UNGNU Backgammon

Alex Haro

Fall Semester 2006
Neural Networks
Professor Keller

Abstract:

There have been many different attempts to teach computers to play the game backgammon. Most of these projects have yielded very successful results. The very best computer backgammon players have already surpassed the best human players at the game. However, there are some flaws that each of these trained computer players have. My project looked at how if I was to combine two strategies, this would yield an even better backgammon computer player. I took the idea of temporal difference learning (one of the first highly successful backgammon training methods) and combined it with a new model of self play. Furthermore, I decided to address the issue where constant training of a neural network developed a network that seemed to be "stuck" with one strategy.

About the Game:

The game of backgammon is based on some very simplistic rules but has a very complex strategy. Two players compete to see who can be the first to remove all their pieces from the board. One player moves his/her pieces clockwise while the other moves his counter-clockwise. The sum of the two rolled dice determines the amount of spaces that the player can move.

On the most basic level, it seems that the player who is able to roll the highest numbers on the dice will be able to move faster and therefore win the game faster. Extrapolating from this observation a good evaluator of each player's equity can simply be calculated by counting how many spaces left before all their pieces are removed from game. However, in backgammon, an opponent can land on one of your pieces and send one of your pieces all the way back to the bar where they have to start moving that piece all the way back from the beginning. An opponent can only do this when you have one

piece on a space, so a defensive strategy would dictate that you always want to move your pieces in such a way that you minimize the number of your pieces that are vulnerable to being bounced back. Also if you have to make one of your pieces vulnerable, you would want to make a piece that's far away from the finish vulnerable. That way, if the piece is bounced back, you minimize the amount of spaces you're set back. A even more advanced thing to note is that you can analyze how vulnerable a certain piece is by seeing what pieces your opponent has that can bounce your piece back. For example, in backgammon, it is most likely that a person will be able to roll a 6[1]. Therefore, having one of your single pieces 6 away from your opponents piece is an extremely dangerous position. Other considerations include that when you bounce one of your opponents pieces back, that piece of yours most likely becomes vulnerable to being bounced back by your opponent.

The endless amounts of different strategies that are involved in backgammon make it an extremely difficult game to master. In fact, expert systems where you can define specific rules for the computer player to follow simply cannot handle the complexity of the game. Neural networks have shown the most promise for being able to handle this game which is why I decided to pursue a neural net project.

1. For example, if you rolled the dice and ended up with a 3 and 4, you would be allowed to move one piece 3 spaces then 4, or the other option is that you can move one piece 3 and another piece 4. Therefore, the chances of a player rolling a 6 is the sum of the probability of a person rolling a sum of 6 and the probability of them rolling at least one 6. So in backgammon, 6 becomes the most likely roll.

Motivation and Goals for Project:

As mentioned previously I find backgammon to be an interesting and challenging problem to study. The game is extremely complex with each turn having 21 possible different dice rolls and on average 20 different possible moves for the player (4). This makes it near impossible for a search tree based computer player to succeed with the game. Expert systems discussed earlier are too limited to do well.

I saw the success of neural network techniques and wanted to see if I can improve on them if I combine two different ones. The original paper about the simple feed forward network indicated that they didn't use any TD learning (2). I wanted to see if the combination of these two would result in something better. There was also the issue where training on a single machine caused the neural network to be stuck at a local minimum/maximum. I addressed this by implementing a system where I have 8 different networks training at the same time. An additional benefit to this would be that I predicted that with 8 machines all training - I would be able to succeed in training a good neural network at a substantially faster rate. Ultimately, I wish to have a neural network that would be a better evaluator of board states than previous networks. Any success or findings I have in my project could also possibly be used to solve other problems where there is a situation with a delayed reward.

TD-Gammon and TD-Learning:

TD-Gammon was one the first neural networks to do significantly well as a computer backgammon player (4). It was developed by Gerald Tesauro and was based on the idea of temporal difference learning. Temporal difference is the idea that the evaluation of a state should be based on the state immediately following it (3). The TD-

Gammon network is a back-propagation net that takes in a board state as input and then outputs a predicted probability of winning for a specific player based on that board state. The network trains itself on the last position with the target score being the outcome of the game. It then trains itself on the second last position and tries to predict the score more accurately. The last position is the only one that gives a reward signal. Such a strategy works really well for problems like backgammon where there is a delayed reward.

Co-Evolution:

Co-Evolution is the process of training two neural networks against each other simultaneously (1). This is the strategy that was used to train TD-Gammon (4). However, instead of training two different networks, it trained against itself in self-play. Researchers at Brandeis University found that while this was a good method of learning, better learning results could be found if the network was trained against a static copy of a neural net. In their model, only one of the neural networks would learn and the other one would be used to help train that network. The idea behind this model is that the stationary network is used to supervise the training of the other network. This leads to better learning results because the stationary network is not evolving as well so it can provide consistent advice to the training network. Therefore, while Co-Evolution is a sufficient model for training it may be possible that training against a static neural network is a better model for learning.

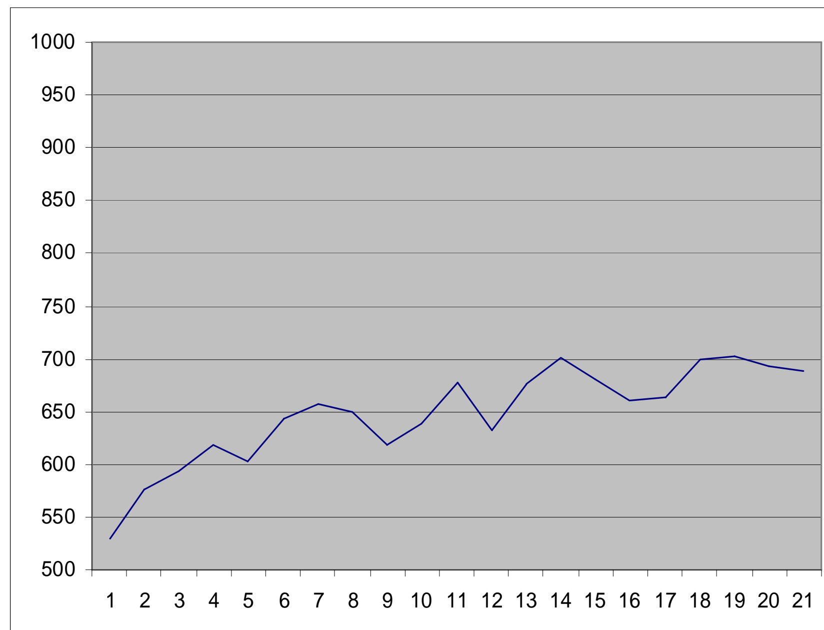
Implementation:

To implement my project, I decided to use the open-source GNU Backgammon project. GNU Backgammon runs on a neural network that is similar to that of TD-Gammon. Additionally, it also includes several tools that allow you to train your own backgammon neural network. I used the code from GNU Backgammon and created a neural network that trained using TD-Learning. My neural network is initialized to weights that are roughly equivalent to that of a neural network called PUBEVAL, which is the standard that most other backgammon players benchmark against (4). I then start training my neural net against a static copy of itself. This is because the co-evolution paper mentioned that training against a static foil seemed to yield better results for them (2). In one case, I ran the neural network training on one computer, every 10,000 generations I would update the static foil I was training against. The second case, I ran it on 8 different computers. Initially, the networks each trained against a static copy of itself and after the first training run, I found a champion among the 8 and then had the networks all train against a static copy of that champion.

For the implementation of the 8 neural networks, I decided to code a MPI implementation that was controlled by a master-slave system. I had one master computer that kept track of all the weights on all the slave computers and was responsible for sending four different tags to the slave computers. These tags were INIT, TRAIN, TOURNAMENT, and KILL. The slave computers received these tags and then had to perform functions depending on the tag. For init, the computers initialized their neural network weights and then sent a message back to the master computer indicating successful initialization. Upon receiving a train tag, the slave computer would begin training the stored weights it had against weights that it receives from the master. It

would then store the training results and send them back to the master. In tournament mode, the slave simply received weights and runs quick competitions and reports back a win loss count to the master. The kill tag simply stops the slave computers.

Results:



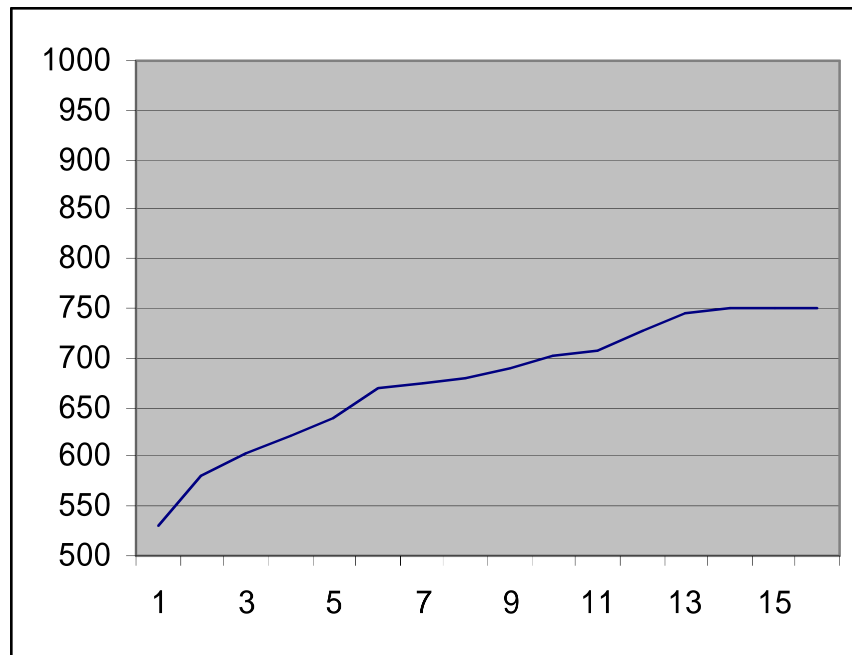
X-Axis: Number of training iterations **scale = 10,000**

Y-Axis: Number of Games won against PUBEVAL out of **1,000**

To evaluate how well my neural network trained, I used a publicly available neural network called PUBEVAL. PUBEVAL is a trained neural network that people run games against so they can get a benchmark for how well there backgammon player is. For comparison, TD-Gammon originally won about 65% of its games against PUBEVAL (it is supposedly much better now). The simplistic co-evolution implementation reported

a 45% win rate against PUBEVAL (2). GNU Backgammon wins roughly 80% of its games against PUBEVAL.

As indicated by the graph above, my first run indicated roughly 50% win rate against PUBEVAL. This was expected since I started out with weights similar to that of PUBEVAL's. After about 4 hours and 200,000 generations I reached a win rate of about 70%. I left the training run for a little while longer and saw that after about 300,000 generations the win rate leveled off around 75%.



X-Axis: Number of training iterations **scale = 10,000**

Y-Axis: Number of Games won against PUBEVAL out of **1,000**

For the 8 neural networks, after every champion was determined, I ran the champion against PUBEVAL to see how well I was doing. Similarly, the network started

out at around 50%. After about 15 hours, I had successfully run 150,000 generations and my champions had topped off at roughly a 75% win rate.

Champion Computer for every tournament:

4, 4, 2, 6, 7, 5, 6, 4, 4, 4, 8, 4, 3, 4, 4, 4

Conclusions:

From my results I can conclude several important things. Based on the single machine run, I saw that combining the two methods yielded a more successful neural network than the two individual methods had. This only took 200,000 generations too! The co-evolution implementation reached a 45% win rate after 300,000 generations (2). By generation 300,000 my neural net had settled to around 75% win rate, almost on par with that of GNU Backgammon itself.

My MPI implementation with 8 computers was also very successful in several aspects. Originally I had issues where I had problems where the code ran too slow. I coded in debug code to make each slave computer report back what machine it was running on and that's when I saw that I was not distributing the load to the machines. After I fixed this error, the code speeded up substantially. As indicated by the results, I reached the 75% win rate in half the number of generations it took the code running on one computer. So I achieved my goal of faster learning. I believe this is due to the fact that with 8 neural networks, I was able to explore a much larger search space for the neural networks. Another surprising result I saw was that my win rate never went down. I attribute this to the fact that with 8 different neural networks training, it was much more likely to see improvement each 10,000 generations.

Future Improvements:

For possible future improvements to this project, I could try addressing some of the goals I hadn't achieved. For example, I hoped to avoid the problem where one computer gets stuck in a local minimum. With 8 computers I hoped that there will be a varied range of champions so that I could avoid that problem. However, computer 4 dominated the wins for my simulation. Secondly, more can be done in speeding up my MPI code. My 150,000 generation run took 15 hours which really surprised me. I think this may be the fact that I always have to wait for all the computers to finish before I can proceed to the next step. If there was one really slow computer, this would significantly slow everything down. Finally, one thing that I would have liked to try given more time is to actually test my neural network against TD-Gammon, Co-Evolution, GNU Backgammon, and even different human players.

References:

[1] Angeline, P. J. and Pollack, J. B. (1994). Competitive environments evolve better solutions for complex tasks. In Forrest, S., editor, *Genetic Algorithms: Proceedings of the Fifth International Conference*.

[2] Pollack, J.B., & Blair, A.D. (1998). Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32.

[3] Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257–277.

[4]Tesauro, G. (1995). Temporal difference learning and TD-Gammon, Comm. *ACM* 38 58–68, HTML version at http://www.research.ibm.com/massive/tdl.html