[2.5] Monte Carlo Tree Search and AlphaZero

David Quarel

ARENA

June 9, 2026

Assumptions

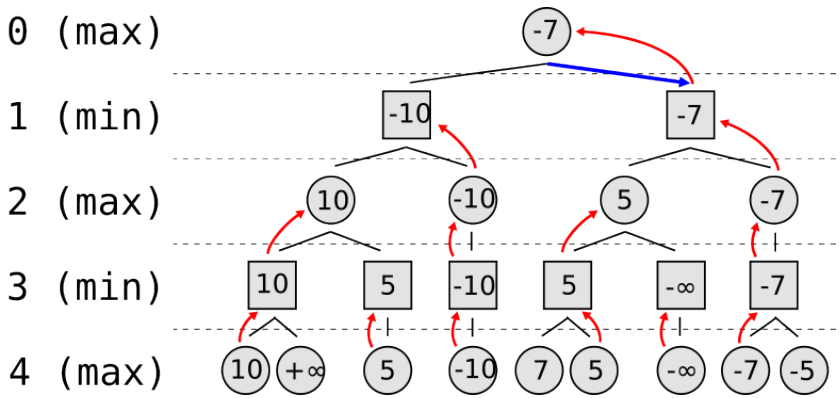
We consider **two-player zero-sum games**:

- **Two players:** Taking turns (e.g., White/Black)
- **Zero-sum:** One player's gain = other's loss
- **Deterministic:** No randomness in transitions
- **Perfect information:** Both players see full state
- **Known rules:** Player knows the game dynamics
- **Simulator access:** Can search/simulate future states

Payoffs: $z \in \{+1, 0, -1\}$ (win / draw / loss)
No discounting: $\gamma = 1$ (only final outcome matters)

Examples: Chess, Go, Checkers, Tic-Tac-Toe, Connect Four

Classical Approach: Minimax



Minimax: Assume opponent plays optimally

- MAX nodes: pick child with highest value
- MIN nodes: pick child with lowest value
- Recursively evaluate to leaves

Alpha-beta pruning: Skip branches that can't affect result

Motivation: Why MCTS?

- Minimax + alpha-beta works for Chess (≈ 35 moves/position)
- Fails for Go (≈ 250 moves/position!)
  - Branching factor too large
  - Hard to write evaluation function for non-terminal states

MCTS Idea: Don't search exhaustively—*sample* to estimate values.

Build the tree *asymmetrically*, focusing on promising moves.

MCTS: The Big Picture

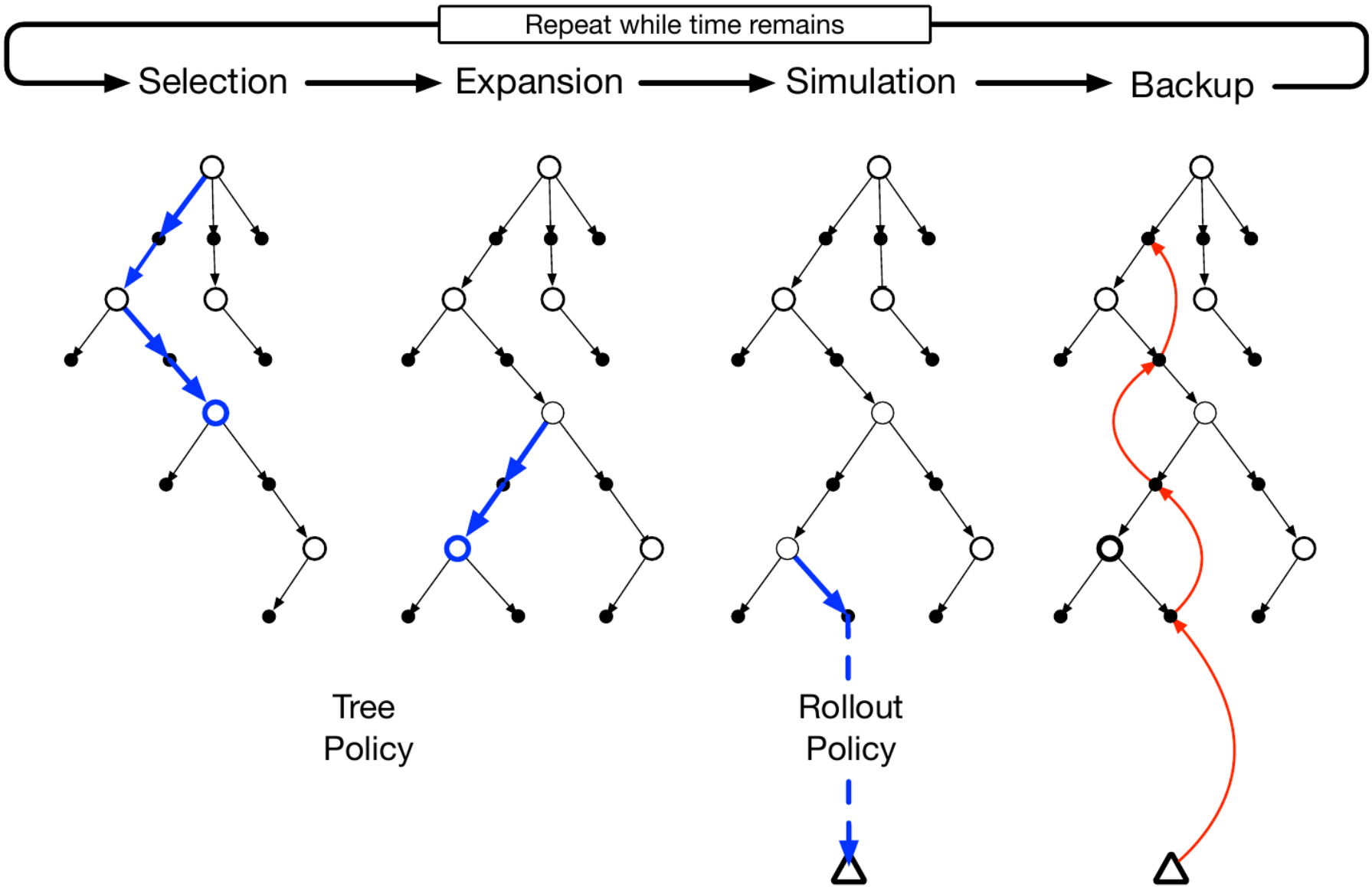
Build search tree incrementally, guided by simulation results.

Four phases (repeated many times):

1. **Selection** — Traverse the tree to a leaf node, guided by heuristic
2. **Expansion** — Add child node(s)
3. **Simulation** — Random rollout to terminal state
4. **Backpropagation** — Update statistics along path

Output: Action with highest visit count from root.

MCTS: Four Phases



Phase 1: Selection (UCT)

Select child nodes from root to leaf. Balance *exploitation* vs *exploration*.

UCB1 Formula

$$\text{UCB}(s,a) = \underbrace{\hat{Q}(s,a)}_{\text{exploit}} + \underbrace{c\sqrt{\frac{\ln N(s)}{N(s,a)}}}_{\text{explore}}$$

- $\hat{Q}(s,a)$ = average reward for action a in state s
- $N(s)$ = total visits to state s
- $N(s,a)$ = visits to state-action pair s,a
- **magic number** $c = \sqrt{2}$ typically

Theorem: UCT converges to minimax as simulations $\to \infty$.

UCB Intuition

$$\text{UCB}(s,a) = \underbrace{\hat{Q}(s,a)}_{\text{exploit}} + \underbrace{c\sqrt{\frac{\ln N(s)}{N(s,a)}}}_{\text{explore}}$$

Why this formula?

- Estimation error of $\hat{Q}(s,a)$ is $\propto \frac{1}{\sqrt{N(s,a)}}$
- Add bonus proportional to uncertainty $\Rightarrow$ explore high-variance actions
- $\ln N(s)$ in numerator ensures every action tried infinitely often

Behavior:

- Low $N(s,a)$: Large bonus $\Rightarrow$ explore
- High $N(s,a)$: Small bonus $\Rightarrow$ exploit $\hat{Q}$
- $\ln$ grows slowly, so exploitation eventually dominates

Phases 2-4: Expansion, Simulation, Backup

Expansion: Add one child node (untried action) to the leaf.

Simulation (Rollout): Play randomly to terminal state $\Rightarrow$ reward $z \in \{-1, 0, +1\}$

Backpropagation: Update all nodes on the path:

$$N(s) \leftarrow N(s) + 1$$
$$W(s) \leftarrow W(s) + z$$
$$\hat{Q}(s) = W(s)/N(s)$$

Note: In two-player games, negate z for opponent's nodes.

MCTS: Pseudocode

```
1: function MCTS(s_0, num_simulations)
2:     Create root node with state s_0
3:     for i = 1 to num_simulations do
4:         v_l ← TreePolicy(root)              ▷ Select + Expand
5:         z ← Rollout(s(v_l))                 ▷ Simulate
6:         Backup(v_l, z)                      ▷ Backprop
7:     end for
8:     return arg max_a N(s_0, a)
9: end function
```

TreePolicy: UCB selection → expand leaf.

Rollout: Random play to terminal.

MCTS: Properties

Advantages:

- **Anytime:** More iterations = better results
- **Aheuristic:** No hand-crafted evaluation needed
- **Asymmetric:** Focuses on promising branches
- **Parallelizable:** Multiple simulations concurrently

Limitations:

- Random rollouts can be weak
- Need a better evaluation function for non-terminal states

MCTS Parallelization

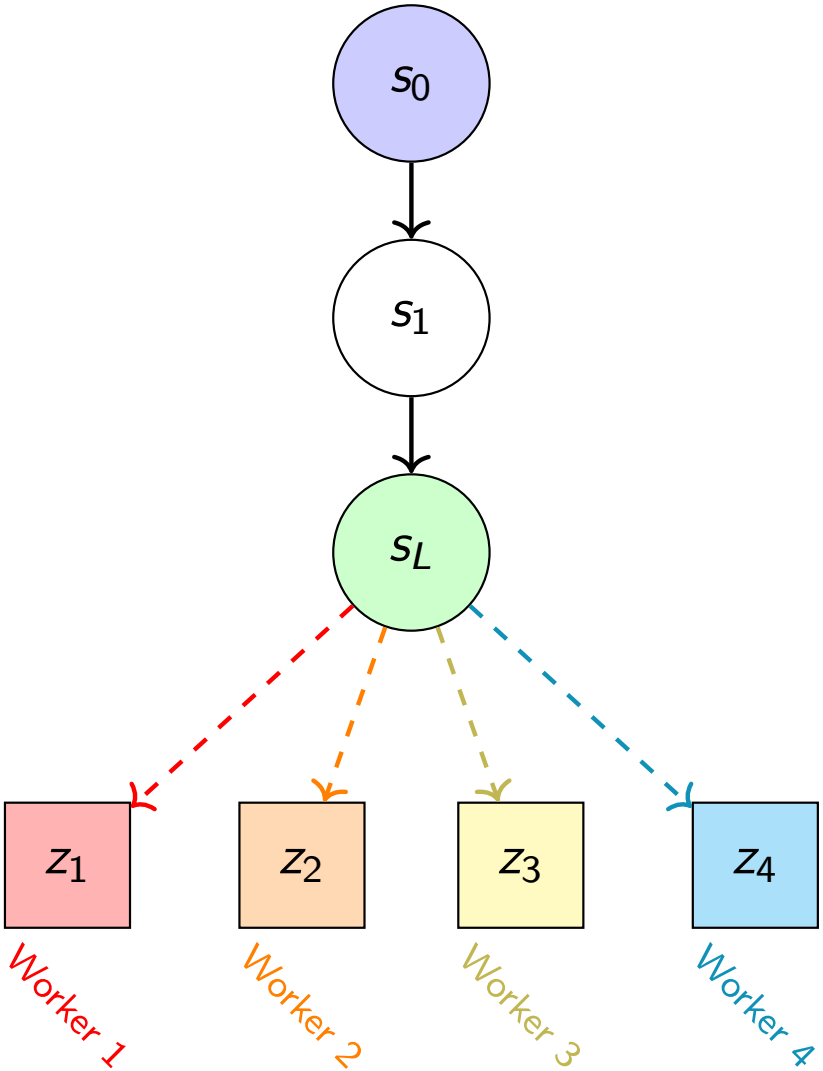
MCTS is naturally parallelizable—but *how* matters!

Three main strategies:

1. **Leaf Parallelization**
2. **Root Parallelization**
3. **Tree Parallelization**

Trade-off: *Implementation complexity* vs *Speedup efficiency*

Leaf Parallelization



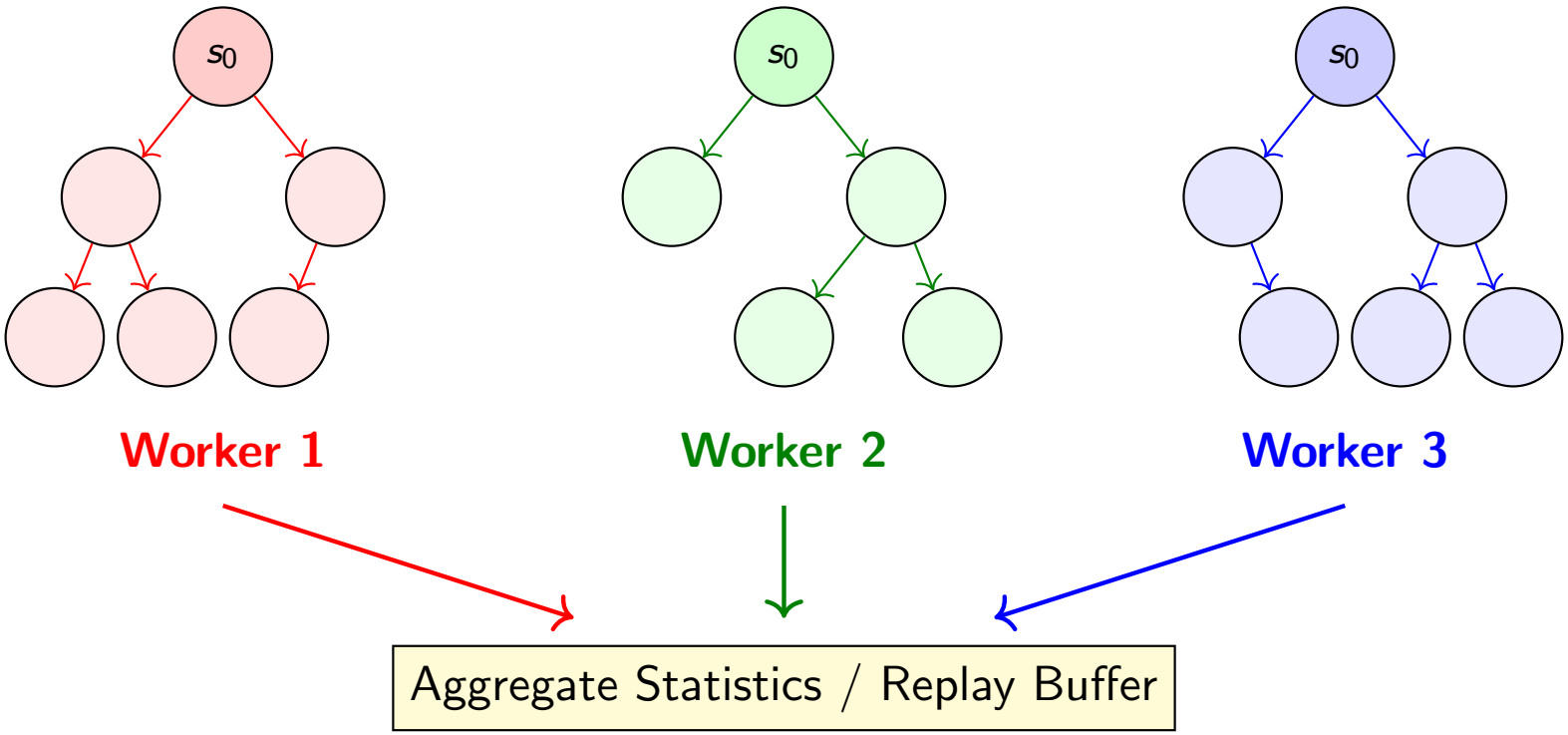
Idea: Multiple workers run rollouts from the same leaf node in parallel

✓ Simple to implement
✓ No synchronization needed

× Limited speedup
× Workers idle during selection
× All explore same branch

Root Parallelization

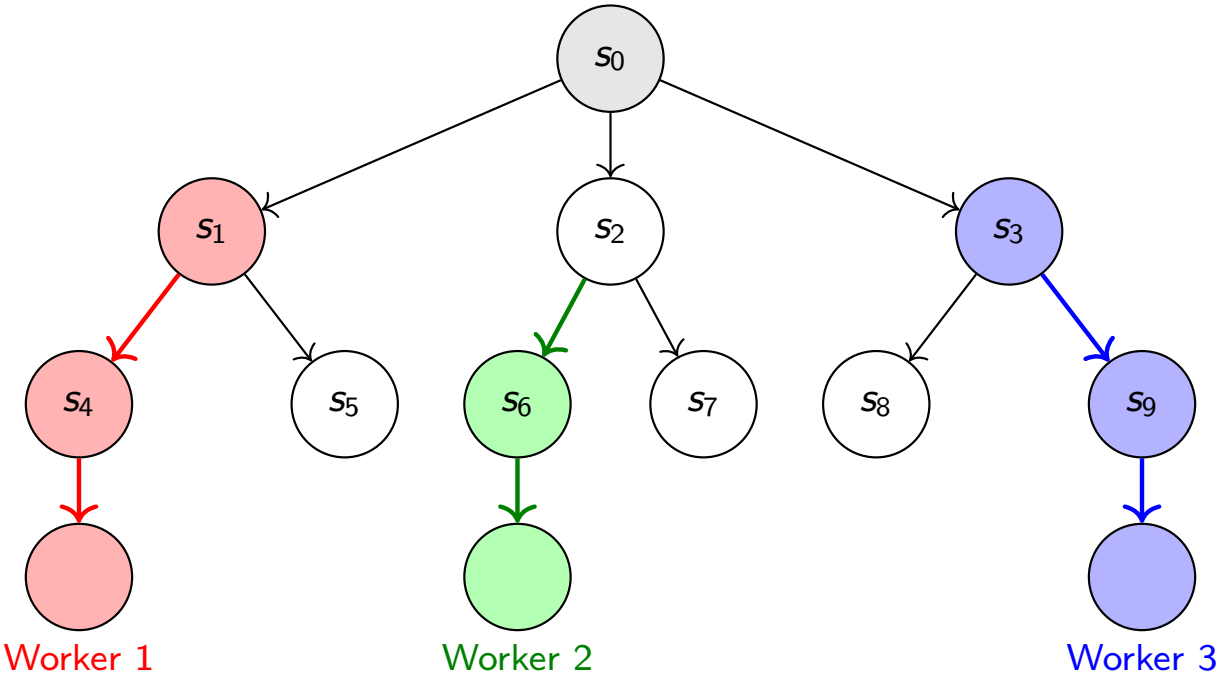
Each worker builds independent tree — merge results at the end



✓ No synchronization ✓ Scales to clusters × Duplicated work across trees

Tree Parallelization

Multiple workers share one tree — different branches simultaneously



✓ No duplicated work—best compute efficiency

× Requires synchronization (locks/atomics)

Tree Parallelization: Synchronization

Problem: Multiple threads read/write $N(s), W(s)$ concurrently

Option 1: Mutex / Locks

- Lock node before read/write, unlock after
- Simple but high contention at popular nodes (root!)

Option 2: Atomic Operations

- Use atomic increment: `atomic_fetch_add(&N, 1)`
- Lock-free, but limited to simple operations

Option 3: Lock-Free with Virtual Loss

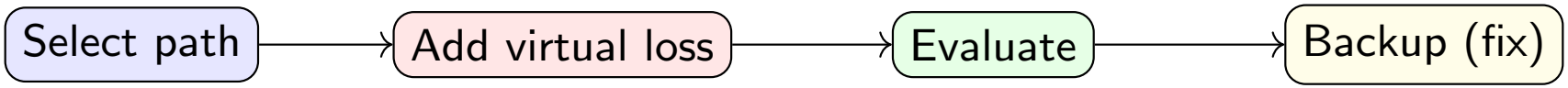
- Atomically add virtual loss during selection
- No locks needed—threads naturally spread out
- **AlphaZero uses this approach**

Virtual Loss: Lock-Free Synchronization

Problem: Threads all select the same "best" node $\Rightarrow$ wasted work

Solution: When thread selects node s, *atomically* apply virtual loss:

$$N(s) \xleftarrow{\text{atomic}} N(s) + 1, \quad W(s) \xleftarrow{\text{atomic}} W(s) - 1$$



Effect: Node looks worse $\Rightarrow$ other threads explore elsewhere

Mutexes mostly not a problem! Still have to use them in some cases. Atomic ops provide thread safety.

AlphaZero: Key Innovations

Two changes to MCTS:

1. **Neural network replaces rollouts**
   - Directly estimate value v — no random simulation

2. **Learned prior guides search**
   - Policy $\mathbf{p}$ focuses on promising moves

$\Rightarrow$ Far fewer simulations needed (800 vs 10,000+)

Neural Network Architecture

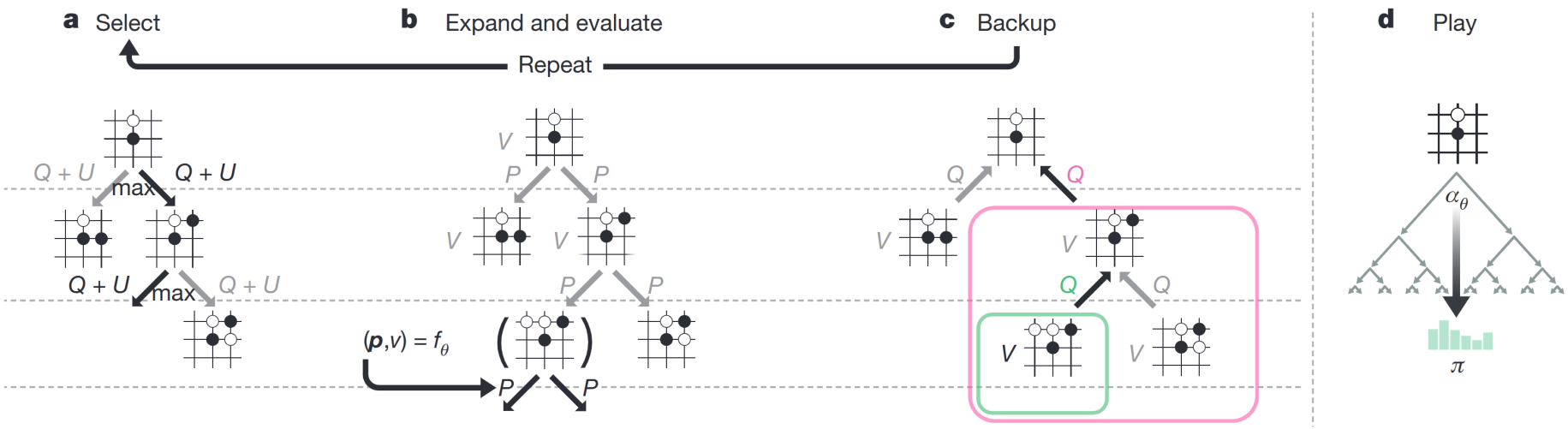
Single network with two heads:

$$f_{\theta}(s) = \big(P_{\theta}(\cdot|s),\ \hat{V}_{\theta}(s)\big)$$

Policy head $P_{\theta}(\cdot\|s)$	Prob. distribution over actions $\Rightarrow$ guides MCTS selection
Value head $\hat{V}_{\theta}(s) \in [-1, +1]$	Estimated win probability $\Rightarrow$ replaces rollouts

Architecture: ResNet with two linear heads

AlphaZero MCTS



Key difference: No simulation/rollout—$\hat{V}_{\boldsymbol{\theta}}(s)$ provides backup signal directly.

PUCT: Selection with Prior

AlphaZero uses **PUCT** instead of UCB:

PUCT Tree Policy

$$a^* = \arg\max_a \left[Q(s,a) + c_{\text{puct}} \cdot P_\theta(a|s) \cdot \frac{\sqrt{1+N(s)}}{1+N(s,a)} \right]$$

Key difference: Prior $P_\theta(a|s)$ from network weights exploration

- Initially: explore actions network likes ($P_\theta(a|s)$ high)
- Later: $Q(s,a)$ dominates (exploitation)
- Search can override prior if data disagrees

Result: Network provides intuition, MCTS refines it.

MCTS Policy: π

After running MCTS, we extract an **improved policy** from visit counts:

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}$$

Temperature τ controls exploration:

- $\tau = 1$: $\pi(a|s) \propto N(s,a)$ (proportional to visits)
- $\tau \to 0$: $\pi(a|s) \to \mathbf{1}[a = \arg\max_{a'} N(s,a')]$ (deterministic)

Key insight: $\boldsymbol{\pi}$ is *better* than network's prior $P_{\boldsymbol{\theta}}(\cdot|s)$!

- MCTS refines the policy through search
- Use $\boldsymbol{\pi}$ as training target to improve the network

AlphaZero: The Loss Function

The neural network $f_\theta(s) = \left(P_\theta(\cdot|s), \hat{V}_\theta(s)\right)$ is trained to minimize:

AlphaZero Loss

$$L(\theta) = \underbrace{(z - \hat{V}_\theta)^2}_{\text{value loss}} + \underbrace{(-\boldsymbol{\pi}^T \log \mathbf{p}_\theta)}_{\text{policy loss}} + \underbrace{c\|\theta\|^2}_{\text{L2 reg.}}$$

Training data from self-play:

- $\boldsymbol{\pi}$ = MCTS visit distribution (target policy)
- $z \in \{-1, +1\}$ = game outcome (target value)

Loss Function: Components

AlphaZero Loss (Expanded)

$$L(\theta) = (z - \hat{V}_{\theta}(s))^2 - \sum_{a \in \mathcal{A}} \pi(a|s) \log P_{\theta}(a|s) + c\|\theta\|^2$$

Value Loss (MSE): learn to predict game outcome

$$L_v = (z - \hat{V}_{\theta}(s))^2$$

Policy Loss (Cross-entropy): distill MCTS into network

$$L_{\pi} = -\boldsymbol{\pi}^T \log \mathbf{p}_{\theta} = -\sum_{a \in \mathcal{A}} \pi(a|s) \log P_{\theta}(a|s)$$

L2 Regularization: prevent overfitting

$$L_{\text{reg}} = c\|\theta\|^2$$

Policy Loss: Why Cross-Entropy?

We want the network policy $\mathbf{p}_{\boldsymbol{\theta}}$ to match MCTS policy $\boldsymbol{\pi}$:

$$L_\pi = D_{\mathrm{KL}}(\boldsymbol{\pi} \| \mathbf{p}_{\boldsymbol{\theta}}) = \sum_a \pi(a) \log \frac{\pi(a)}{P_{\boldsymbol{\theta}}(a)}$$

Expanding and differentiating:

$$\nabla_{\boldsymbol{\theta}} L_\pi = \nabla_{\boldsymbol{\theta}} \left[\underbrace{\sum_a \pi(a) \log \pi(a)}_{-H(\boldsymbol{\pi}),\ \text{const. w.r.t. } \boldsymbol{\theta}} - \sum_a \pi(a) \log P_{\boldsymbol{\theta}}(a) \right]$$

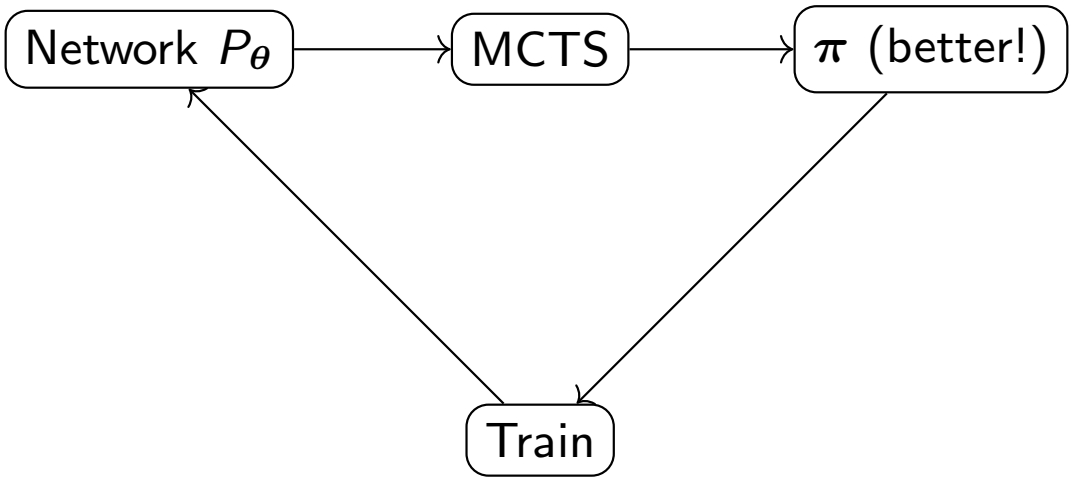
$$= -\nabla_{\boldsymbol{\theta}} \sum_a \pi(a) \log P_{\boldsymbol{\theta}}(a)$$

$\Rightarrow$ Minimizing KL $\Leftrightarrow$ minimizing **cross-entropy**:

$$L_\pi = -\sum_a \pi(a|s) \log P_{\boldsymbol{\theta}}(a|s)$$

Policy Improvement via Distillation

Key insight: MCTS improves the policy!



Better network $\rightarrow$ better MCTS $\rightarrow$ better targets $\rightarrow$ better network...

Self-Play Training Loop

1: Initialize network f_θ randomly
2: **loop**
3: **Self-play:** Run MCTS, collect $(s, \boldsymbol{\pi}, z)$
4: **Train:** Minimize $L(\boldsymbol{\theta})$ on minibatch
5: **end loop**

Temperature τ controls move selection: $\pi(a) \propto N(s,a)^{1/\tau}$

- $\tau = 1$: Sample proportional to visits (exploration)
- $\tau \to 0$: Pick most visited (exploitation)

AlphaZero: $\tau = 1$ for first 30 moves, then $\tau \to 0$.

Summary

MCTS: Select $\rightarrow$ Expand $\rightarrow$ Simulate $\rightarrow$ Backup

AlphaZero: Replace simulation with neural network $f_{\boldsymbol{\theta}}(s) = \left(P_{\boldsymbol{\theta}}(\cdot|s), \hat{V}_{\boldsymbol{\theta}}(s)\right)$

Loss:

$$L = \underbrace{(z - \hat{V}_{\boldsymbol{\theta}})^2}_{\text{value}} + \underbrace{(-\boldsymbol{\pi}^T \log \mathbf{p}_{\boldsymbol{\theta}})}_{\text{policy}} + \underbrace{c\|\boldsymbol{\theta}\|^2}_{\text{reg}}$$

Parallelization: Leaf, Root, or Tree (with virtual loss)

References

- Sutton & Barto, *Reinforcement Learning: An Introduction*, Chapter 8.11 http://incompleteideas.net/book/the-book-2nd.html
- Silver et al., *Mastering the game of Go with deep neural networks and tree search* (Nature, 2016) https://www.nature.com/articles/nature16961
- Silver et al., *Mastering the game of Go without human knowledge* (Nature, 2017) https://www.nature.com/articles/nature24270
- Silver et al., *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play* (Science, 2018) https://www.science.org/doi/10.1126/science.aar6404
- Browne et al., *A Survey of Monte Carlo Tree Search Methods* (2012) https://ieeexplore.ieee.org/document/6145622