DS-20250724-Mastering the game of Go with deep neural networks and tree search

Michael.Niu, Haoxing.Liu, Kailun.Dong

July 2025

1 Highlights

- Use **value networks** to evaluate board positions.
- Use **policy networks** to select moves.
- Use **Monte Carlo tree search** to simulate random games of self-play.
- Combine **Monte Carlo search** with **value** and **policy** networks.

2 Reduce Search Space

A traditional game with perfect information has a "winning strategy," that is, a set of actions that guarantees a winning (or non-losing) state. The complexity of the strategy depends on the game's **breath**, the number of legal moves at each position, and the game's **depth**, the game length.

- Depth of search:

  truncate search tree at state s and replace the subtree by approximating value function that predicts the outcome of that state.

- Breadth of search:

  sample actions from a policy $p(a|s)$, which is a probability distribution over movement and position.

  Is the distribution of action is a continuous function?

- Board representation:

  A image of 19×19 is used to represent the board as passed to CNN to reduce both depth and breadth search space.

3 Training Pipeline

- Supervised Learning (SL) policy network p_σ trained from human moves. SGA(SGD) to train the network with σ as weights:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

σ : model weights that need to be trained.

s : the board state

a : the action

$p_\sigma(a|s)$: the probability of making move a at state s, given parameters σ

Stochastic Gradient Ascent (SGA) is an optimization algorithm used to find the maximum value of a function. In the SL policy network, we aim to maximize the likelihood $p_\sigma(a|s)$, or more precisely, the loglikelihood, for any state s, where a is the move made by professional players, by modifying the value of σ gradually in the corresponding gradient direction.

$$\sigma_{n+1} = \sigma_n + \alpha \frac{\sum_{k=1}^{m} \frac{\partial \log p_{\sigma_n}(a^k|s^k)}{\partial \sigma_n}}{m}$$

α : an annealing step size that gradually reduces as we get closer to the solution

m : size of the sample batch of action-state pairs

$\frac{\partial \log p_{\sigma_n}(a^k|s^k)}{\partial \sigma_n}$: the partial derivative, or gradient, is computed using automatic differentiation at σ_n

- Fast policy p_π: rapidly sample action from rollouts.

  This simpler network focuses on the previous action that leads to the current state s and the local patterns around a potential action. These features include small 3x3 patterns surrounding a potential move and 12-point diamond-shaped patterns surrounding the previous move. The primary factors are stone color, liberty counts, and some rule-specific features. The policy is faster than SL policy network but has lower accuracy (about half).

- Reinforcement Learning (RL) policy network p_ρ: improves SL network p_σ by optimizing the final outcome of games of self-play. The goal is to go beyond mimicking human behaviors and actually wins the game (optimize towards final stage output).

  The RL training stage is divided into three steps:

  1. Initialization: It begins as the trained SL policy network $p_\sigma(a|s)$
  2. Self-Play: Playing with randomly selected versions of itself (different parameters ρ)
  3. Terminal Reward: Only rewarded at the end with winning (+1) or losing (-1). reward: $r(s_t) = 0$ for $t < T$ and $r(s_T) = 1$.

  Beginning with two identical copies, $\rho_1 = \sigma$ and $\rho_2 = \sigma$, it updates the weight by giving advantages to moves that lead to a win. The following games are played between the current state of p_ρ and a randomly selected previous iteration of the policy network. The gradient for SGD at time t is roughly:

  $$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho}(z_t - v_\theta(s_t))$$

  $z_T = \pm r(s_T)$, the outcome

  $v_\theta(s_t)$: the value at time t. This defaults to 0 and the trained value network will be used later.

  Note that at the first run, where $v_\theta(s_t) = 0 \,\forall\, t$, the gradient only has value at T where it sees the outcome of the finished game. But once we have a trained value network (discussed below), the gradient considers all the sequential steps leading to the final outcome, which stabilizes the reinforcement learning process and makes the training more effective.

- Value network $v_\theta(s)$: predicts the winner of RL network against itself. Architecture similar to the policy network, but outputs a single prediction instead of a probability distribution.

  SGD for the value network:

  $$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$$

  specifically:

  $$\Delta\theta = \frac{\alpha}{m}\sum_{k=1}^{m}(z^k - v_\theta(s^k))\frac{\partial v_\theta(s^k)}{\partial \theta}$$

4 Search with policy and value networks

- Action value: $Q(s, a)$. Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v\theta(\cdot)$ in the subtree below that action.
- Action selection:

$$a_t = \arg\max_a \left(Q(s_t, a) + u(s_t, a)\right)$$

- Bonus term: proportional to the prior probability but decays with repeated visits to encourage exploration.

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

more specifically:

$$u(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)}$$

- Leaf node evaluation:

1. by the value network $v_\theta(s_L)$.

2. by the outcome z_L of a random rollout played out until terminal step T using the fast rollout policy p_π.

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L$$

- Action value function:

$$N(s, a) = \sum_{i=1}^{n} 1(s, a, i)$$

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{n} 1(s, a, i) V(s_L^i)$$

or:

$$Q(s, a) = (1 - \lambda) \frac{W_v(s, a)}{N_v(s, a)} + \lambda \frac{W_r(s, a)}{N_r(s, a)}$$

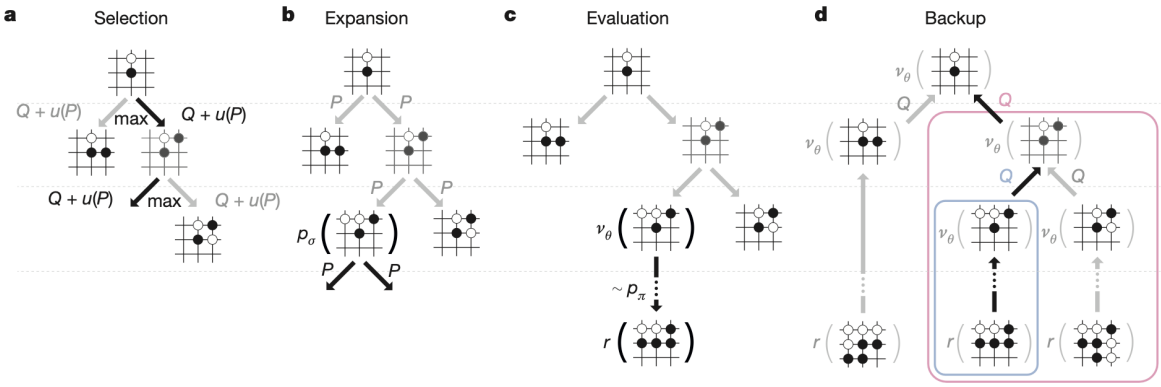

Figure 1: Asynchronous policy and value MCTS (APV-MCTS)

- Backup: Update the value and visit counts of all nodes along the path from the leaf back to the root, based on the evaluated value.

  At each in-tree step, the rollout statistics are updated with virtual loss, which discourages other threads from exploring the identical variation.

  At the end of the simulation, the rollout statistics are updated in a backward pass through each step $t <= L$, replacing the virtual losses by the outcome.

5 Connection to our Scenario Generator

We can formulate our program of generating a game scenario as gradually filling out an $m \times n$ board with symbols one at a time.

- Similarities:
  - The 2D board is the same.
  - At each move, we place a symbol.
  - At a certain condition, the symbols will disappear (getting eaten in Go).
  - The payout (win condition) only happens at the ending stage.
- Differences:
  - We have more than two pieces/players in each game. We need to decide what symbols to play at a move.
  - Our "game" ends when the target value is obtained, but not when the board is filled or there is no more legal move.
  - We have specific rules for clearing symbols, and each clearing (cascading) generates a partial payout.
  - Our winning rules will be constantly changing (for different games). The training process needs some transferability.
  - While we can have an example of the end board, we technically don't have examples of "next-move" - but we might be able to direct certain preferences based on paylines and win patterns.
  - Instead of a win/lose outcome (1/0), we have payouts as the final target.
  - Technically, we cannot employ the self-gaming strategy.
- The Supervised Learning Stage:

  We trained the network using existing scenarios. At each stage, the next move is to fill another empty position with a symbol that makes it closer to the winning payout. The main issue is that we do not have a pool state-action pair. However, perhaps we can derive some priority logic/reward based on the game rules. We need additional stopping criteria for each symbol player based on their target value and board dimension.
- The Reinforced Learning Stage:

  We need to properly set up the self-play scheme, as there is no actual competition in our scenario generation. One possible idea is to set the losing state to a 0 payout and have the two or more players compete towards the value of the symbols' target outcome.

  This stage helps create a better winning strategy, or in our case, "better scenario.". But do we need a better scenario as long as they have the same payout?
- The Value Network (crucial)

  Once a scenario is formed, the value(payout) can be directly computed. We need to train a network to come up with a single value prediction of the final output based on the current board. This should account for the existing partial payouts.

- Monte Carlo Tree Search (MCTS)

  Once the value network is set up and we have partial value for each board stage, we will be able to give direct guides/rewards to each move of the Scenario. In our case, MC was already heavily used to simulate outcomes, and the system only needs to select and prune. The Value Network and the Learning Policies will help us find fast paths to feasible solutions for Scenarios.