Implementing MCTS in RL

Contents

1 AlphaGo Paper

Self-trained.

1.1 Deep Network

In Go, policy controls the moves to win a game, to model uncertainty the policy is a probability distribution $p(s, a)$ the chance of taking a move a from the position s. Value function is the likelihood of winning from position s, denoted as $v(s)$. So there's a probability of making a move, and then a value of winning from that move. A single deep network f, of convolutional layers, estimates p and v. Takes board position s as input and outputs both p and v:

$$(p, v) = f_\theta(s) \tag{1}$$

$$p_a = Pr\{a|s\} \tag{2}$$

1.2 MCTS

In MCTS, we use a search tree to record all sequence of moves that we search (play). A node represents a board position and an edge represents a move. Starting with a board position, we search possible moves and evaluate policy and value function using the deep network f. After making an action, we expand the tree and add a new layer, same as above it has the probability of making an action and then the value of winning from that state. Repeat as necessary.

1.2.1 Action-Value function Q

Measures the value of making a move.

If we make a move and the probability of winning from there is 0.9 then $Q = 0.9$. Now the more steps you take Q is just the average of the value of each step. So if you make n moves and the value of winning from each move is $v_1, v_2, \ldots, v_n$ then:

$$Q(s, a) = \frac{1}{n} \sum_{i=1}^{n} v_i \tag{3}$$

However, Q also considers the previous number of times that the path has been traversed (refer to the example below):

up with a 0.9 chance of winning. So *Q* is 0.9. In (b), we make one more move and end up with a 0.6 chance of winning. Now, we have taken the move *a3* twice (visit count *N*=2). The *Q* value is simply the average of previous results, .i.e. *W*=(0.9+0.6), *Q* = *W*/2=0.75. In (c), we explore another path. Now, *a3* is picked 3 times and the *Q* value is (0.9+0.6+0.3)/3 = 0.6.

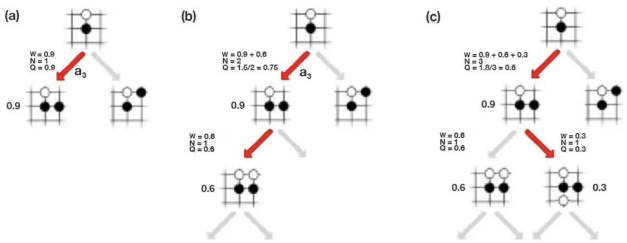

Figure 1: MCTS Q Value Calculation Example

Now to prioritize the search functionality so you don't keep searching the search space we consider exploitation and exploration:

- Exploitation: Perform more searches that look promising (i.e. high Q value).
- Exploration: Perform searches that are not frequently explored (i.e. low visit count N).

Mathematically the move a is selected according to:

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s,a) + u(s,a)) \tag{4}$$

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)} \tag{5}$$

In this equation, Q controls the exploitation and u (inverse to its visit count) controls the exploration. Starting from the root, we use this policy (tree policy) to select the path that we want to search further.

In the beginning, MCTS will focus more on exploration but as the iterations increase, most searches are exploitation and Q is getting more accurate.

The policy is actually derived from the visit count N:

$$\pi_a \propto N(s,a)^{1/\tau} \tag{6}$$

$$\pi(a|s) = \frac{N(s,a)^{t/\tau}}{\sum_b N(s,b)^{1/\tau}} \tag{7}$$

After the initial iterations, moves with higher Q value will be visited more frequently. We use the visit count to calculate the policy because it is less prone to outliers. τ is a temperature parameter, when $\tau \to 0$ the policy becomes greedy (only select the most visited move) so no exploration. MCTS improves our policy evaluation (improves Q), and we use the new evaluation to improve the policy (policy improvement). Then we re-apply the policy to evaluate the policy again. These repeated iterations of policy evaluation and policy improvement are called policy iteration in RL.

1.3 Self-Training MCTS

Starting with an empty board position $s1$, use MCTS to get a policy π_1. Then sample a move a_1 from π_1 and then repeat to get new policy π_2 and move a_2, etc.

This whole self-play game creates a sequence of input board position, policy and game result z (1 if win, -1 if lose).

$$(s_1 \to (\pi_1, z), s_2 \to (\pi_2, z), \ldots, s_T \to (\pi_T, z)) \tag{8}$$

This is the dataset we used to train the deep network f using supervised training. AlphaGo Zero plays games with itself to build up a training dataset. Then it randomly selects samples from the dataset to train f.

The next step is NN training. Loss function has:

- The Mean Square error between value function estimation v and true label z.
- Cross Entropy loss between policy p and MCTS improved policy π.

- $L2$ regularization on the network weights θ with $c = 0.0001$.

Some images for visualization:

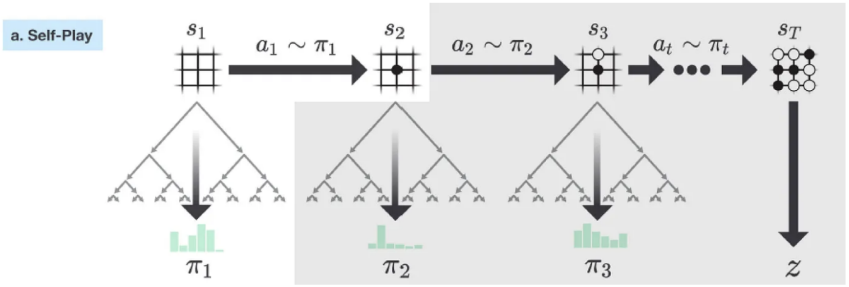

Figure 2: MCTS Self-Play Training Loop

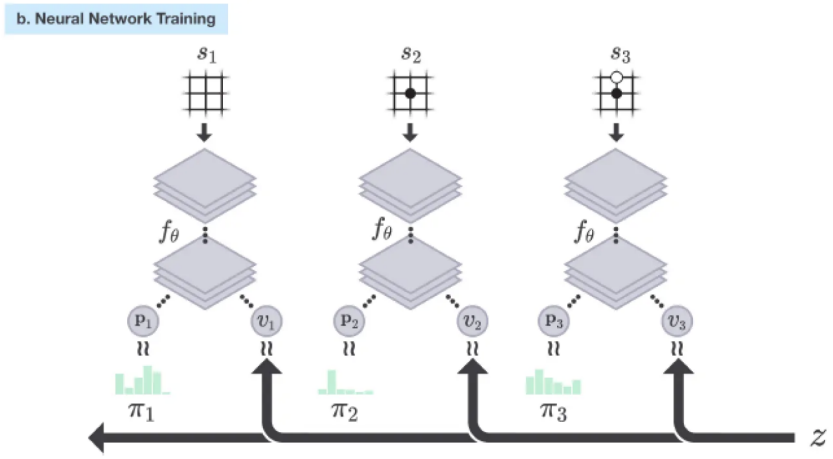

Figure 3: MCTS Search Tree Example

1.4 Self-Play Training Pipeline

Games used by the best models are used for the deep network f training. Pipeline contains 3 modules:

- Optimization: Using samples in the training dataset to optimize f and checkpoint models every 1,000 training iterations.
- Evaluator: If the MCTS using the new checkpoint models beats the current best model, we use it as the current best model instead.
- Play 25,000 games with the current best model and add them to the training dataset. Only the last 500,000 self-play games are kept for training.

Note that in the evaluator step we don't want exploration because we want to evaluate what's already been learned, so we set $\tau \to 0$ to make the policy greedy.

The network f is trained concurrently when AlphaGo is playing games with itself. Use policy and the value estimation from the network f, MCTS builds a search tree to formulate a more precise policy for the next move. In parallel, AlphaGo Zero lets MCTS play against each other using different version of the deep network f. Then it picks the best one as the current best model to create training samples needed to train f. Once the training is done, we can use the MCTS with the most optimal network f to plan for the next move in a real game. Once training is done use MCTS with most optimal network f to plan for next move in a real game.

2 Summary

This document reviewed how modern Monte Carlo Tree Search (MCTS) is combined with deep learning to produce strong game-playing agents, using AlphaGo Zero as the canonical example. The core idea is a single neural network f_θ that jointly predicts a policy $p(a \mid s)$ and a value $v(s)$ from a position s, and those predictions are used as priors and leaf evaluations inside MCTS. During tree traversal actions are chosen by maximizing a sum of an empirical action-value $Q(s, a)$ (exploitation) and an exploration bonus $u(s, a)$ derived from the network prior and the visit count. After many simulations the visit counts $N(s, a)$ define an improved policy π which is less sensitive to outliers than a single network forward pass.

Self-play with MCTS turns this procedure into a training pipeline: games are generated by repeatedly running MCTS from the current root, sampling moves from the improved visit-count policy, and storing (s, π, z) training tuples where z is the game outcome. The network is trained on these tuples with a loss that combines mean-squared error on the value (v vs. z), cross-entropy between predicted policy p and the MCTS policy π, and weight decay. Iterating this loop (self-play data collection, network optimization, and evaluator-based checkpoint selection) yields progressively stronger models.

Practically, the method balances exploration and exploitation via the $Q + u$ selection rule and uses a temperature parameter during sampling to control stochasticity early in training and to make greedy evaluations later. The combination of a learned prior (the network) with statistical search (MCTS) gives robust policy improvement, and using visit counts to form training targets stabilizes learning compared to training on single-network actions. These principles generalize beyond Go to other sequential decision problems where a fast policy/value estimator can be combined with limited search to improve performance.

3 Understanding AlphaGo Zero: UCB, MCST and UCT

3.1 Upper Confidence Bound (UCB)

Greedy action selection picks the action with highest estimated value:

$$A_t = \operatorname*{argmax}_a Q_t(a) \tag{9}$$

This ignores actions that may be inferior short-term but superior long-term. Epsilon-greedy fixes this by sometimes picking random actions with probability ϵ, but it doesn't model uncertainty in action value estimates.

3.1.1 Hoeffding's Inequality

For t independent, identically distributed random variables bounded between 0 and 1, Hoeffding's inequality states:

$$Pr[|\bar{X}_t - \mathbb{E}[X]| > m] \leq e^{-2tm^2} \tag{10}$$

The probability that sample mean differs from expected value by more than threshold m decreases exponentially with increasing sample size t and increasing threshold m.

3.1.2 UCB Policy Derivation

Apply Hoeffding's inequality to action values. The threshold m becomes upper confidence bound $U_t(a)$:

$$Pr[|Q_t(a) - q_*(a)| > U_t(a)] \leq e^{-2N_t(a)U_t(a)^2} \tag{11}$$

Setting this probability to small number $l = t^{-4}$ and solving for $U_t(a)$:

$$l = e^{-2N_t(a)U_t(a)^2} \tag{12}$$

$$U_t(a) = \sqrt{-\log l / 2N_t(a)} \tag{13}$$

$$U_t(a) = C\sqrt{\log t / N_t(a)} \tag{14}$$

Final UCB policy:

$$UCB(a) = \underset{a}{\operatorname{argmax}} \left(Q_t(a) + C\sqrt{\frac{\log t}{N_t(a)}} \right) \tag{15}$$

Parameter C quantifies degree of exploration. Large C increases exploration. However $U_t(a)$ decays over time since denominator $N_t(a)$ (visit count) increases faster than numerator $\log t$.

3.1.3 UCB1 Algorithm

Multi-armed bandit algorithm using UCB:

Algorithm 1 UCB1 Algorithm

Initialize:
p # arm pull number
a # possible actions
c # exploration parameter
$Q(a) = 0$ # action value estimates
$N(a) = 0$ # action selection counts
$t = 0$
for pull in range(p) **do**
  $t = t + 1$
  $A = \text{argmax}_a \left(Q(a) + c\sqrt{\frac{\ln(t)}{N(a)}} \right)$
  $R = \text{Bandit}(A)$ # receive reward
  $N(A) = N(A) + 1$
  $Q(A) = Q(A) + \frac{R - Q(A)}{N(A)}$ # update estimate
end for

3.2 Monte Carlo Search Tree (MCST)

MCST is a heuristic search algorithm for optimal decision making in games and sequential decision processes. Works for any domain described as (state, action) tuple.

3.2.1 Four Steps of MCST

1. Selection: Start at root node R (initial game state) and traverse tree according to current policy until reaching leaf node L. If L is:

- Terminal node (final game state): jump to step 4
- Non-terminal with unexplored children: continue to step 2

2. Expansion: Expand one child node C of leaf L.

3. Rollout: From node C, continue playing according to some policy (e.g., random) until reaching terminal node.

4. Update: At terminal node, game score is returned. Propagate score (add to current node value) through all visited nodes starting from C, through selection path, up to root R. Update both node value and visit count.

3.3 Upper Confidence Bound for Search Trees (UCT)

UCT combines MCST with UCB. During selection phase, evaluate child nodes using UCB formulation:

$$UCT(j) = \bar{X}_j + C\sqrt{\frac{\log(n_p)}{n_j}} \tag{16}$$

Where:

- $\bar{X}_j$ is average value of node (total score / visit count)
- C is exploration-exploitation trade-off constant
- n_p is parent node visit count
- n_j is node j visit count

3.3.1 Properties of UCT

- Requires minimal prior knowledge (only legal moves and terminal state scores)
- Focuses search towards most valuable branches
- Tunable for speed vs iteration count trade-off
- Can become slow for large combinatorial spaces

UCT successfully blocks opponent strategies and quickly develops effective counter-strategies in game play.

4 Final Remark

It is able to do this by using a novel form of reinforcement learning, in which AlphaGo Zero becomes its own teacher. The system starts off with a neural network that knows nothing about the game of Go. It then plays games against itself, by combining this neural network with a powerful search algorithm. As it plays, the neural network is tuned and updated to predict moves, as well as the eventual winner of the games.

This updated neural network is then recombined with the search algorithm to create a new, stronger version of AlphaGo Zero, and the process begins again. In each iteration, the performance of the system improves by a small amount, and the quality of the self-play games increases, leading to more and more accurate neural networks and ever stronger versions of AlphaGo Zero.