AlphaGo - Overview

Guruprasad Sridharan

May 2017

1 What is Go?

Go is a board game typically player on a board with 19x19 tiles. It's a two-player adversarial, zero-sum, perfect information, deterministic game. The main objective of the game is to capture territories on the board by placing stones. A territory is defined as any area on the board that is surrounded by stones of one particular player. There are two coloured stones - black and white - used by the two players respectively. Black plays first and the players take alternate turns. It is possible to surround your opponent with your stones and the entrapped stones of the opponent within that territory become prisoners of war. In the end of the game, when there aren't any more moves to make or there are two consecutive passes, the player with more number of points wins. Each prisoner captured and territory square contribute one point to the overall score.
For the past half century, many computer scientists have tried to build an AI agent that is better than humans at playing Go. The high branching factor and endless possibilities of moves make it computationally very expensive to look ahead even beyond a couple of moves in the search tree. The number of ways a particularly Go game can be played is comparable to the number of atoms in the visible part of the universe. Owing to these reasons, it is very hard to build an AI agent that beats even a rookie human player consistently let alone a professional champion. The difficulty stems from the fact that it is very hard to design a static evaluation function to calculate the utility of game states. Historically, humans have always been better at pattern recognition tasks than machines until the advent of Deep Learning algorithms.

2 Key Challenges

- High branching factor limiting ability to look-ahead since it is computationally expensive (time and memory constraints). Large number of possible moves at each turn.
- Unlike other games like chess, tic-tac-toe, etc. standard counting based evaluation functions don't work well. It's harder for computers to develop human-level intuition about what states are good and what states are bad.

There is inherent long range dependency between game states ie. moves made long back can have a significant impact on the game later on.

- Complex decisions to make in local search - groups of connected cells that can be kept alive or can be attacked - evaluating strength of groups of cells - joining groups together. It's very hard to do it without domain expertise. Probabilistic systems have generally fared better in overcoming these obstacles. It's quite hard to define what a good position in layman terms.

3 AlphaGo

In March 2016, AlphaGo, a new go playing AI agent designed by Google Deepmind played a series of 5 games against Lee Sedol, the reigning world champion at that time. AlphaGo beat Sedol 4-1. It was a phenomenal achievement at that time, even quite unimaginable.

3.1 Summary

AlphaGo uses deep neural networks to learn policy and value functions. It uses supervised learning to learn from games played by experts and reinforcement learning to learn from self-play. It also uses a new search algorithm that combines Monte Carlo Simulation (MCTS) with value and policy networks. Using these neural networks, they build an agent that doesn't do any look-ahead search and achieve a 99.8% winning rate against existing state-of-the-art Go playing agents based on Monte Carlo Tree Search.

3.2 Detailed Description

The Machine Learning pipeline includes two stages - Supervised Learning and Unsupervised Learning. The agent uses supervised learning to learn from expert games. The objective is to learn the optimal policy function (ie.) what action to pick given a particular state. Two different neural networks are used - a SL policy network p_σ and a fast rollout policy network p_π. The SL policy network p_σ is a 13-layer deep convolutional neural network (CNN) using alternating convolution (weights given by σ) and rectifying linear unit (RLU) layers with softmax to calculate probability of an action being performed by an expert human player. The CNN takes as input a simple representation of the board state and a history of the past moves. It is trained on randomly sampled (s,a) pairs from a dataset of expert games. The SL policy network achieved accuracy of 57% on held-out test data. The fast rollout policy network p_π is a shallow network trained using linear softmax of small pattern features with weights π which achieved an accuracy of 24.2% but is much faster to evaluate than p_σ.

The RL policy network p_p is used to improve the SL policy network by policy gradient Reinforcement Learning. Intially, the RL policy network is initialized to the SL policy network (ie.) initially p = σ. The current policy network

plays against a randomly chosen earlier version of the same network (to reduce overfitting) and depending on the outcome of the game adjusts the weights p in the network with back propagation so as to maximize the outcome of winning against the opponent. Thus the SL policy network is optimized for winning rather than accurately predicting the move made by an expert human player given the board state. So both the SL and RL policy networks, output a conditional probability distribution $P_\sigma(a|s)$ or $P_p(a|s)$ over all available legal moves given a state s. The RL policy network samples from the distribution $P_p(a|s)$ for making its moves. It was able to win more than 80% of its games against the SL policy network. A value network $v_\theta(s)$ with convolution layers (weights θ) is trained with reinforcement learning that outputs a scalar value indicating the expected outcome of the game from the perspective of current player at state s. The value network is trained with randomly chosen (state, outcome) pairs to minimise RMSE (root mean square error) on prediction of outcome given a particular state. from independent games (to reduce overfitting) in a large self-play dataset.

The search algorithm combines MCTS with the policy and value networks. MCTS uses look-ahead search to evaluate states. Each edge in the search tree corresponds to a particular state-action pair (s,a). Each edge stores an action value Q(s,a), visit count N(s,a) and prior probability P(s,a). During every simulation, the tree is traversed by descending from the root. At each step t, we choose the legal action that maximizes the sum of Q(s_t,a) and bonus u(s_t,a). u(s_t,a) is a function that scales the prior probability with the visit count. During step L, if traversal reaches a leaf node, it may be expanded. The leaf nodes are processed only once by the SL policy network which sets the prior probabilities for the edges spanning out. Two evaluations happen on leaf nodes, first the leaf node is evaluated using the value network $v_\theta(s_L)$. Second, a random rollout is played from the leaf until terminal step T by sampling actions from the fast policy network and the outcome z_L is calculated. Then both the values are combined using a parameter λ which lies between 0 and 1. After the end of the simulation, action values and visit counts of all traversed edges are updated. The action values are calculated from $V(s_L)$ averaged over all simulations and the visit counts N(s,a).

3.3 Results

AlphaGo combines Monte Carlo Tree Search with deep learning algorithms to achieve professional level in playing Go. The novel combination of Deep Supervised and Reinforcement Learning has allowed AlphaGo to build effective approximations of optimal policy and value functions. This has imparted the ability to make complex decisions and bestow a human-level intuition for board positions.