Research Review: Alpha Go

Summary of the Paper Including New Techniques Introduced

According to the provided AlphaGo paper a computer should not have been able to win Go games against the best human player for a decade. Still, AlphaGo did.

As far as I was able to understand from the provided paper there was not a single new technique introduced by the AlphaGo team that caused their win, but instead a novel combination and refinements of multiple existing techniques.

Complexity of Go

Go is a game with simple rules, but many combinations.

The average branching factor b is 250 and the average depth d is 150 moves. With a resulting complexity of b^d an exhaustive search of all possible moves is - even with Google's resources - not feasible.

We learned some techniques in the class so far that would help to cut down on the sheer number of combinations, but it was educational to see that they were not used by AlphaGo. More on that later.

A noteworthy consequence of the simple Go rules is, that the analysis happens on a less mechanical level, but more intuitively instead. We'll see how the AlphaGo team dealt with this in the next section.

The Approach

Capturing Human Knowledge

A large source of knowledge for the AlphaGo team came from a database with 30 million positions from past Go games played by Go masters. Using a deep neural network this knowledge was used for supervised learning to create capabilities to predict the next move that a human player would likely do from a given position.

The resulting network is the SL Policy. It captures the intuition of human players.

Hardening Through Reinforcement Learning

Training the SL policy focused on predicting the immediate next move, but did not take into account how valuable this move is, i.e. if the move ultimately lead to a win.

Instead the SL policy was taken as a basis for random games based on two SL policy players. Reinforcement Learning was applied to derives new policies and tune them with further games, resulting in the RL Policy.

At this point human intuition as well as more mechanical capabilities were fused together.

Value Function: Value Network

A position's value can best be judged by following likely paths to terminal nodes in the game search tree and then backing up the win/loss/draw results.

As mentioned in the beginning this is not feasible for the middle part of a Go game.

Instead a value function is applied that approximates the value of the current position with respect to the anticipated ultimate outcome.

For AlphaGo this was done by consulting a Value Network and by simulating scenarios from the current position to the end.

The Value Network was trained in concert with the RL Policy and captured the results of the automatically played games - these were played to the end - and the prior positions leading up to the result.

Value Function: Monte Carlo Tree Search

The value network was created during the training phase. However during the actual game this value network was not used exclusively.

Instead it was augmented by the policy network.

The AlphaGo team applied a Monte Carlo Tree Search (MTCS), that does simulate scenarios from the current position to terminal states. It is important to understand here, that evaluating all combinations would still not be feasible, but following individual deep paths to the end is.

Traditionally MTCS would choose which path to pick randomly at first and then adjust further scenarios runs based on the terminal states it encountered before. It would also favor picking unexplored paths over going repeatedly down paths that are initially the same as something already seen before.

In AlphaGo's variant of MTCS they are also putting an emphasis on exploration, but extend MTCS so that they don't make initial choices randomly, but use prior knowledge from the policy networks to make informed decisions.

Given that this is just an approximation and that the speed of evaluation is critical - a fast evaluation allows more scenarios to be played and the search tree to be deepened farther - a slimmed down version of the policy network (P_π) was used.

The team experimented with different weights for incorporating the results of policy and value network and found that giving them equal weight brought the best results.

Distributed AlphaGo

The AlphaGo team relied on parallel execution of search and evaluation, using GPUs for the latter. This is already true in an AlphaGo single node setup. However AlphaGo also works in a distributed setup across multiple nodes.

When looking at Figure 4 in the provided paper the gain from using the distributed mode seem modest. However taking into account that the last percent of performance is the hardest, it is quite significant. Henceforth it comes as no surprise that the distributed version of AlphaGo beat the single node version 77% of the time.

Relation to What We Have Learned So Far

$\alpha\beta$-Pruning

We learned about $\alpha\beta$-Pruning as a technique to cut down a game search tree by removing paths that would not change the outcome anyway.

This was not applied by AlphaGo, but given that the next moves were not picked by a depth-first-search algorithm, but from the SL Policy, this is likely inherent knowledge and the MTCS takes into account the relative value of different branches and therefore low value branches will only be explored very sparsely.

Minimax

Minimax takes into account the adversarial nature of games like Go and for each level in the game search tree alternates between picking the best move for one player or the other, each time maximizing their respective results.

Using the predicted moves from the SL Policy seems to do this inherently. There is a change in perspective as to predicting the next human player move alternating black and white.

Iterative Deepening

As an exhaustive search of a game tree may be infeasible, the search is often cut off at a specific depth and the nodes at that depth are evaluated by approximating the anticipated outcome if we were to go to the terminal nodes.

Iterative Deepening starts with a search that is cut-off at a level that is easily reachable. In case time runs out during later steps the result from the previous run will be returned. But if time does not run out, another evaluation of the game tree down to one more leven than before is conducted. Here we take into account the results from the previous run to order the nodes in fashion that makes $\alpha\beta$-Pruning efficient.

AlphaGo does not do Iterative Deepening. They use MTCS instead, which runs its rollouts from the root of the tree and can run as many scenarios as time permits, continuously updating the previous results with respect to the best next move and the most valuable target to further explore.

This looks like a good alternative to Iterative Deepening.

Value Function

Alpha Go doesn't use a value function that computes everything based on the current position on the fly, but uses a value network trained prior to the actual game. It also used MTCS to take into account the actual game played.