned using neural nets and straightforward genetic algorithms.

A logical next step would be to find a good way to force additional variety into the games. If novel mutations continued to accumulate after 100,000 generations, as in Pollack and Blair's

backgammon network, the agent might become quite competitive. It is already substantially better than random after only 2,000. More aggressive random movement, coupled with a more stringent litmus test for the rookie, might go a long way towards accomplishing this.

References

1) https://pdfs.semanticscholar.org/c1ec/5116d71176aaca80b1df944c01e82cc35212.pdf
2) http://www.bkgm.com/articles/tesauro/tdl.html
3) http://neuralnetworksanddeeplearning.com/chap1.html
4) http://fileadmin.cs.lth.se/ai/Proceedings/aaai07/06/AAAI07-098.pdf
5) https://en.wikipedia.org/wiki/Arthur_Samuel
6) https://en.wikipedia.org/wiki/Chinook
7) http://cs.nyu.edu/courses/spring12/CSCI-UA.0472-001/checkers2.pdf

All graphics are original.