Brad Cable

May 3, 2010

Illinois State University

Chunk Based Passive Reinforcement for Go

The goal of this research was to make a fast, learned, and hopefully intelligent agent to play Go. Unfortunately, this proved to be more difficult than I had imagined, even though I went into the task knowing that I would probably fail. The algorithm used for learning was a passive reinforcement temporal-difference learning (TD-learning) algorithm. The algorithm used is defined in *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2003)

Similar research has been done on TD-learning in respect to Go (Schraudolph et al., 1994). This research was conducted by a team of researchers who had large amounts of time to put toward their goal, and they were actually somewhat successful, if even they did not outperform everything out at the time. Today, the AI is much smarter and runs on much more sophisticated hardware, so that just adds to the challenge.

The speed of the agent was probably the biggest issue due to time-constraint, but despite splitting the data up into non-optimal chunks, the results were unsuccessful, but not uninteresting.

THE PLAN

GNU Go was pitted against itself in order to generate data for the agent to learn off of. The goal was to beat or at least replicate some of GNU Go's abilities in a rudimentary way. At all points in the learning or playing process GNU Go was used in some way, either by generation of moves or by playing against it. This was necessary to utilize GNU Go's ability to decide the winner. Because no algorithm is perfect at deciding the winning state, the resulting learner was biased based on the blind faith of GNU Go's scoring system. This was done to avoid writing another system to score Go, a likely failure as well.

Learning was used based on chunk *type* (corner, side, middle) by examining the ways each chunk is played out. All chunk types were rearranged internally to a standard orientation to maximize learning. These chunks were constructed for two situations in each move. The first was the standard 3x3 absolute position that are associated with the Go board. The last chunk was a relative chunk with the move being the center of the 3x3 grid.

The rewards associated with each learning chunk were for the amount of killed stones, the amount of liberties associated with each move, any of the winner's moves (scaled up), and any of the loser's moves (scaled down). Winning and losing states were the only used states for each move rather than a factor of how much the winner won by. The reason for this is that in Go, winning the game by 30 points *does not* necessarily mean you are way better than that person in comparison to winning by 1 point. All that is needed to know is who won to judge who played better. Observation of their tactics is the way to tell by how much the player is better, not by final score. This abstract analysis is obviously not very easy to represent.

RESULT

5000 games were generated by GNU Go and were stored for easy access for the learning process. These games were all learned in order, then learned again for 50 to 100 trials to produce the results found. Those games that I did manually supervise (which turned out to be a lot of them), GNU Go appeared to have pretty good accuracy at scoring the final board, unlike the abilities of other software such as qGo. I say this with no intent on bashing qGo here, as it's my favorite Go client, merely an observation.

Komi is used in Go to offset the advantage that black has over white. Usually this is given as a number that is between two integers, such as 5.5, 6.5, or 7.5 (American Go Association, 2010) to the white player, which is also used to prevent draws. Before discovering that the default komi for GNU Go in 9x9 mode is 0, black was winning 723 times out of 1000. The komi value was adjusted to AGA's standard komi of 5.5 and the results were 562 wins out of 1000, which is much more realistic.

After playing around with the various settings, the resulting preferred attribute values were the following:

Reward Ratios	
Killed Stones	0.5
Winner's Move	0.1
Loser's Move	-0.05
Liberties	0.9
TD-Learning Values	
Discount Factor	1
Initial Learning Rate	0.75
Learning Reduction Ratio	0.9
Iteration Counts	
Games	5000
Iterations Per Game	50-100

The agent turned out to be incredibly fast to play after the learning had been performed. While GNU Go was taking many seconds at times to play its moves, the agent was consistently playing at the millisecond level. This speed had its price, however.

Unless there were no possible moves for the agent to play, the agent had no way of telling when it should stop playing. This led the agent to play on clusters that were already dead when it could not find any other moves to make. This led to severely self-destructive behavior on the agent's part. An attempt was made to have it pass the move when the utility function was negative, but that led to excessive passing.

Because of the structure of the code, it turned out to be difficult to judge when a move was illegal in the sense that it would leave it, or the cluster it to which it was associated, with zero liberties. The quick and dirty solution to this was that since these situations were rare to begin with, it was to try to perform random moves which would in all cases be legal except in the same circumstances, which led it to a loop to generate the random move. The best solution would have been to locate the second best move and go with that.

The results of the chunking proved to be the biggest downfall. Because the chunks were 3x3 blocks of chunks, the agent was pretty good at defending 3x3 corners and sides. Unfortunately, in the game of Go a well fortified 3x3 block does not give the player enough to stay alive. In order to develop more than 1 eye, you must occupy a minimum of a 4x4 area (including any sides or corners as a part of that). Splitting the board up into 4x4 blocks would be much more difficult to keep track of and deciding where they should go, but might lead to more promising results. My attempt to fix this was with the relative chunks and did not appear to make any difference. Perhaps keeping track of a larger relative chunk would be better as well.

From my experience with this project, I think the hardest part of coding Go AI is knowing how to count score, knowing when to stop playing on certain clusters, and knowing when to stop playing altogether. These factors cannot be simply learned by observation in the traditional sense, but must be learned in more complicated ways. Another thing to note is that even GNU Go does not know how to score Go 100% accurately. From the games that I saw, it seemed to score them correctly since the excessive movements that led to the agent essentially committing suicide all over the board.

Another observation I noticed was that if the same settings were kept for a certain number of trials, the game would play out the exact same way every single time. This is to be expected of my agent, but not of GNU Go. From past experience playing with GNU Go, it usually does not do the same thing every single time, though that observation is from 19x19 games. It is interesting that GNU Go decides to go the same route every single time in 9x9 games, at least with this input.

All in all, this research was not a complete failure. It at least provided interesting insights into what is *not* good to do for Go AI and also which factors are the hardest to account for. It also shows that passive TD-learning can be used to learn how to develop rudimentary structures for Go.

References

Benson, T. (2010). A change in komi. http://www.usgo.org/org/komi.html.

Russell, S., Norvig, P. (2003). Artificial intelligence: A modern approach. *2*, 767-771.

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