Research Review on AlphaGo's 2016 Paper on Nature

As part of fulfillment of project 2 in AIND, Udacity

(based on this distribution: https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf)

1 Introduction

For a two-player zero-sum game, to build an agent is essentially to do an effective adversarial search. An obstacle for building agents for a large game is the enormous search space. To recursively compute the optimal value function in a search tree with average breadth b and average depth d needs to consider approximately b^d legal moves. In the game of Go, $b \approx 250$ and $d \approx 150$ (comparing to chess that is already a large game with $b \approx 35$ and $d \approx 80$), implies that exhaustive search is infeasible. If one could find a good estimate to the optimal value function, $v(s)$ from a board position/ state s that predicts the outcome, the search depth could be reduced. But this is another obstacle in building agents to play Go, that a good estimate to $v(s)$ is hard to find. With these two obstacles, Go is no doubt the most challenging classic games for artificial intelligence.

Before the rise of DeepMind's AlphaGo, previously state-of-the-art approach was to use Monte-Carlo tree search (MCTS) to narrow the search to consider only high probability actions, and to sample actions during rollouts. Such an approach has achieved amateur 6 *dan* (by Crazy Stone) but limited to shallow policies and value functions based on a linear combination of input features, that required manual designs, selections and tuning, as the heuristic function. Using features might not be a very good idea when Deep Blue[2] created in about two decades ago for solving a relatively smaller game, chess, already required as many as 8000 features.

AlphaGo[1] used convolutional neural networks (CNN) to overcome these two obstacles. To reduce the breadth of the search, policy networks were introduced to select moves. To evaluate board positions, value networks were introduced to reduce the depth of the search. By combining Monte-Carlo simulation with value and policy networks, AlphaGo's search algorithm was proved to be successful. It achieved a 99.8% winning rate against other Go programs, and even defeated Fan Hui, a human European champion as well as a professional 2 *dan*, by 5 games to 0.

2 Methodology

In view of recent unprecedented performances in visual domains achieved by deep convolutional neural networks, DeepMind introduced policy networks and value networks to determine the priority in Monte-Carlo simulation to build their agent AlphaGo. In more details their neural networks has a few types of architecture and is trained in the following pipeline:

To begin with, the first network in the pipeline is a supervised learning (SL) policy network, $p_\sigma(a|s)$, a probability distribution of action a given a state s. This is supervised by expert human moves from 30 million positions on the KGS Go Server. In brief, SL is built by random sampling of state-action pairs (s, a) and followed by a stochastic gradient ascent to maximize the likelihood of the human move a in state s. By alternating between convolutional layers with weights σ, applying rectifier non-linearities, and attaching the final softmax layer, SL is completed with the output probability distribution $p_\sigma(a|s)$ over all legal moves given any state.

SL is fast, learns efficiently with immediate feedback and gives high quality gradients. Its accuracy is 57.0% on a held out test set, using all input features, and 55.7% using only raw board position and move history as inputs, high than another other software at that time which is at most 44.4%.

DeepMind also trained an even faster policy. The rollouts policy, $p_\pi(a|s)$, can rapidly sample actions during rollouts. With a lower accuracy at 24.2%, p_π requires just 2 μs to select an action, compared to 3 ms in SL.

Now, the major policy network, $p_\rho(a|s)$, is ready to be trained by reinforcement learning (RL). RL improves the SL policy network by optimizing the final outcome of games of self-play. This process is a

policy gradient reinforcement learning, aiming to adjust the policy towards the correct goal of winning games, instead of maximizing predictive accuracy. $p_\rho(a|s)$ of RL is initialized with $p_\sigma(a|s)$ of SL. Keeping the same structure, only the weights ρ would be updated in self-play at each time-step t by stochastic gradient ascent in the direction that maximizes expected outcome via a reward function $r(s)$. To prevent overfitting to the current policy, games were self-played between the current policy network p_ρ and a randomly selected policy network formed the pool formed by previous iterations.

Another important CNN is the value network, $v_\theta(s)$, that predicts the outcome from position s of games played by using policy p_ρ for both players. Using data of complete games to predict the winner of games played by the RL policy network against itself leaded to overfitting. A reason given by DeepMind is that, "successive positions are strongly correlated, differing by just one stone, but the regression target is shared for the entire game." So they generated their self-play data set from 30 million distinct positions, with each sampled from a separate game.

Finally, with all policy and value networks built, they are ready to perform Monte-Carlo simulation. There are two main steps. One is to selects action by lookahead search with policy network. This step is performed to maximize the sum of the action value and a bonus that is proportional to the prior probability but decays with repeated visits. Thus, the search would also encourage exploration. The other is to evaluate leaf nodes of the tree search. Leaf nodes are evaluated by a linear combination of value network $v_\theta(s)$ and outcome of a random rollout played out until terminal step under $p_\pi(a|s)$, the fast rollout policy. Throughout a game, the action values and visit counts of all traversed edges are updated after each simulation.

3 Results

Several variants of AlphaGo had gone through different tournament. And all of them achieved outstanding results.

When played against the strongest Go programs available, including Crazy Stone, Zen, Pachi, Fuego and GnuGo, (given 5 s per move), a single-machine AlphaGo had a winning rate of 99.8%. Even in four-handicap-stone-games, AlphaGo had a winning rate from 77%, 86% and 99%, against Crazy Stone, Zen and Pachi respectively. A distributed version of AlphaGo only lose to a single-machine AlphaGo with probability 23%, but never lose against any other programs.

Recall that the position evaluation function of AlphaGo is a linear combination of $v_\theta(s)$ and a random rollout played until terminal step under $p_\pi(a|s)$. The tournament also assessed the performance of different weighting of the linear combination. It turned out that the mixed evaluation with equal weight to the two components is the best, with a 95% winning rate againts other variants.

The last evaluation in AlphaGo's 2016 paper was the match between a distributed version of AlphaGo and a human professional 2 *dan* Fan Hui in October 2015. The result is mentioned previously and is well known. In the words of DeepMind team, "This is the first time that a computer Go program has defeated a human professional player, without handicap, in the full game of Go—a feat that was previously believed to be at least a decade away."[1]

References

[1] David Silver, Aja Huang, Chris Maddison, et al., *Mastering the game of Go with deep neural networks and tree search. Nature*, 529, 484489, 2016. doi:10.1038/nature16961.

[2] Murray Campbell, A. Joseph Hoane Jr., Feng-hsiung Hsu, *Deep Blue. Artificial Intelligence*, 134(2002): 57-83, 2002. doi:10.1016/S0004-3702(01)00129-1.