Planning and Learning

CMPS 4660/6660: Reinforcement Learning

Acknowledgement: slides adapted from David Silver's RL course

Agenda

- Model-Based Reinforcement Learning
- Integrated Architectures
- Simulation-Based Search

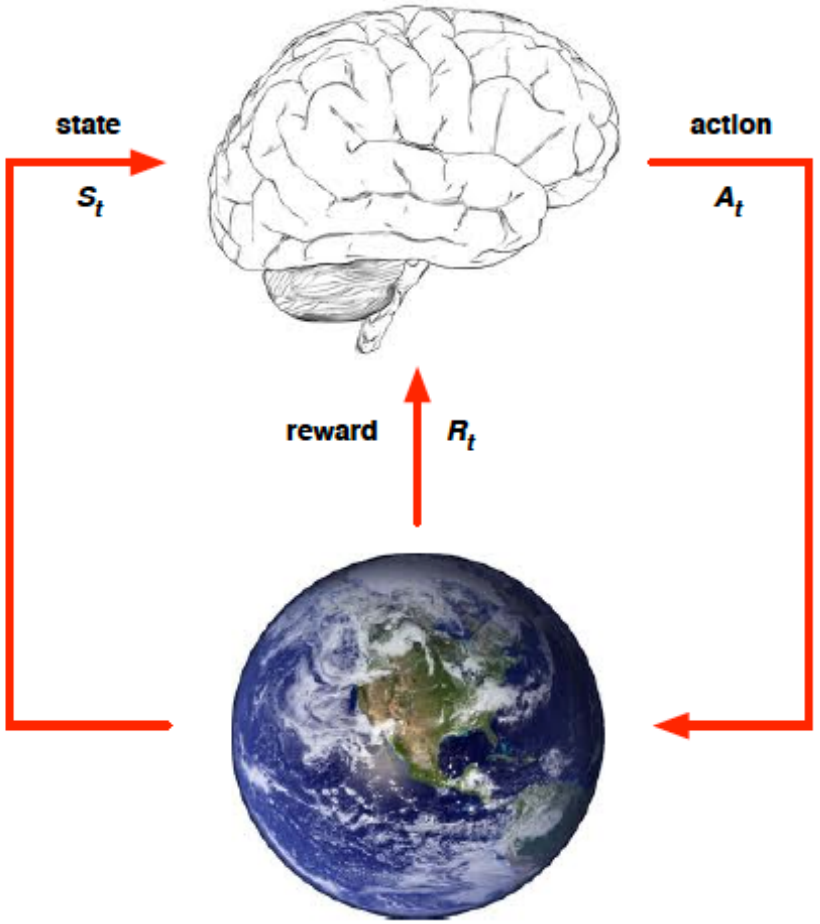
Model-Based Reinforcement Learning

- Policy gradient: learn policy directly from experience
- Value function approximation: learn value function directly from experience
- Model-based RL: learn model directly from experience
  - use planning to construct a value function or policy
  - Integrate learning and planning into a single architecture

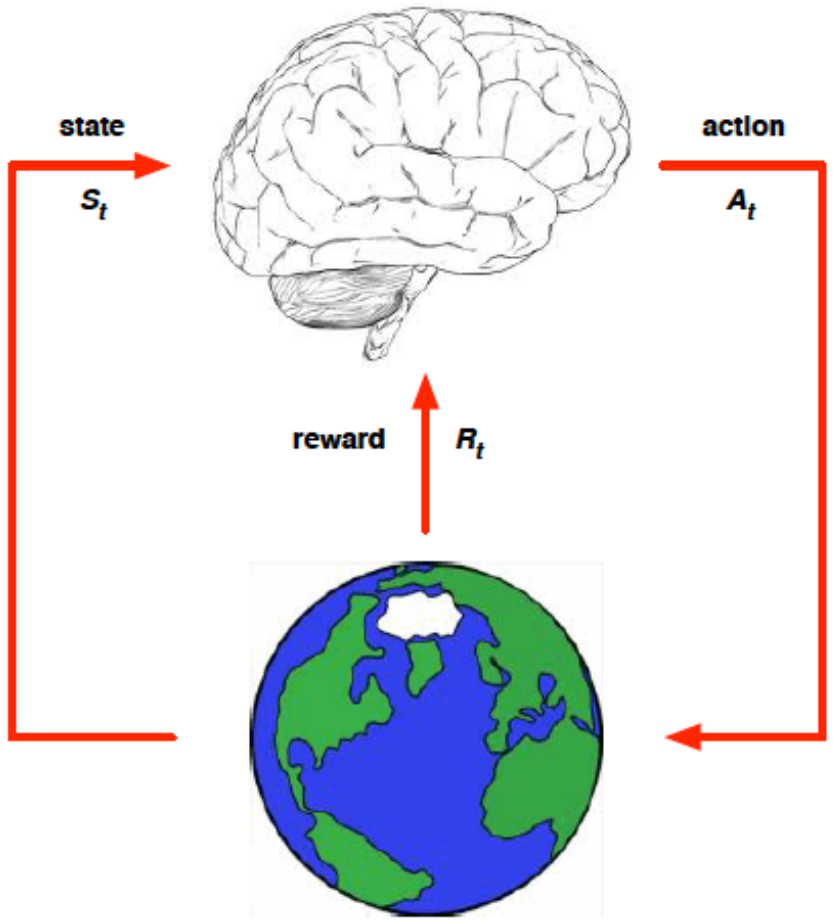
Model-Based vs. Model-Free RL

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience
- Model-Based RL
  - Learn a model from experience
  - Plan value function (and/or policy) from model

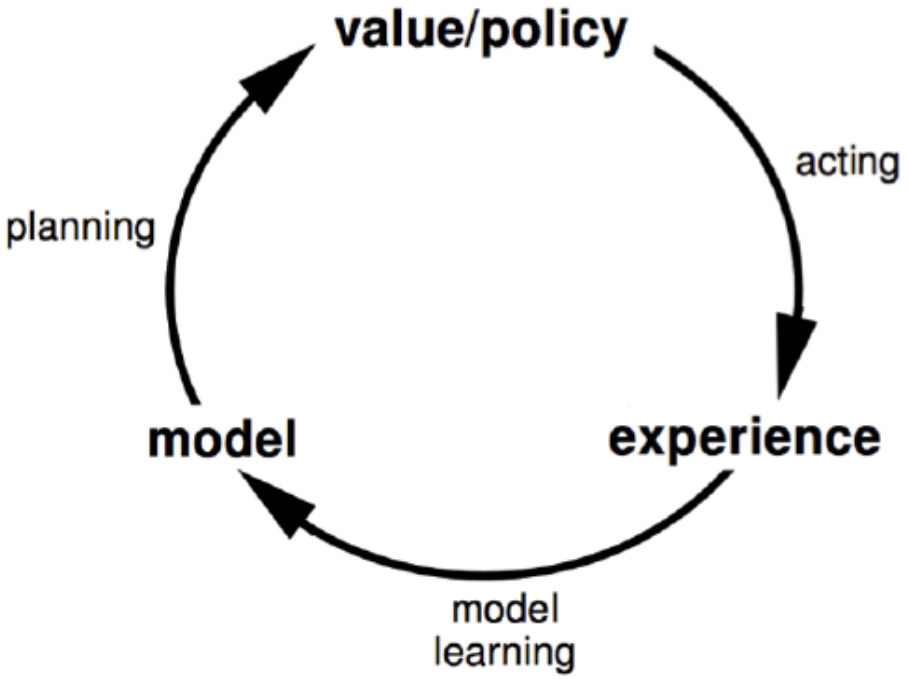
Model-Free RL



Model-Based RL



Model-Based RL



Advantages of Model-Based RL

- Advantages:
  - Can efficiently learn model by supervised learning methods
  - Can reason about model uncertainty
- Disadvantages:
  - First learn a model, then construct a value function

    ➡ two sources of approximation error

What is a Model?

- A model predicts what the environment will do next
- P predicts the next state
- r predicts the next (immediate) reward, e.g.

$$P_{ss'}(a) = \Pr\{S_t = s' | S_{t-1} = s, A_{t-1} = a\}$$

$$r(s,a) = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a]$$

What is a Model?

- A model M is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, P, r \rangle$ parametrized by η
- We will assume state space $\mathcal{S}$ and action space $\mathcal{A}$ are known
- So a model $M = \langle P_\eta, r_\eta \rangle$ represents state transitions $P_\eta \approx P$ and rewards $r_\eta \approx r$

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$

$$r_{t+1} = r_\eta(S_t, A_t)$$

- Typically assume conditional independence between state transitions and rewards

$$\Pr[S_{t+1}, R_{t+1}|S_t, A_t] = \Pr[S_{t+1}|S_t, A_t] \Pr[R_{t+1}|S_t, A_t]$$

Model Learning

- Goal: estimate model M_η from experience $\{S_0, A_0, R_1, \ldots, S_T\}$
- This is a supervised learning problem

$$S_0, A_0 \rightarrow R_1, S_1$$
$$S_1, A_1 \rightarrow R_2, S_2$$
$$\vdots$$
$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- Learning $s, a \rightarrow r$ is a regression problem
- Learning $s, a \rightarrow s$ is a density estimation problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters η that minimize empirical loss

Examples of Models

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Deep Belief Network Model
- ...

Table Lookup Model

- Model is an explicit MDP
- Count visits $N(s,a)$ to each state action pair

$$\hat{P}_{ss'}(a) = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t, S_{t+1} = s, a, s')$$

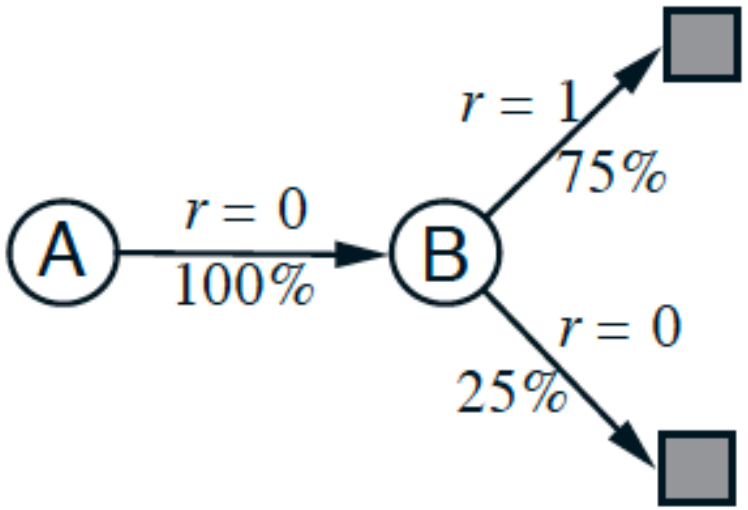
$$\hat{r}(s,a) = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t = s, a) R_t$$

- Alternatively
  - At each time-step t, record experience tuple $\langle S_t, A_t, R_t, S_{t+1} \rangle$
  - To sample model, randomly pick tuple matching $\langle s, a, \cdot, \cdot \rangle$

AB Example

Two states A, B; no discounting; 8 episodes of experience

$A, 0, B, 0$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 0$



- We have constructed a table lookup model from the experience

Planning with a Model

- Given a model $M_\eta = \langle P_\eta, r_\eta \rangle$
- Solve the MDP $\langle \mathcal{S}, \mathcal{A}, P_\eta, r_\eta \rangle$
- Using favorite planning algorithm
  - Value iteration
  - Policy iteration
  - Tree search
  - …

Sample-Based Planning

- A simple but powerful approach to planning
- Use the model only to generate samples
- Sample experience from model

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$

$$r_{t+1} = r_\eta(S_t, A_t)$$

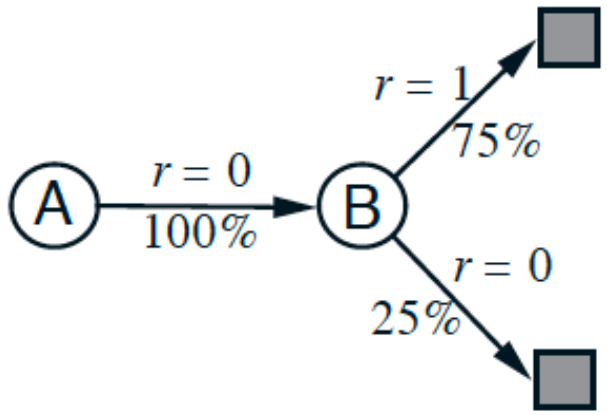
- Apply model-free RL to samples, e.g.:
  - Monte-Carlo control
  - Sarsa
  - Q-learning
- Sample-based planning methods are often more efficient

Back to the AB Example

- Construct a table-lookup model from real experience
- Apply model-free RL to sampled experience

Real experience

$A, 0, B, 0$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 0$



Sampled experience

$B, 1$
$B, 0$
$B, 1$
$A, 0, B, 1$
$B, 1$
$A, 0, B, 1$
$B, 1$
$B, 0$

- e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

Planning with an Inaccurate Model

- Given an imperfect model $\langle P_\eta, r_\eta \rangle \neq \langle P, r \rangle$
- Performance of model-based RL is limited to optimal policy for approximate MDP $\langle \mathcal{S}, \mathcal{A}, P_\eta, r_\eta \rangle$
  - i.e. Model-based RL is only as good as the estimated model
- When the model is inaccurate, planning process will compute a suboptimal policy
  - Solution 1: when model is wrong, use model-free RL
  - Solution 2: reason explicitly about model uncertainty

Agenda

- Model-Based Reinforcement Learning
- Integrated Architectures
- Simulation-Based Search

Real and Simulated Experience

- We consider two sources of experience
- Real experience Sampled from environment (true MDP)

$$S' \sim P_{ss'}(a)$$
$$r = r(s, a)$$

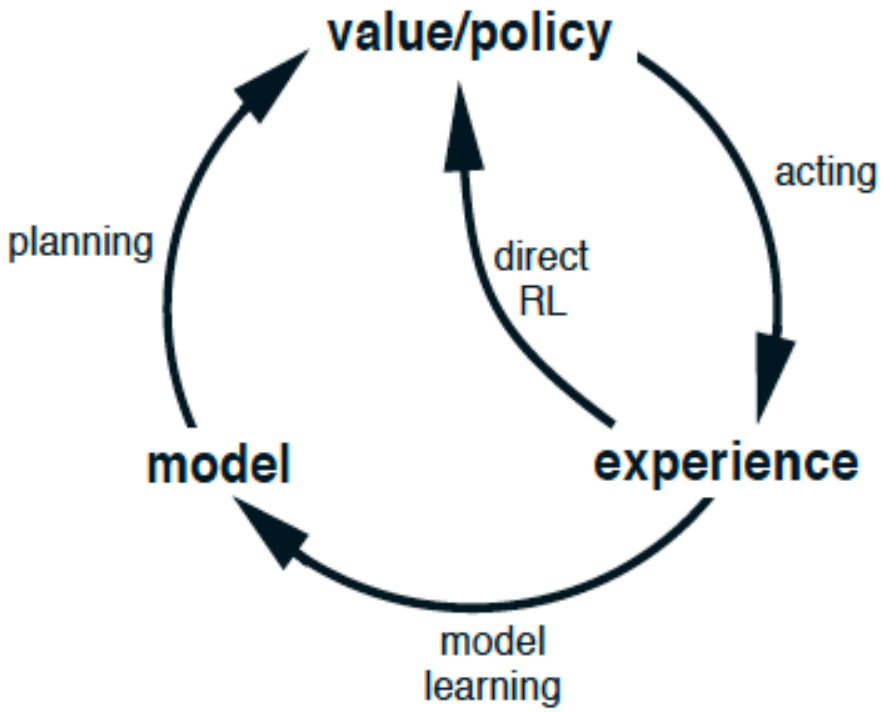
- Simulated experience Sampled from model (approximate MDP)

$$S' \sim P_\eta(s'|s, a)$$
$$r = r_\eta(s, a)$$

Integrating Learning and Planning

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
  - Learn a model from experience
  - Plan value function (and/or policy) from simulated experience
- Dyna
  - Learn a model from real experience
  - Learn and plan value function (and/or policy) from real and simulated experience

Dyna Architecture



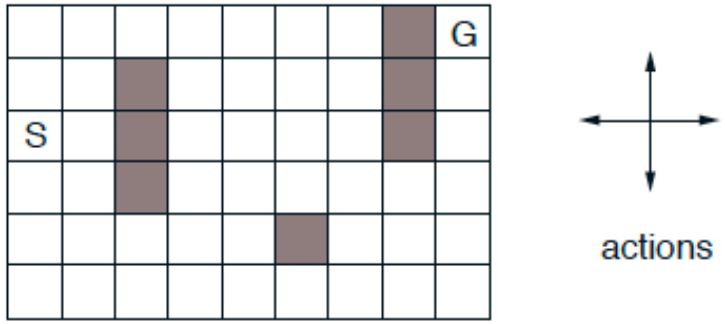
Dyna-Q Algorithm

Tabular Dyna-Q

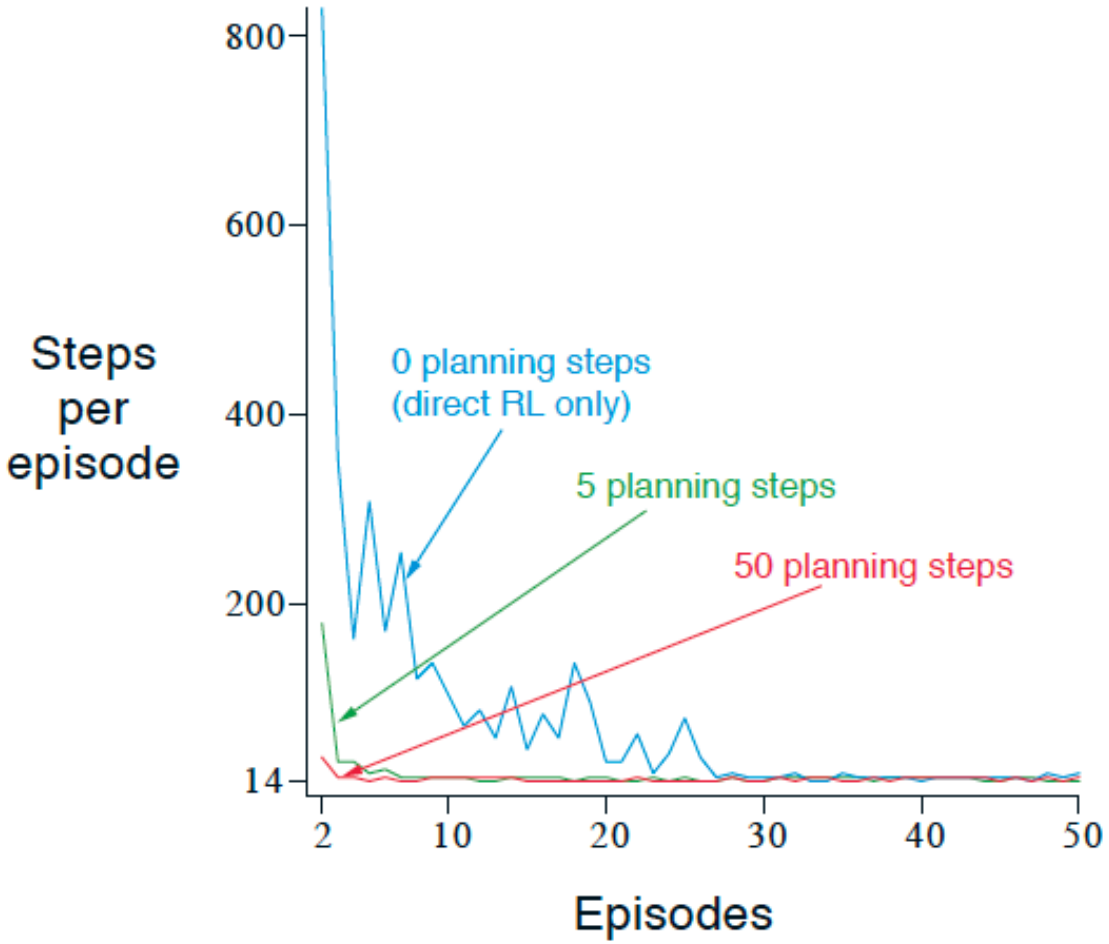
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

planning steps

Dyna-Q on a Simple Maze

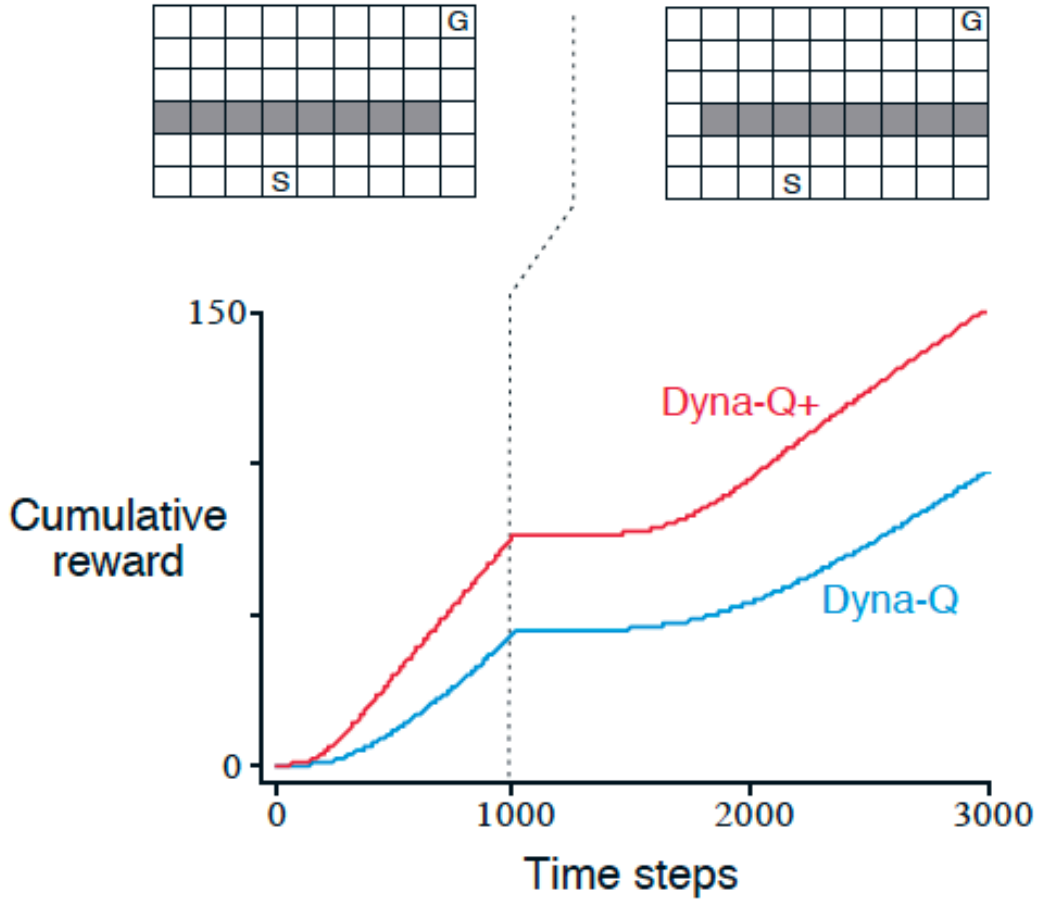


- 47 states, 4 actions
- The agent remains where it is when movement is blocked by an obstacle or the edge of the maze
- Reward is zero on all transitions, except those into the goal state, on which it is +1.
- $\gamma = 0.95$



Dyna-Q with an Inaccurate Model

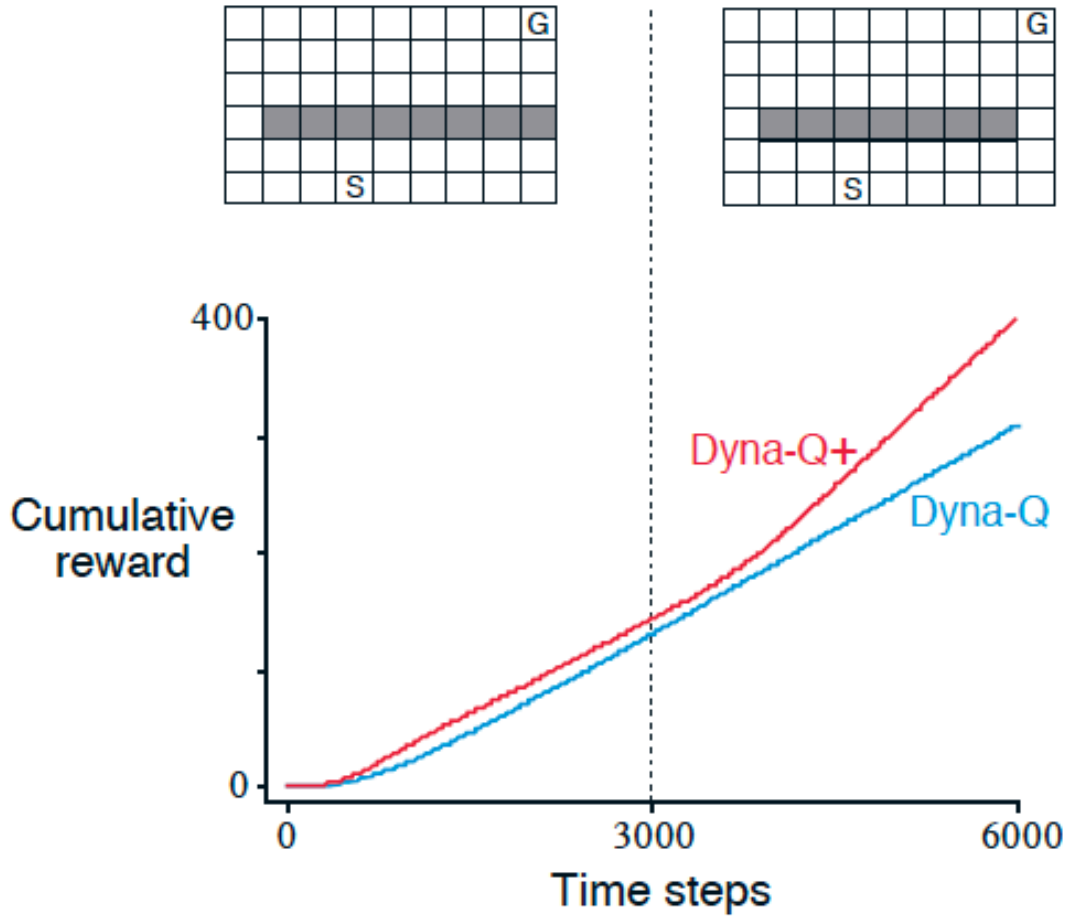
- The changed environment is harder



Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration

Dyna-Q with an Inaccurate Model (2)

- The changed environment is easier



Dyna-Q agent never switched to the shortcut

Agenda

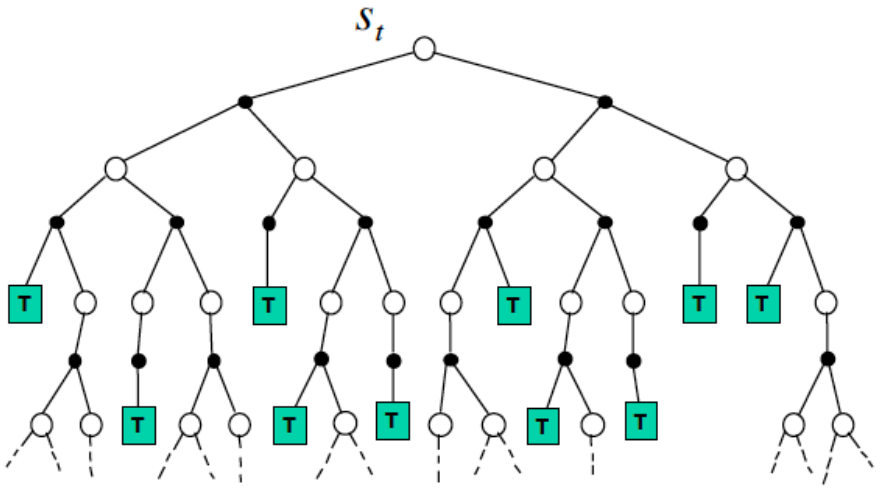
- Model-Based Reinforcement Learning
- Integrated Architectures
- Simulation-Based Search

Background and Decision-Time Planning

- Background planning
  - Examples: dynamic programming, Dyna
  - Use planning to gradually improve a policy or value function
  - Planning has played a part well before an action is selected for the current state
  - Generally better for applications that require low latency action selection
- Decision-time planning
  - Examples: heuristic search, Monte-Carlo search, TD search
  - Begin and complete planning after encountering each new state
  - Values and policy are specific to the current state and the action choices available
  - and are typically discarded after being used to select the current action
  - Most useful in applications in which fast responses are not required
- The two ways of thinking about planning can blend together in natural and interesting ways

Forward Search

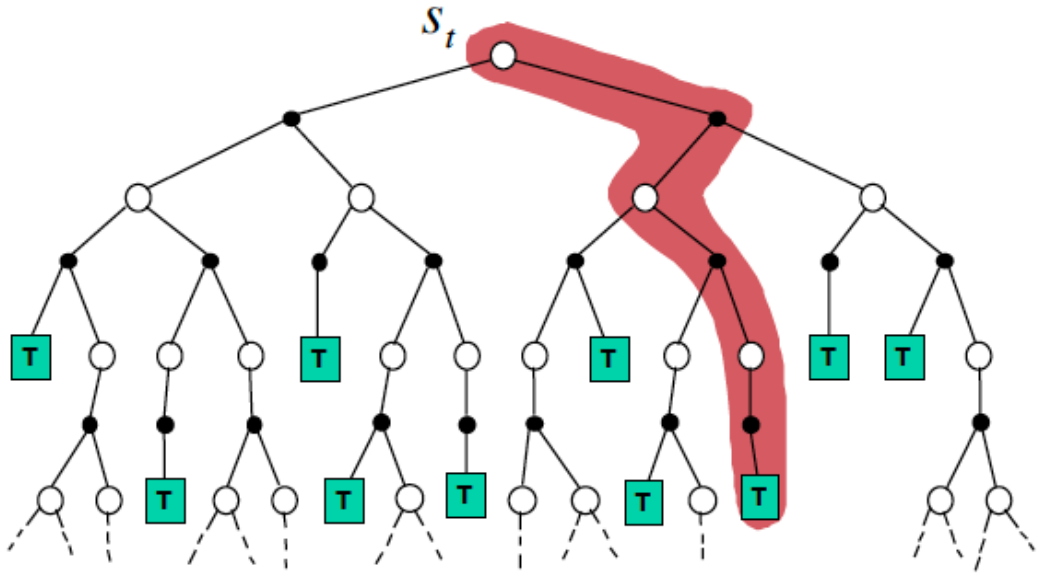
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- No need to solve whole MDP, just sub-MDP starting from now

Simulation-Based Search

- Forward search paradigm using sample-based planning
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



Simulation-Based Search (2)

- Simulate episodes of experience from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\eta$$

- Apply model-free RL to simulated episodes
  - Monte-Carlo control → Monte-Carlo search
  - Sarsa → TD search

Simple Monte-Carlo Search (Rollout)

- Given a model M_η and a simulation policy (rollout policy) π
- For each action $a \in A$
  - Simulate K episodes from current (real) state s_t

$$\{s_t^k, a, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\eta, \pi$$

  - Evaluate actions by mean return (Monte-Carlo evaluation)

$$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^{K} G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s_t, a)$$

What Can Rollout Accomplish

- Consider two stationary policies π and π'
- The Policy Improvement Theorem indicates that if

  (1) π and π' are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s

  (2) $q_\pi(s, a) > v_\pi(s)$

  Then π' is better than π
- This applies to rollout algorithms where s is the current state and π is the rollout policy
  - average returns of simulated trajectories to produce estimates of $q_\pi(s, a)$
  - select an action a that maximizes $q_\pi(s, a)$ to improve π
  - Like asynchronous value iteration

Monte-Carlo Tree Search (Evaluation)

- Given a model M_η
- Simulate K episodes from current (real) state s_t using current simulation policy π

$$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\eta, \pi$$

- Build a search tree containing visited states and actions
- Evaluate states $Q(s,a)$ by mean return of episodes from s, a

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(S_u, A_u = s, a) G_u \xrightarrow{P} q_\pi(s,a)$$

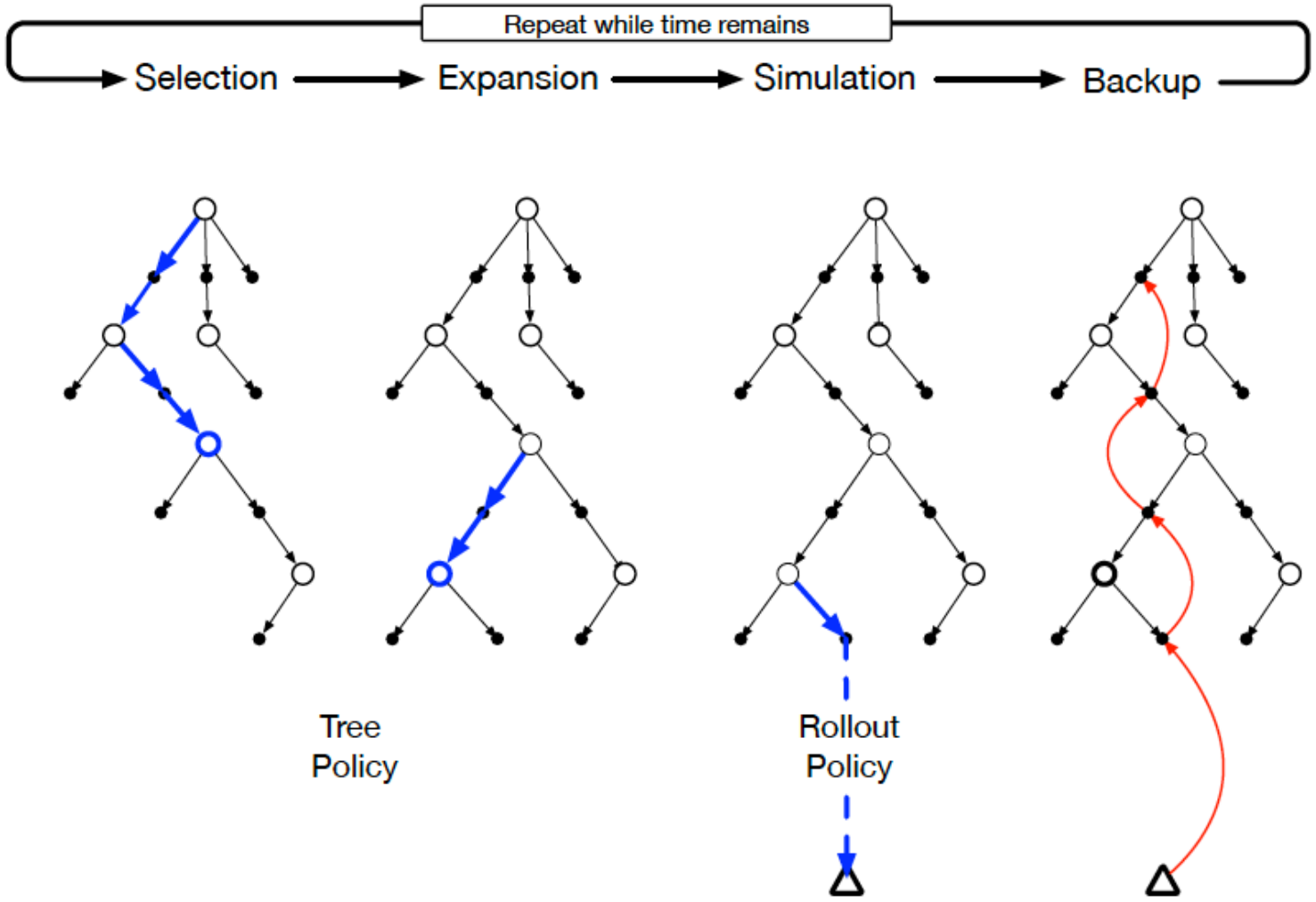
- After search is finished, select current (real) action with maximum value in search tree

$$a_t = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s_t, a)$$

Monte-Carlo Tree Search (Simulation)

- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximize $Q(S, A)$
    - needs to balance exploration and exploitation: e.g., ϵ-greedy or UCB
  - Rollout policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by ϵ-greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$

Monte-Carlo Tree Search



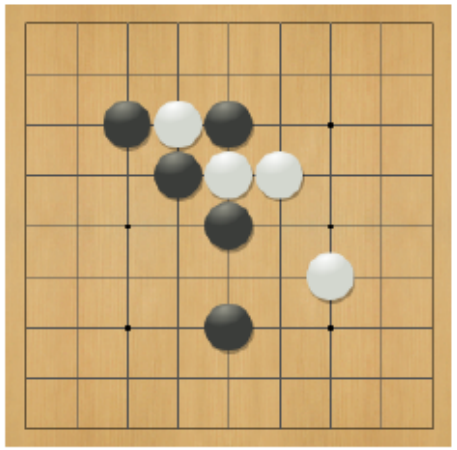
- MCTS repeats the four steps, starting each time at the tree's root node, as many iterations as possible
- Finally picks an action according to accumulated statistics in the root node's outgoing edges
- After transitioning to a new state, MCTS is run again with the root node set to the new current state
- The search tree at the start of the new execution might be just this new root node, or it might include descendants of this node left over from the previous execution

Case Study: the Game of Go

- The ancient oriental game of Go is 2500 years old
- Considered to be the hardest classic board game
- Considered a grand challenge task for AI (John McCarthy)
- Traditional game-tree search has failed in Go

Rules of Go

- Usually played on 19x19, also 13x13 or 9x9 board
- Simple rules, complex strategy
- Black and white place down stones alternately
- Surrounded stones are captured and removed
- The player with more territory wins the game



Position Evaluation in Go

- How good is a position s?
- Reward function (undiscounted):

$$R_t = 0 \text{ for all non-terminal steps } t < T$$

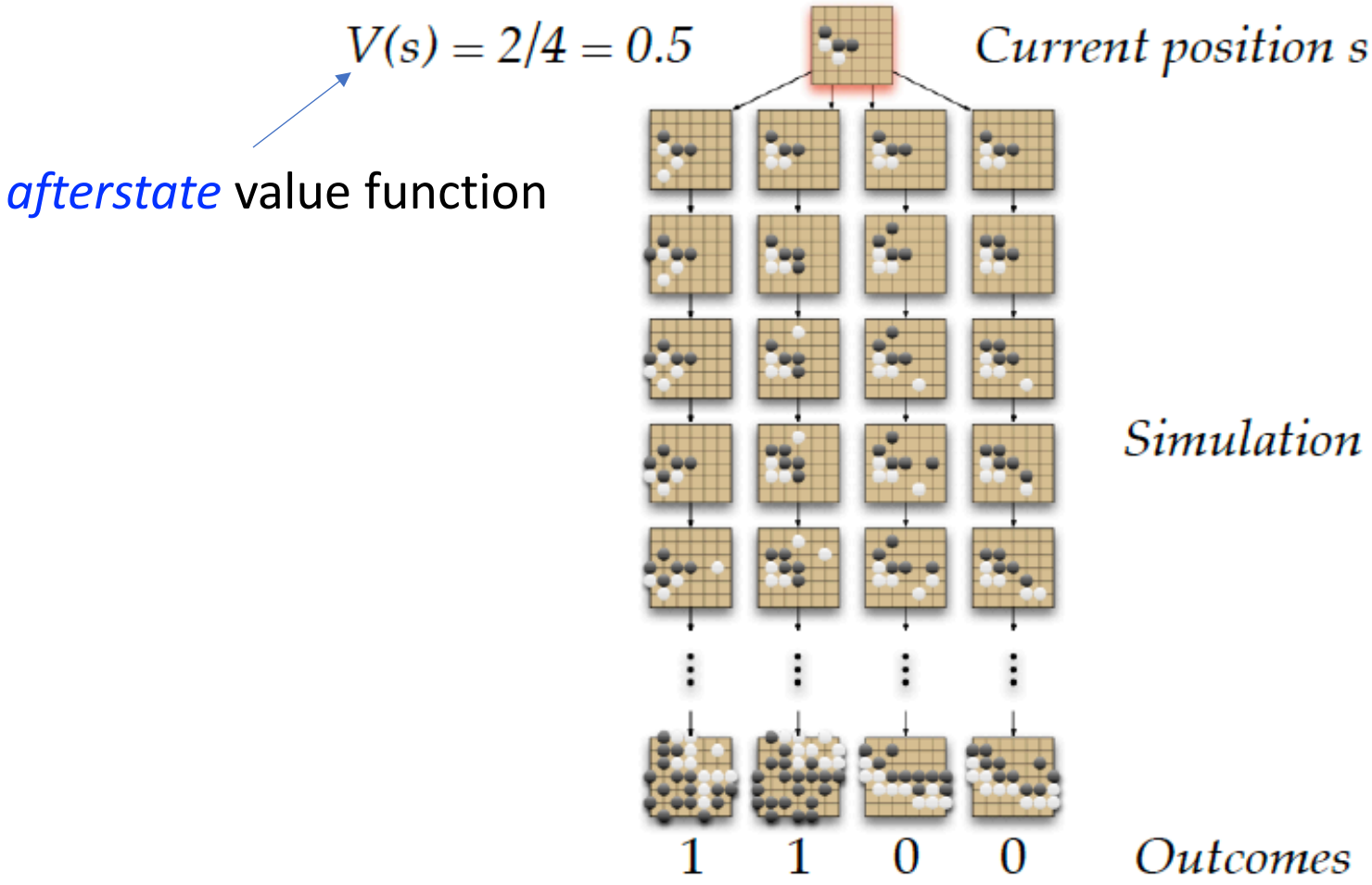
$$R_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

- Policy $\pi = \langle \pi_B, \pi_W \rangle$ selects moves for both players (self play)
- Value function (how good is position s):

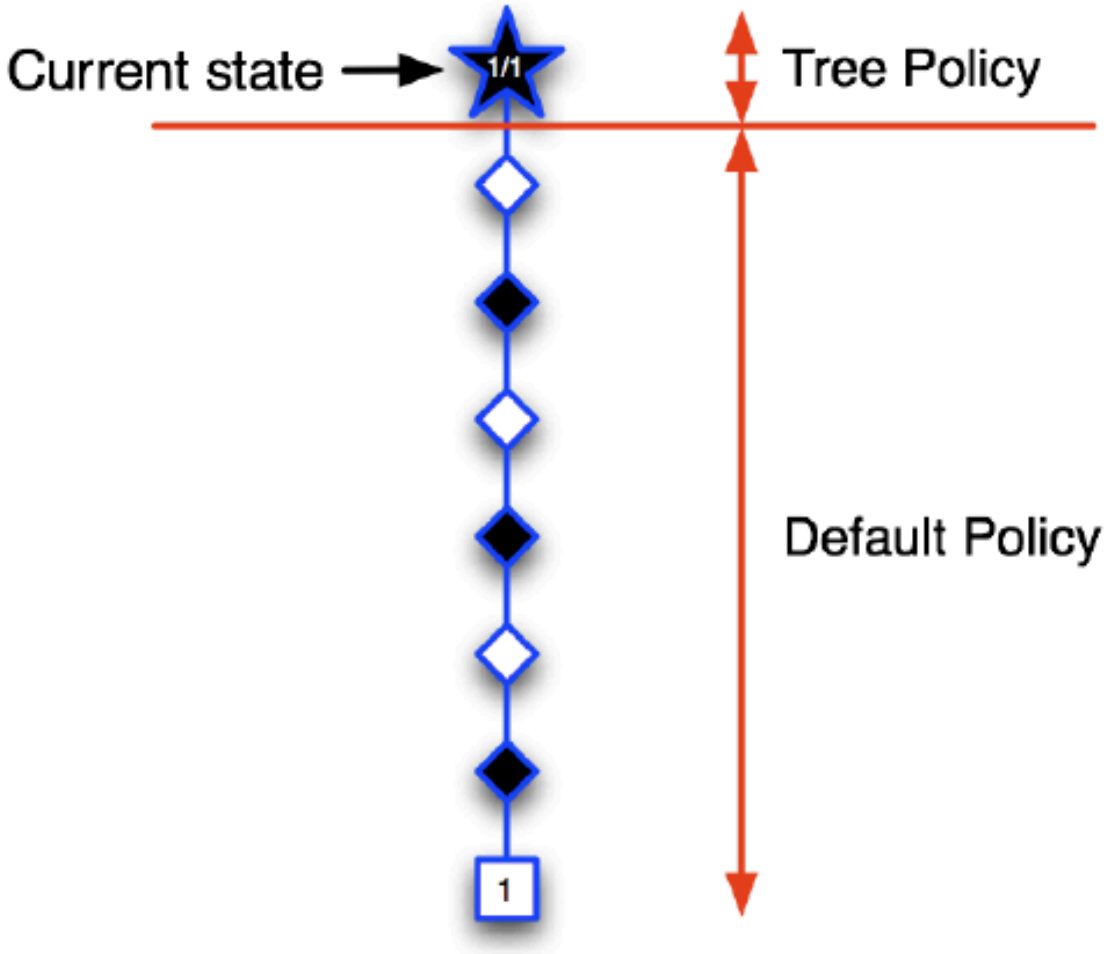
$$v_\pi(s) = \mathbb{E}_\pi \left[R_T \mid S = s \right] = \mathbb{P} \left[\text{Black wins} \mid S = s \right]$$

$$v_*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

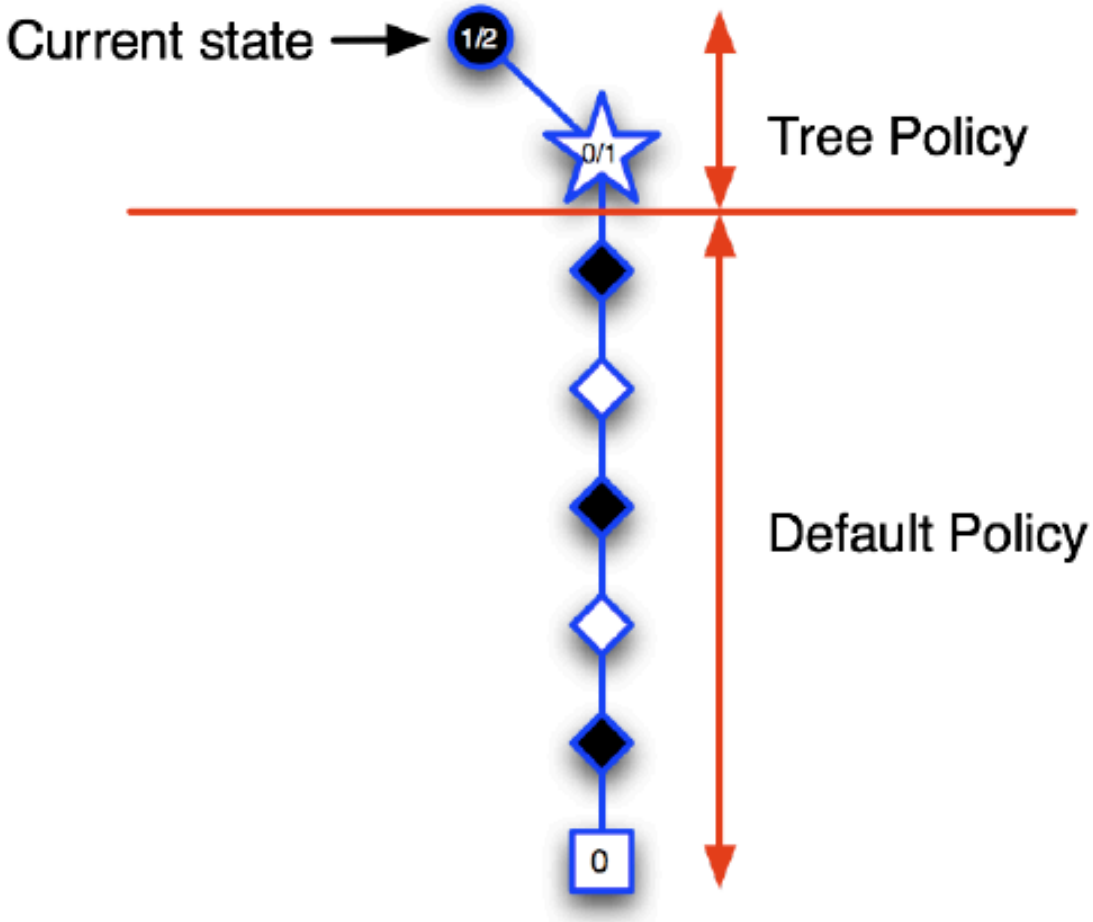
Monte-Carlo Evaluation in Go



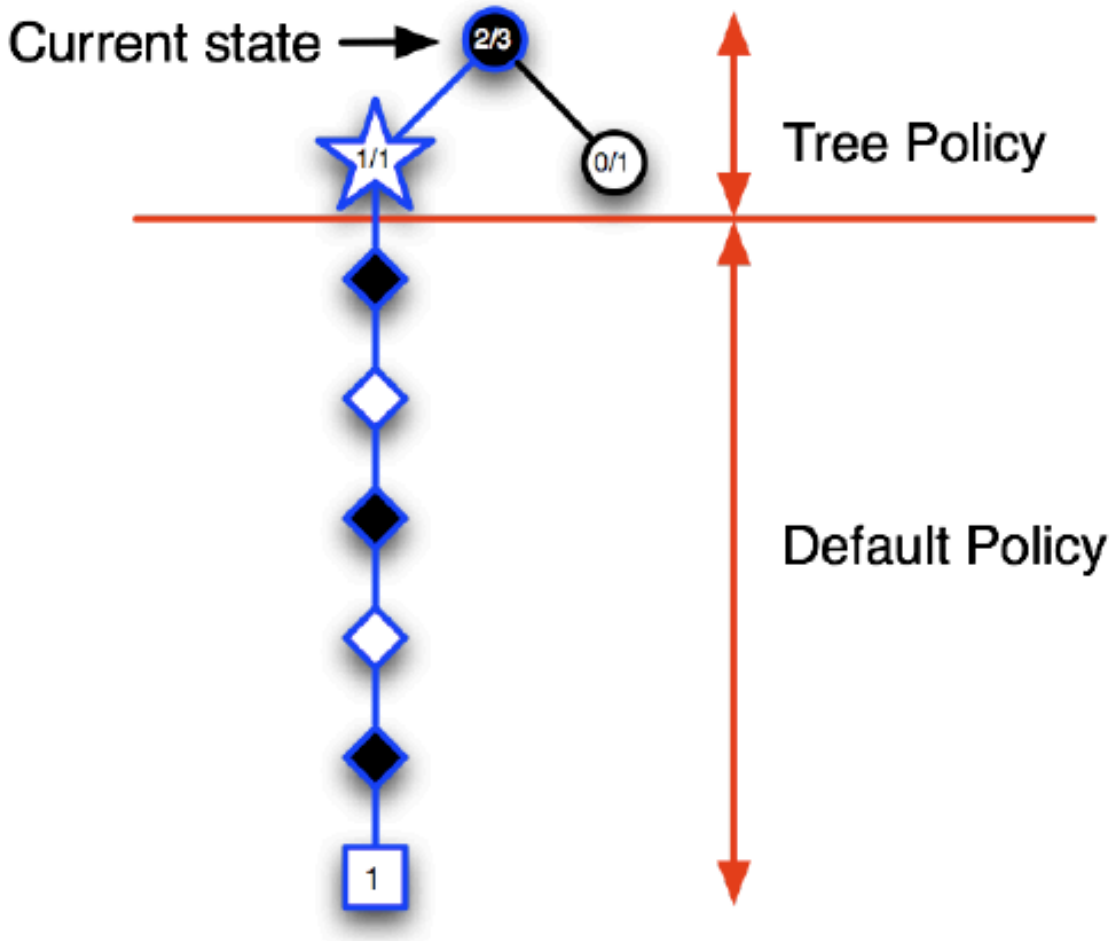
Applying Monte-Carlo Tree Search (1)



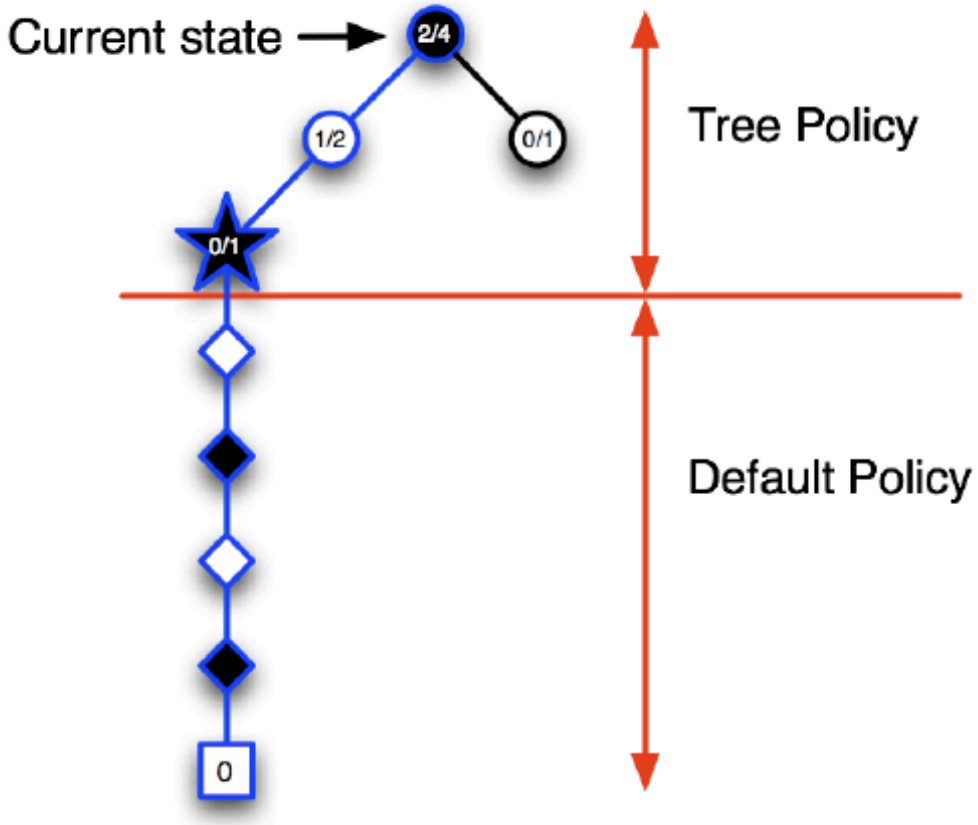
Applying Monte-Carlo Tree Search (2)



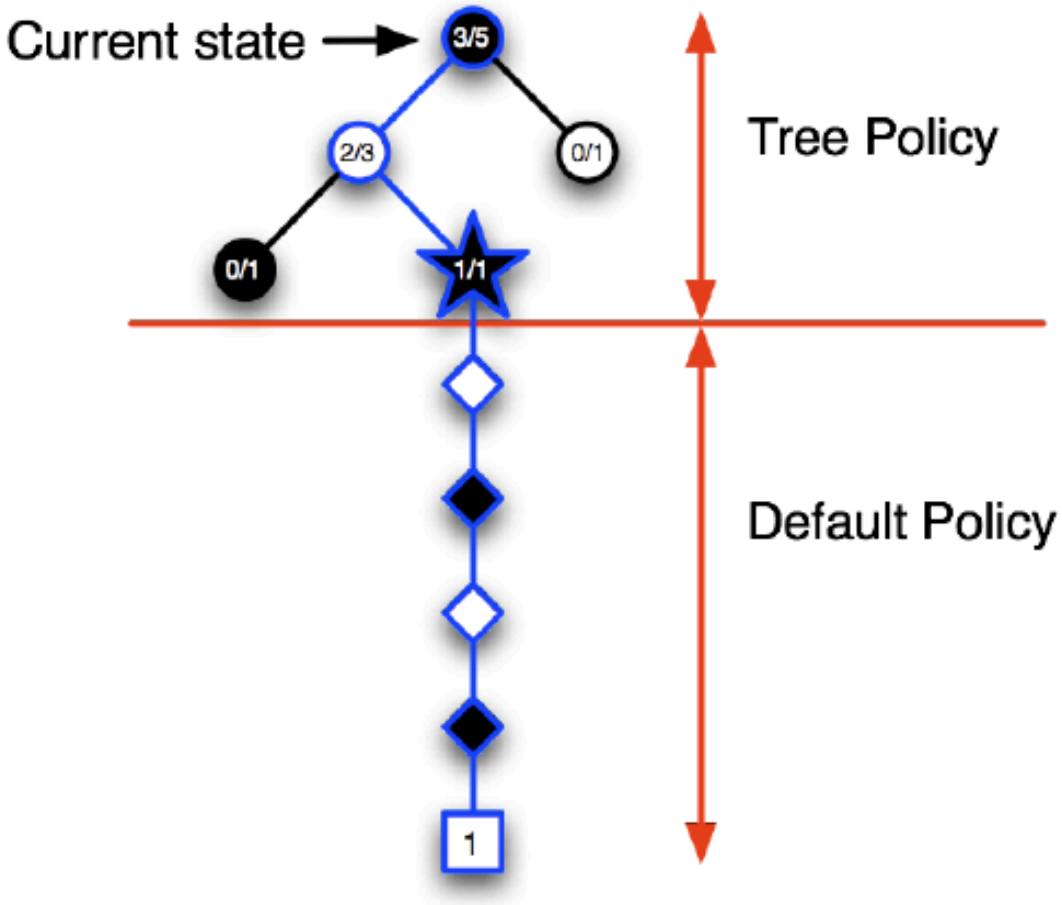
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



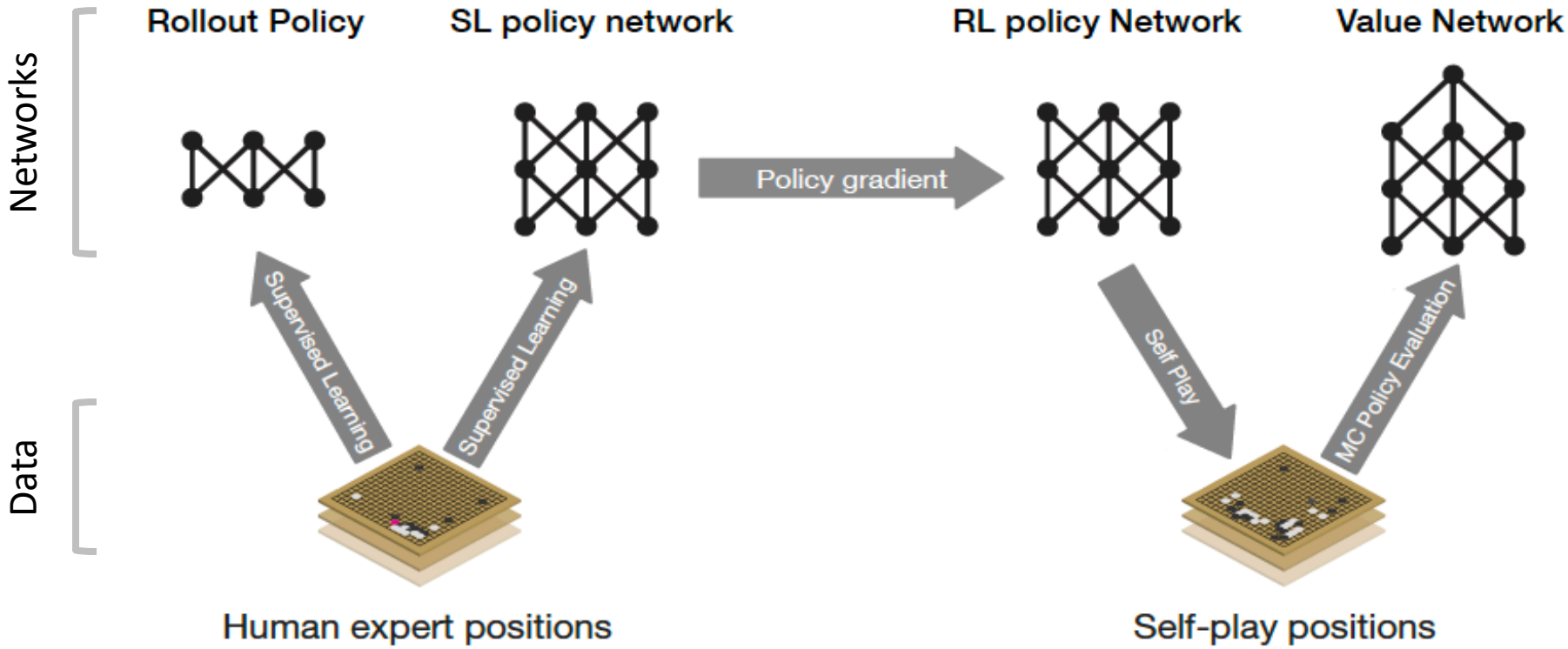
Applying Monte-Carlo Tree Search (5)



Asynchronous Policy and Value MCTS in *AlphaGo*

- Tree expansion
  - chooses an edge according to probabilities supplied a SL-policy network
  - a 13-layer deep convolutional network trained previously by supervised learning to predict moves contained in a database of nearly 30 million human expert moves
- New state evaluation
  - If s is a newly added state, $v(s) = (1-\eta)v_\theta(s) + \eta G$
  - v_θ is returned by a value network, another 13-layer deep convolutional network
  - G is the return of a fast rollout policy provided by a simple linear network, also trained using supervised learning
- The most-visited edge from the root node was selected as the action to take

AlphaGo Pipeline



- All networks were trained before any live game play took place, and their weights remained fixed throughout live play

Advantages of MC Tree Search

- Highly selective best-first search
- Evaluates states dynamically (unlike e.g. DP)
- Uses sampling to break curse of dimensionality
- Works for “black-box" models (only requires samples)
- Computationally efficient, anytime, parallelizable

Exploration and Exploitation

CMPS 4660/6660: Reinforcement Learning

Acknowledgement: slides adapted from David Silver's RL course

Exploration vs. Exploitation Dilemma

- Online decision-making involves a fundamental choice:
  - Exploitation Make the best decision given current information
  - Exploration Gather more information
- The best long-term strategy may involve short-term sacrifices
- Gather enough information to make the best overall decisions

Examples

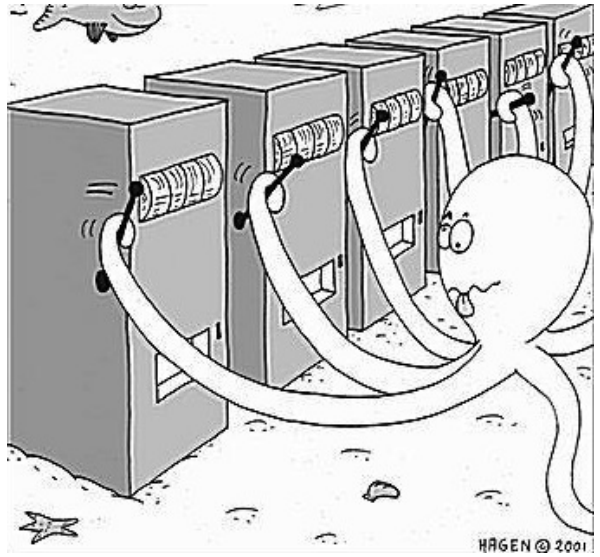
- Restaurant Selection
  - Exploitation Go to your favorite restaurant
  - Exploration Try a new restaurant
- Online Banner Advertisements
  - Exploitation Show the most successful advert
  - Exploration Show a different advert
- Oil Drilling
  - Exploitation Drill at the best-known location
  - Exploration Drill at a new location
- Game Playing
  - Exploitation Play the move you believe is best
  - Exploration Play an experimental move

Principles

- Naive Exploration
  - Add noise to greedy policy (e.g. ϵ-greedy)
- Optimistic Initialization
  - Assume the best until proven otherwise
- Optimism in the Face of Uncertainty
  - Prefer actions with uncertain values
- Probability Matching
  - Select actions according to probability they are best
- Information State Search
  - Lookahead search incorporating value of information

The Multi-Armed Bandit

- A (stochastic) multi-armed bandit is a tuple $\langle \mathcal{A}, \mathcal{D} \rangle$
- $\mathcal{A}$ is a known set of K actions (or "arms")
- $\mathcal{D}_a(r) = \Pr(r|a)$ is an unknown probability distribution over rewards
  - Assumption: $\mathcal{D}_a$ is supported on [0,1] for all $a \in \mathcal{A}$
- At each time step t, the agent selects an action $a_t \in \mathcal{A}$
- The environment generates a reward $r_t \sim \mathcal{D}_{a_t}$
  - Assumption: r_t is an *i.i.d.* sample of $\mathcal{D}_{a_t}$



- The goal is to maximize (expected) cumulative reward $\mathbb{E}[\sum_{t=1}^{T} r_t]$
  - where T is a given time horizon

Regret

- The *action-value* is the mean reward for action a,

$$\mu(a) = \mathbb{E}_{r \sim \mathcal{D}_a}[r|a]$$

- The *optimal-value* μ_* is

$$\mu^* = \mu(a^*) = \max_{a \in \mathcal{A}} \mu(a)$$

- The *regret* is the opportunity loss of one step

$$\mu^* - \mu(a_t)$$

- The *total regret* (over a time horizon T) is the total opportunity loss

$$R(T) = \sum_{t=1}^{T} [\mu^* - \mu(a_t)]$$

- Maximize cumulative reward ≡ minimize total expected regret $\mathbb{E}[R(T)]$

Counting Regret

- The count $N_t(a)$ is expected number of selections for action a up to t
- The gap Δ_a is the difference in value between action a and optimal action a^*, $\Delta_a = \mu^* - \mu(a)$
- Regret is a function of gaps and the counts

$$\begin{aligned}\mathbb{E}[R(T)] &= \mathbb{E}[\textstyle\sum_{t=1}^{T}(\mu^* - \mu(a_t))] \\ &= \textstyle\sum_{a\in\mathcal{A}} \mathbb{E}[N_T(a)]\left(\mu^* - \mu(a)\right) \\ &= \textstyle\sum_{a\in\mathcal{A}} \mathbb{E}[N_T(a)]\,\Delta_a\end{aligned}$$

- A good algorithm ensures small counts for large gaps
- Sublinear regret: $\mathbb{E}[R(T)] = o(T) \Rightarrow \lim_{T\to\infty} \dfrac{\mathbb{E}[R(T)]}{T} = 0$

Explore-First

1. Exploration phase: try each arm N times
2. Estimate the value of each action by Monte-Carlo evaluation

$$\bar{\mu}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau \mathbf{1}(a_\tau = a)$$

3. Select the arm $\hat{a}$ with the highest average reward (break ties arbitrarily)
4. Exploitation phase: play arm $\hat{a}$ in all remaining rounds

By taking $N = (T/K)^{2/3} \cdot O(\log T)^{1/3}$, $\mathbb{E}[R(T)] \leq T^{2/3} \times O(K \log T)^{1/3}$

- Poor performance in the exploration stage
- Can lead to linear regret when N is not properly chosen (either too small or too big)

ϵ-Greedy

for each round $t = 1,2, \ldots$ do

Toss a coin with success probability ϵ_t

if success then

explore: choose an arm uniformly at random

else

exploit: choose the arm with the highest average reward so far

end

By taking $\epsilon_t = t^{-1/3}(K\log t)^{1/3}$, $\mathbb{E}[R(t)] \leq t^{2/3} \times O(K \log t)^{1/3}$ for each round t

- Decaying ϵ_t is necessary to obtain sublinear regret

Adaptive Algorithms

- A big flaw of the previous two algorithms: exploration schedule does not depend on the observed rewards
- Adaptive algorithms
  - Successive Elimination
  - Optimism under uncertainty: UCB
  - Probability matching: Thompson sampling
  - ..

$$\mathbb{E}[R(T)] \leq O\left(\sqrt{KT\log T}\right)$$

- Can we do better?
  - Lower bound: fix T and K, there is a problem instance such that $\mathbb{E}[R(T)] \geq \Omega(\sqrt{KT})$

Upper Confidence Bounds

- Estimate an upper confidence $U_t(a)$ for each action value
- Such that $\mu(a) \leq \bar{\mu}_t(a) + U_t(a)$ with high probability
- This depends on the number of times $N_t(a)$ has been selected
  - Small $N_t(a) \Rightarrow$ large $U_t(a)$ (estimated value is uncertain)
  - Large $N_t(a) \Rightarrow$ small $U_t(a)$ (estimated value is accurate)
- Select action maximizing Upper Confidence Bound (UCB)

$$a_t = \text{argmax}_{a \in \mathcal{A}} \bar{\mu}_t(a) + U_t(a)$$

Hoeffding's Inequality

Theorem (Hoeffding's Inequality)

Let $X_1, \dots, X_t$ be $i.i.d.$ random variables in [0,1], and let $\bar{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau$. Then

$$\Pr[|\bar{X}_t - \mathbb{E}[X]| \geq u] \leq 2e^{-2tu^2}$$

- We will apply Hoeffding's Inequality to rewards of the bandit conditioned on selecting an action a

$$\Pr[\mu(a) \geq \bar{\mu}_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

Calculating Upper Confidence Bounds

- Pick a probability p that true value exceeds UCB
- Now solve for U_t

$$e^{-2N_t(a)U_t(a)^2} = p$$

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

- Reduce p as we observe more rewards, e.g., $p = t^{-4}$
- Ensures we select optimal action as $t \to \infty$

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}}$$

UCB1

- This leads to the UCB1 algorithm

$$a_t = \text{argmax}_{a \in \mathcal{A}} \bar{\mu}_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}}$$

UCB1 obtains regret $\mathbb{E}[R(t)] \leq O(\sqrt{Kt\log T})$ for $t \leq T$

Q-learning with UCB

- Consider tabular Q-learning in the episodic case
  - S: number of states, A: number of actions
  - H: length of each episode, K: number of episodes
  - $T = KH$: total number of steps
- Q-learning with ϵ-greedy leads to a regret $\Omega(\min\{T, A^{H/2}\})$
- Q-learning with UCB achieves a regret of $\tilde{O}(\sqrt{H^3SAT})$
  - Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I. Jordan, "Is Q-learning Provably Efficient?", NeurIPS 2018

Q-learning with UCB

Algorithm 1 Q-learning with UCB-Hoeffding

initialize $Q_h(x,a) \leftarrow H$ and $N_h(x,a) \leftarrow 0$ for all $(x,a,h) \in \mathcal{S} \times \mathcal{A} \times [H]$.
for episode $k = 1, \ldots, K$ **do**
  receive x_1.
  for step $h = 1, \ldots, H$ **do**
    Take action $a_h \leftarrow \mathrm{argmax}_{a'} Q_h(x_h, a')$, and observe x_{h+1}.
    $t = N_h(x_h, a_h) \leftarrow N_h(x_h, a_h) + 1$; $b_t \leftarrow c\sqrt{H^3\iota/t}$.
    $Q_h(x_h, a_h) \leftarrow (1-\alpha_t) Q_h(x_h, a_h) + \alpha_t [r_h(x_h, a_h) + V_{h+1}(x_{h+1}) + b_t]$.
    $V_h(x_h) \leftarrow \min\{H, \max_{a' \in \mathcal{A}} Q_h(x_h, a')\}$.

Learning rate $\alpha_t = \frac{H+1}{H+t}$, $\iota = \log(SAT/p)$