ALPHAGO

Introduction

AlphaGo is a Go program that performs at superhuman levels at Go. It uses value and policy networks for evaluation of state and action values. It uses supervised learning from expert games to bootstrap and self play via RL. The authors also introduce a search algo that combines MCTS and value and policy networks.

Why is Go hard?

Go is a perfect information game with a huge state space. It also becomes difficult to evaluate how good a given position is or how to select an action. Since this is a perfect game, it has an optimal value function $v^*(s)$. So, the game can be solved by recursively computing the value function in the search tree. However, if b is the number of legal moves and d is the game length, you've to evaluate roughly b^d positions. In Go, b ~ 250 and d ~ 150; something something greater than the #atoms in the universe. So, that throws naive exhaustive search out of the window. Now, the question is how to improve on this.

Well, in the paper, they provide 2 ways to mitigate this.

1. Reduce the depth of the tree search\ by replacing the subtree below a depth by it's approximate value function. (TD(n) and bootstrapping).
2. Sample a policy p(a|s).

AlphaGo also uses a MCTS search. Traditionally, you use MC rollouts to estimate the value of a state. Asymptomatically, you should reach the optimal value function via rollouts and value estimation. Prior work has used shallow rollouts with linear combinations of features.

Neural nets take in the image of the board and output either a value function of the board or give the P(a|s).

Pipeline

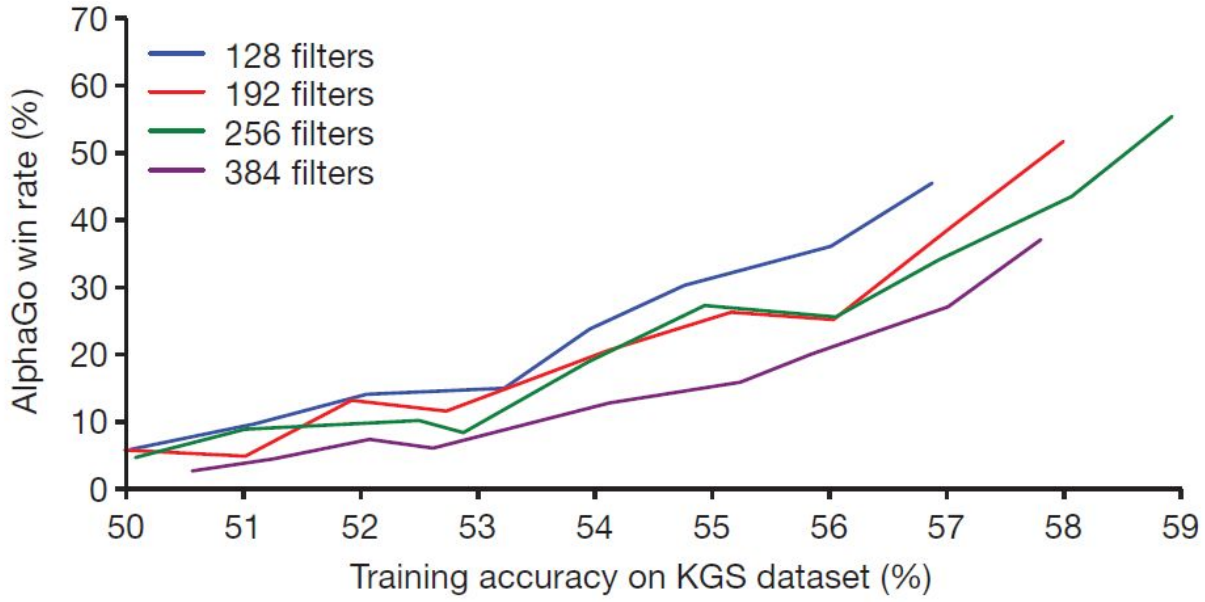
Training happens in the following stages.

- Supervised learning of policy network p_σ (from humans)
    - This provides fast, learning updates.

- Train a fast policy network, p_π , that can be used to rapidly sample actions during rollouts.
- Train a RL policy network, p_ρ , that improves the SL policy network by optimizing for the outcome via RL.
- This adjusts the policy network towards winning the game rather than predictive accuracy.
- Finally, train a value network, v_θ that predicts the winner of the games played by the RL policy itself.

Supervised Learning of policy networks

The SL policy network is learned first. A 13 layer deep CNN was used to predict the expert move from the board position.. This was trained on 30 million moves from KGS Go Server. It had an accuracy of ~55-57 % on expert moves on a held out test set. They also trained on a much faster but less accurate rollout policy network.(acc of around 24%). The difference is in the speed (2us vs 3ms). There are additional features as well that are used as input features (+2 %).



RL of policy networks

They use policy gradient to train the RL policy network, , p_ρ. The RL network is identical in structure to the SL policy network and the weights are initialized to the SL policy network. So, self games ensue. You play using the current policy against one of the previous iterations of the policy. This randomization is to stabilize training and prevent overfitting. The reward function is one that gives out a + 1 or -1 for winning and losing respectively.

RL of value network

The final stage in the pipeline focuses on position evaluation, estimating the value function that predicts the outcome from the position s of the games played by policy p for both players.

While you need the optimal policy to get the optimal value function, you don't have that. The next best thing is to use the strongest policy that was learnt during self play and that is what is used. This network has a similar structure to the RL policy network but outputs a single prediction instead. Given a state-outcome tuple, gradient ascent is used to optimize on a MSE error (between the outcome and the value output).

One thing to note is that temporally close by positions will be highly correlated. So, they found that the network started to memorize outcomes of games when they didn't break this correlation. They created a new data set consisting of 30 million positions and these were evaluated using the RL policy network via self play. They found it helped prevent overfit.

They found out that the position evaluation using the value network was consistently more accurate than compared to MC rollouts using the SL fast rollout policy network. A single evaluation of this approached the MC rollout of the RL policy network.

Searching with policy and value networks.

AlphaGo combines the policy and the value networks in a MCTS that selects an action by lookahead. Each edge of the search tree stores an action value Q(s,a), visit count N(s, a) and a prior probability P(s,a). The tree is traversed by simulation starting at the root state. At each time step, ana action is selected from the state s_t.

$$a_t = \text{argmax}(Q(s_t, a) + u(s_t,a))$$

$$u(s, a) \sim P(s,a) / (1+ N(s,a))$$

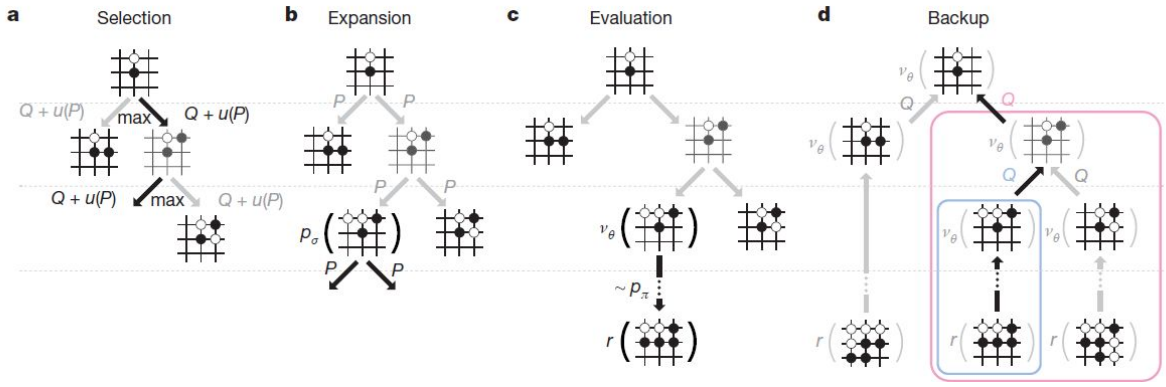
The bonus is there to help in exploration. When the traversal reaches a leaf node at some time step L, the leaf node may be expanded. The said leaf node is processed by the SL policy network to obtain P(s,a) = P(a|s). The leaf node is then evaluated using the value network as well as the using the rollout network until the termination. These evaluations are combined using a mixing parameter, lambda. (similar to TD(lambda)).

$$V(S_L) = (1- \text{lambda})\, V(s_L) + \text{lambda} * z_L$$

At the end of the simulation, the action values and the visit count gets updated for all traversed edges.

$$N(s,a) = \text{Sum}(i,1,n) \text{ of } 1(s, a, i)$$
$$Q(s,a) = 1/N(s,a)\, \text{Sum}(i, 1,n) \text{ of } 1(s,a, i) * V(s_L^i)$$

Once the search is complete, the edge with the most visits is taken.



One thing to note here is that the SL policy is used. They found that the human derived policy was better because of better diversity of move selection when compared to the RL policy. However, they found that the value function from the SL performed significantly worse.