Learn to play Go

Albert Liu
Department of Computer Science
Stanford University
albertpl@stanford.edu

1 Infrastructure

As illustrated in Figure 1, our simulator includes a Go gameplay engine, which is based on Pachi [1], an open source Go framework. We choose Pachi due to its efficiency and simple APIs. Each agent is given a 2D numpy array (e.g. 19×19), representing current board with each element encoding the color of stone, 0 for empty, 1 for black stone and 2 for white stone. Then each agent returns the next move to play, encoded as position of the board (e.g. E4) and a pass action. Each agent can optionally persist the game records, numpy arrays that hold the board representation, moves and etc., to disk, which are used as experiences for the reinforcement learning agents. We use Keras as our deep learning framework.

2 Approach

Minimax search or AlphaBeta search is intractable due to enormous search space. Instead, we start with Monte Carlo Tree Search [2]. Then we explore policy gradient based methods. Finally we will combine tree search and function approximation approach, following AlphaGoZero [3]. But first, we discuss the baseline and oracle.

2.1 Baseline

For baseline, we write a random agent which selects legal move randomly, except that it won't commit suicide, i.e. filling in its own eye, which is an empty position where all adjacent positions and three out of four diagonally adjacent positions are its stones (or edges).

2.2 Oracle

We use Pachi built-in *UCT* engine as our oracle which is said to achieve highest amateur expect level (KGS 7 dan) on 9×9 or KGS 2d rank on reasonable hardware, where dan is expert Go ranking, from 1 to 7 (low to high). Technically the *UCT* engine implements RAVE [4], instead of classic UCT. This engine incorporates lots of heuristics and predefined patterns above the basic Go rules, e.g. self-atari detector and ladder testing [1].

2.3 Monte Carlo Tree Search

The MCTS evaluate each node of the game tree by simulating random games. Each node s is a game state, consisting of a tuple of (board representation, the player to play). And there is an edge between s and s' if and only if for some move a, $s' = \text{Succ}(s, a)$, where $\text{Succ}(s, a)$ is function that returns the next game state s' after playing the move a. For each edge (s, a), we maintain visit count $n(s, a)$ and value $q(s, a)$

Preprint. Under review.

Input: root node s_0: current game state
Input: $c > 0$: parameter to control the degree of exploration
while *within computation bound* **do**
  $s \leftarrow s_0$
  $\Delta \leftarrow \emptyset$
  // selection based on tree policy: Upper Bound Confidence
  while s *is in the tree* **do**
    $a \leftarrow \arg\max\{q(s,a) + c\sqrt{\frac{\log \sum_{a'} n(s,a')}{n(s,a)}}\ \}$
    $\Delta \leftarrow \Delta \cup \{(s,a)\}$
    $s \leftarrow \mathrm{Succ}(s,a)$
  expand tree to add s
  simulate a random game from s and let r be the reward
  // update each node on the trajectory within the tree
  for $s, a \in \Delta$ **do**
    $n(s,a) \leftarrow n(s,a) + 1$
    $q(s,a) \leftarrow q(s,a) + \frac{1}{n(s,a)}(r - q(s,a))$

Algorithm 1: MCTS with UCB policy

2.4 Self-play

In order to apply reinforcement learning, we need to collect experiences (i.e. game records). The standard approach is to have agents play against themselves and save the game records. We record the board position at each step perceived by the player, the move of the player, the color of the player, the final reward (-1 if white player wins, +1 if black player wins) and network predictions. Then later the agent can use these experiences to update its network.

2.5 Actor-critic with baseline

The policy gradient based methods, such as REINFORCE [TODO] and actor-critic [TODO], allow us to learn stochastic policy naturally. For our episodic cases, we define the objective function, $J(\theta)$, to be the start value of any state S, following our parameterized policy function π_θ which is approximated by a deep neural network, i.e., $J(\theta) = V_{\pi_\theta}(S)$. By Policy Gradient Theorem [TODO], we can compute the gradient as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[Q_{\pi_\theta}(S_t, A_t)]\nabla_\theta \log \pi_\theta(A_t|S_t)$$

REINFORCE method uses Monte Carlo method to give unbiased sample G_t for $Q_{\pi_\theta}(S_t, A_t)]$ and the update rule is

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi_\theta(A_t|S_t)$$

But Monte Carlo gradient typically has high variance and produces slow learning. Actor-critic with baseline is suggested [TODO] to solve such high variance issue. Let advantage function $A_{\pi_\theta}(S_t, A_t) = Q_{\pi_\theta}(S_t, A_t) - V_{\pi_\theta}(S_t)$, the update rule is

$$\theta_{t+1} = \theta_t + \alpha A_{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t|S_t)$$

2.6 Tree Search guided with NN

3 Error analysis

3.1 Monte Carlo Tree Search

Here are our results for MCTS approach against baseline and oracle.

We see our MCTS result gets better with more rollouts per move, as expected. But it still loses more games against pachi *UCT* engine. And we speculate that reasons are

Table 1: Results: MCTS

opponent policy	number of rollouts per move	win rate	median time per episode
random	100	0.92	2.4
random	1000	0.99	29.3
pachi *UCT* engine	1000	0.16	35.1

- Rollout policy

  Unlike our MCTS, pachi entails more Go knowledge when selecting next move during the game simulation. This is called **heavy** or strong rollout policy, whereas ours is called light rollout policy. Pachi's rule based simulation policy is handcrafted to mix in heuristics such as, if the last move has put its own group in *atari* we capture it; *Nakade*a move is played inside the eyeshape to prevent the opponent from splitting it into two eyes [1].

- Priors

  We expand a new node by randomly selecting a legal move. But Pachi apply again a set of heuristic function to this decision, equivalently adding a prior probability distribution. Specifically it takes the progressive bias strategy [4], which adds virtual simulations based on the applied heuristics with decreasing weights as the nodes are explored.

- RAVE

  MCTS updates values of nodes with the reward of the simulation, along the sampled path from root to the leaf. But RAVE [4] distributes the update to larger set of nodes. Basically it applies AMAF heuristics [5] (all-move-as-first), which keep previous simulations and apply it whenever we examine the same (s, a) edge and then combines it with regular simulations.

  We believe the tree search guided by NN is a better approach. The NN is used to approximate an evaluation function and prior distributions.

4 Figures, tables, references

4.1 Figures

References

[1] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011.

[2] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[4] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[5] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004.

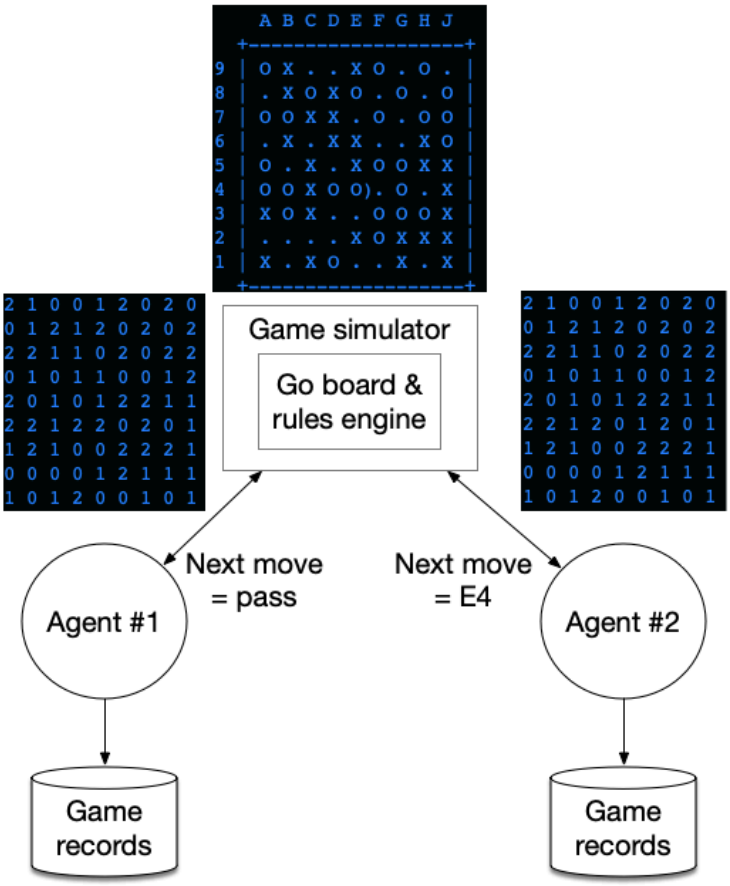

Figure 1: Environment setup