Lecture 8: Integrating Learning and Planning

David Silver

Outline

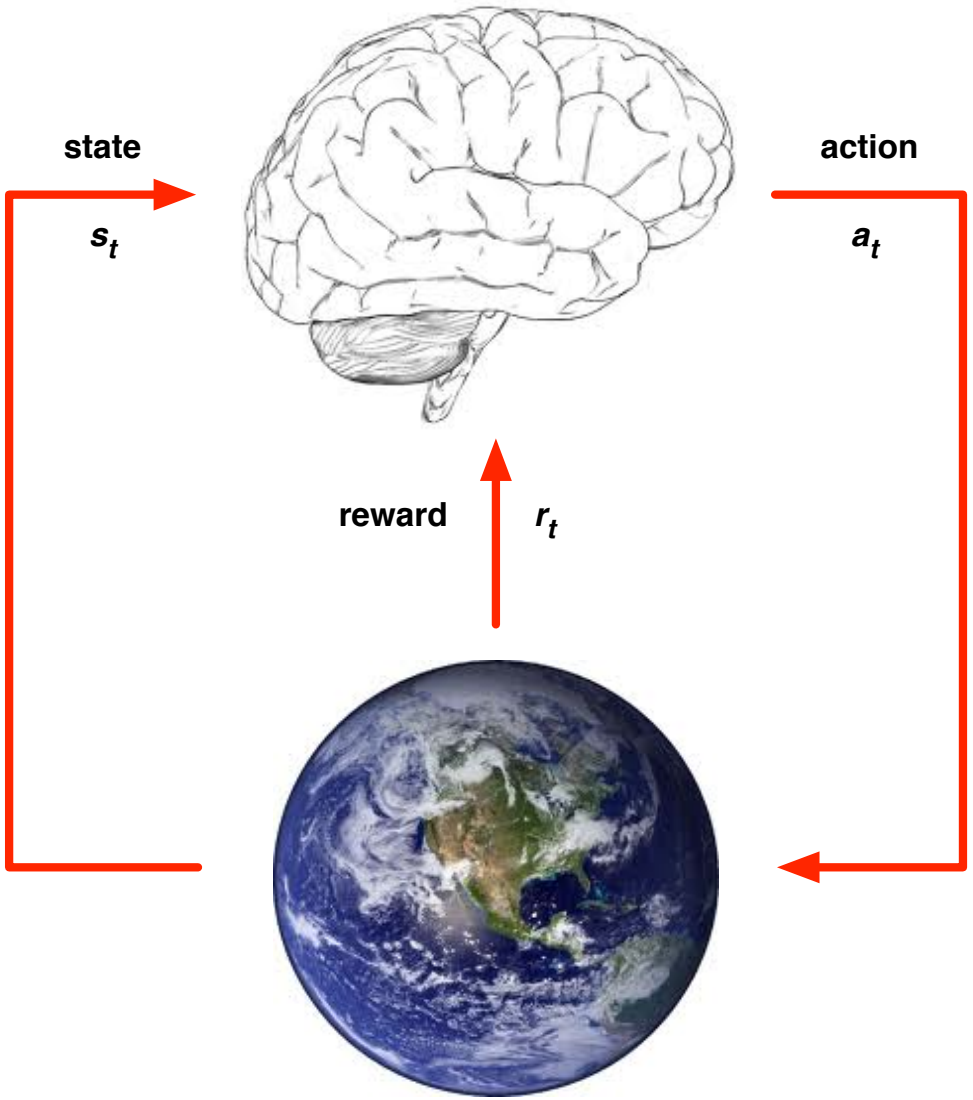
Model-Based Reinforcement Learning

- *Last lecture*: learn policy directly from experience
- *Previous lectures*: learn value function directly from experience
- *This lecture*: learn model directly from experience
- and use planning to construct a value function or model
- Integrate learning and planning into a single architecture

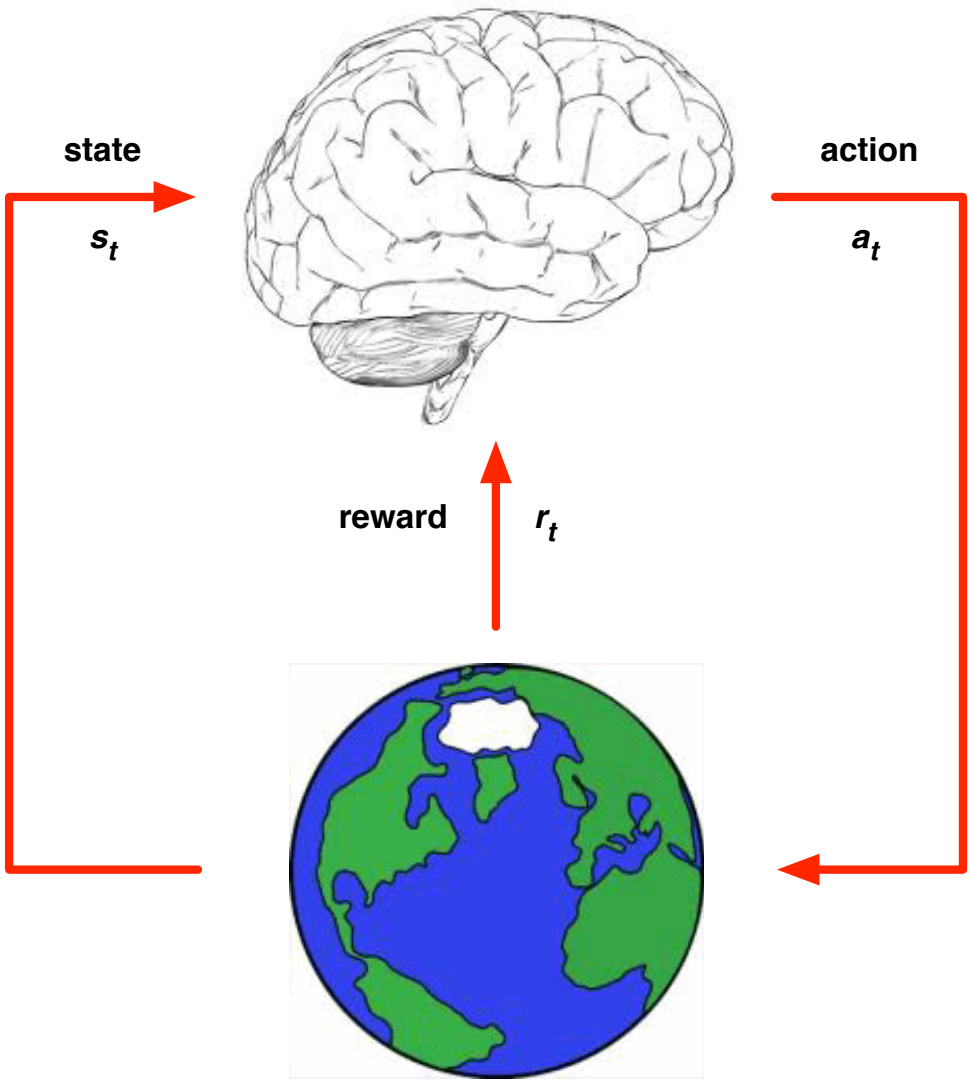
Model-Based and Model-Free RL

- 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

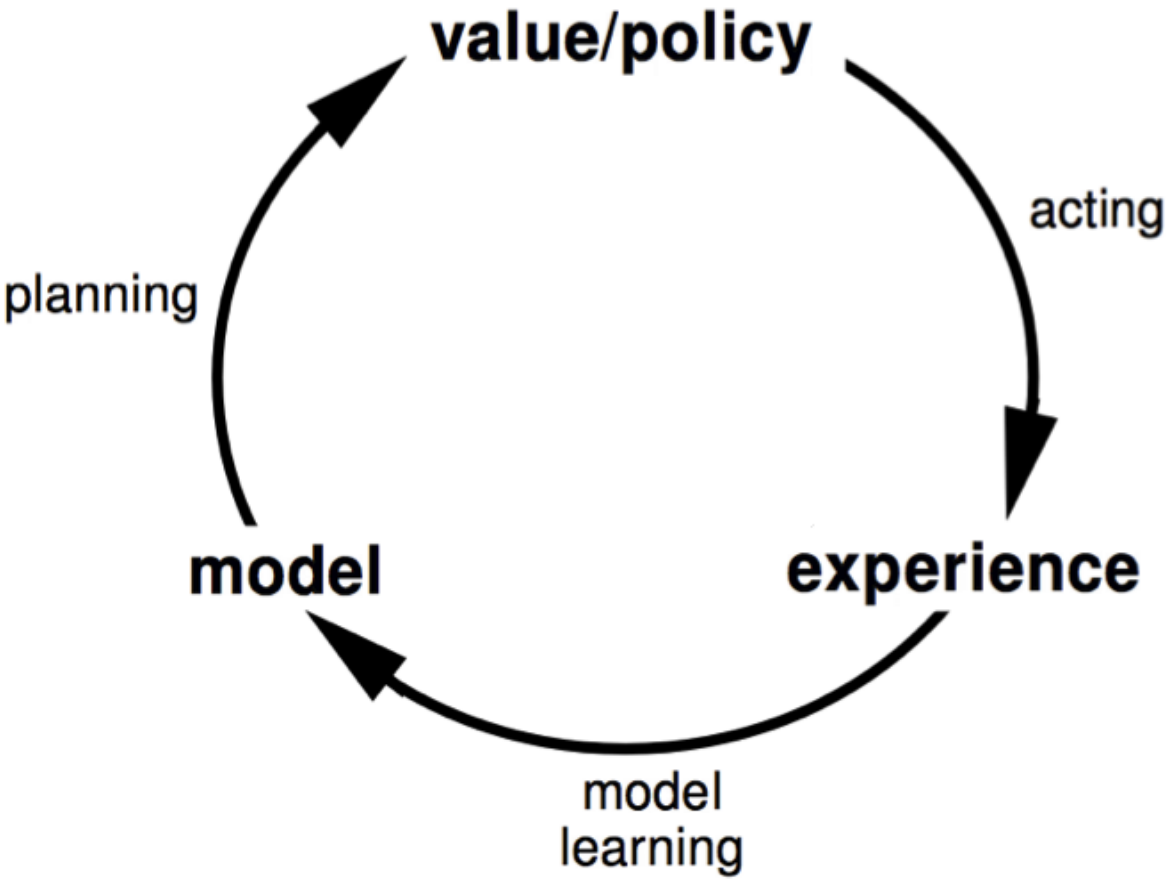
Model-Free RL



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
  $\Rightarrow$ two sources of approximation error

What is a Model?

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

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Can assume conditional independence between state transitions and rewards

$$\mathbb{P}\left[S_{t+1}, R_{t+1} \mid S_t, A_t\right] = \mathbb{P}\left[S_{t+1} \mid S_t, A_t\right] \mathbb{P}\left[R_{t+1} \mid S_t, A_t\right]$$

Model Learning

- Goal: estimate model $\mathcal{M}_\eta$ from experience $\{s_1, a_1, r_2, ..., s_T\}$
- This is a supervised learning problem

$$
\begin{aligned}
s_1, a_1 &\rightarrow r_2, s_2 \\
s_2, a_2 &\rightarrow r_3, s_3 \\
&\vdots \\
s_{T-1}, a_{T-1} &\rightarrow r_T, s_T
\end{aligned}
$$

- 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 minimise 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, $\hat{\mathcal{P}}, \hat{\mathcal{R}}$
- Count visits $N(s, a)$ to each state action pair

$$\hat{\mathcal{P}}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(s_t, a_t, s_{t+1} = s, a, s')$$

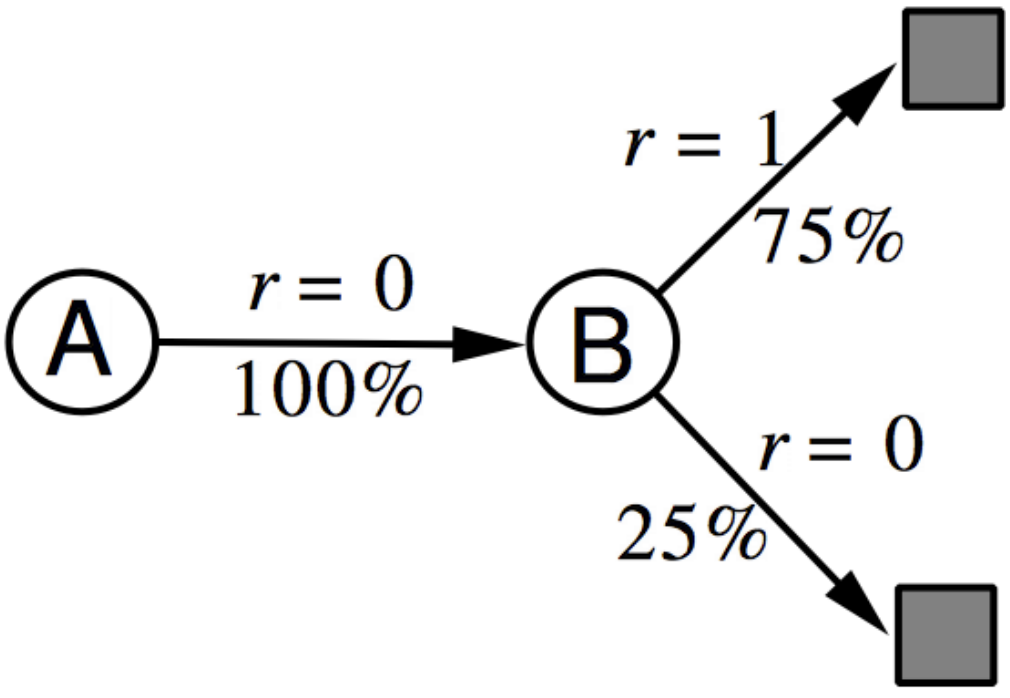
$$\hat{\mathcal{R}}^a_s = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(s_t, a_t = s, a) r_t$$

- Alternatively
  - At each time-step t, record experience tuple $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$
  - 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

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$



We have constructed a table lookup model from the experience

Planning with a Model

- Given a model $\mathcal{M}_\eta = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Solve the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Using favourite 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

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

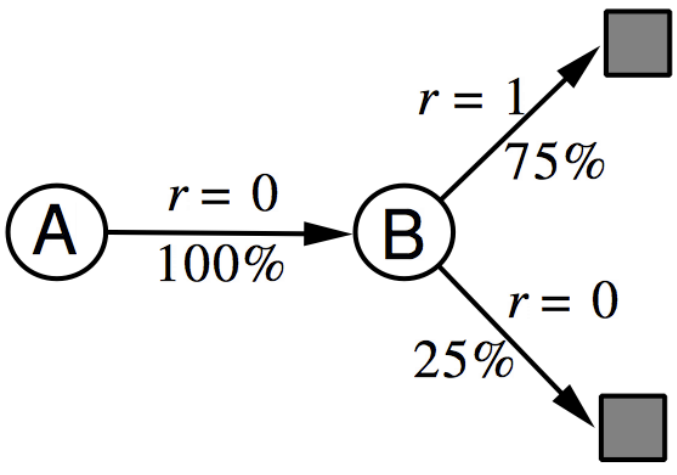
- Apply model-free RL to samples, e.g.:
  - Monte-Carlo control
  - Sarsa
  - Q-learning
- Sample-based planning methods are often more efficient

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, 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 $\langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle \neq \langle \mathcal{P}, \mathcal{R} \rangle$
- Performance of model-based RL is limited to optimal policy for approximate MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- 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

Real and Simulated Experience

We consider two sources of experience

Real experience Sampled from environment (true MDP)

$$s' \sim \mathcal{P}^a_{s,s'}$$
$$r = \mathcal{R}^a_s$$

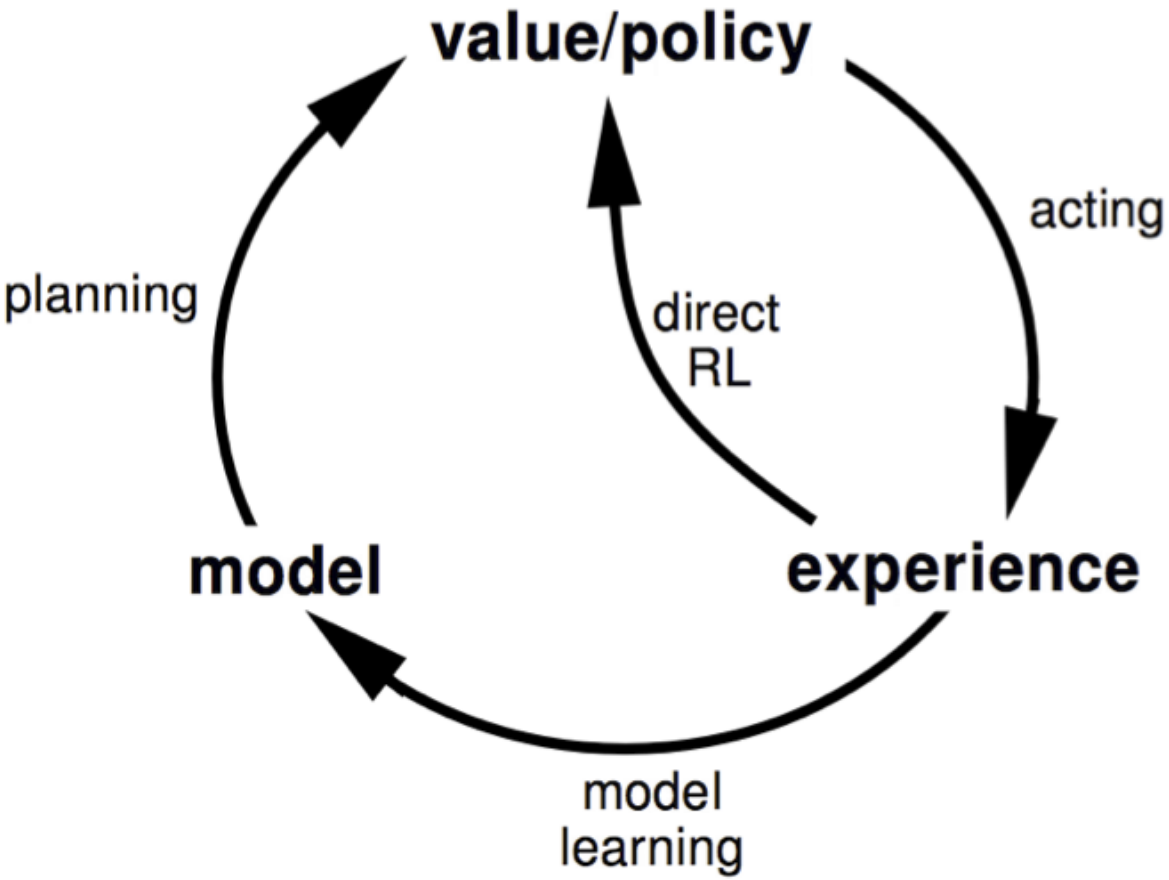
Simulated experience Sampled from model (approximate MDP)

$$s' \sim \mathcal{P}_\eta(s' \mid s, a)$$
$$r = \mathcal{R}_\eta(r \mid s, a)$$

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 real 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

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:

(a) $s \leftarrow$ current (nonterminal) state

(b) $a \leftarrow \varepsilon$-greedy(s,Q)

(c) Execute action a; observe resultant state, s', and reward, r

(d) $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

(e) $Model(s,a) \leftarrow s', r$ (assuming deterministic environment)

(f) Repeat N times:

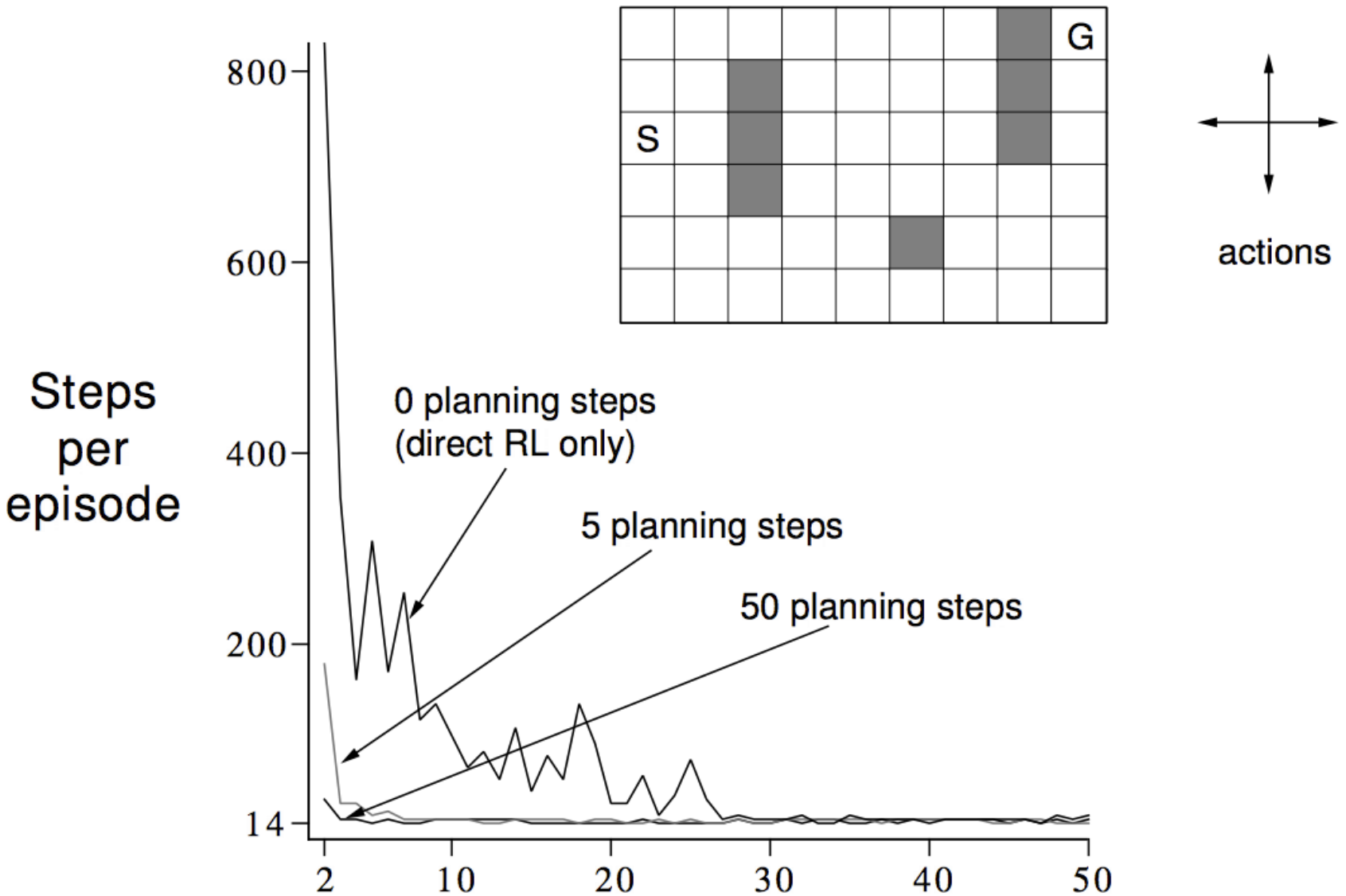
$s \leftarrow$ random previously observed state

$a \leftarrow$ random action previously taken in s

$s', r \leftarrow Model(s,a)$

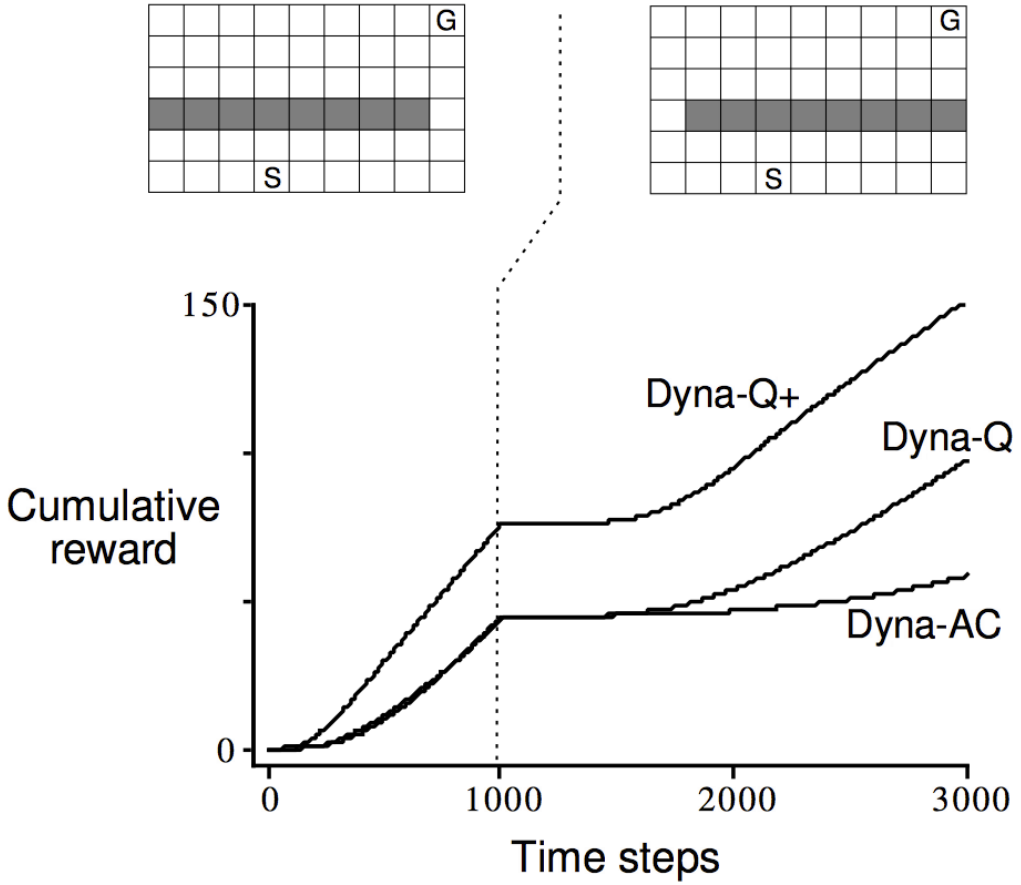
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Dyna-Q on a Simple Maze



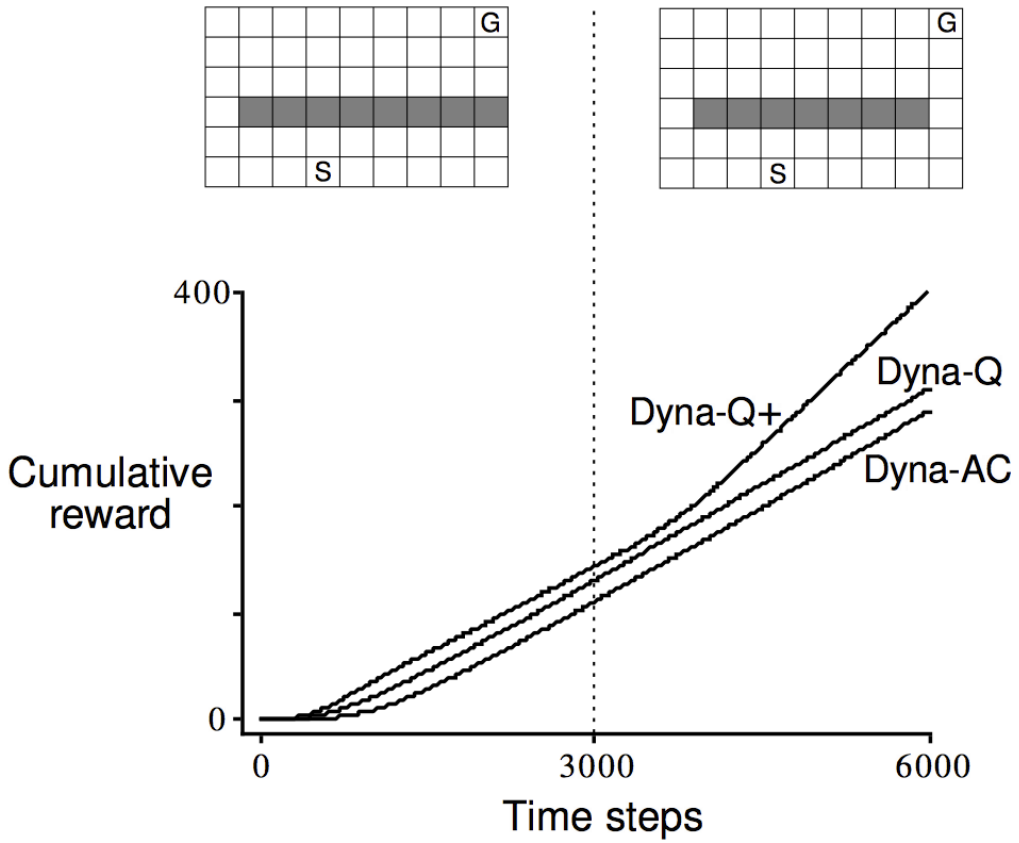
Dyna-Q with an Inaccurate Model

- The changed environment is harder



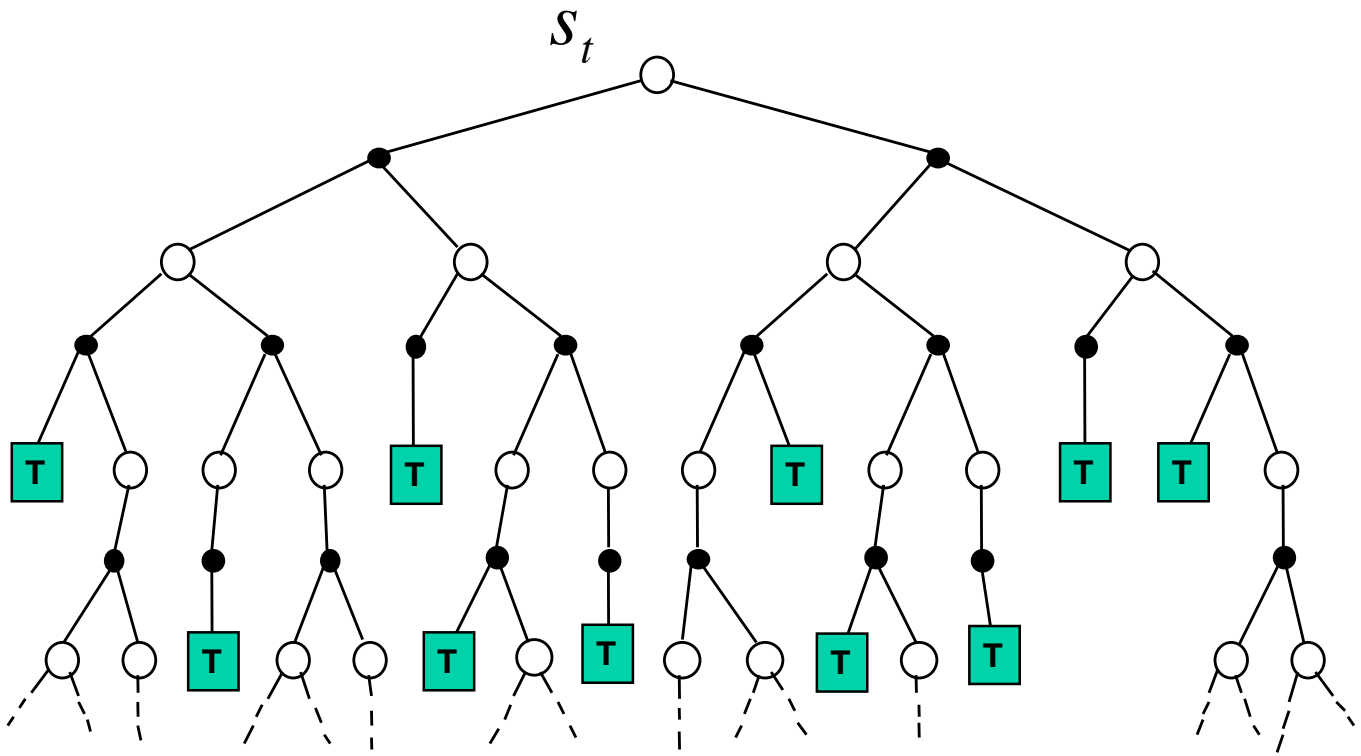
Dyna-Q with an Inaccurate Model (2)

- The changed environment is easier



Forward Search

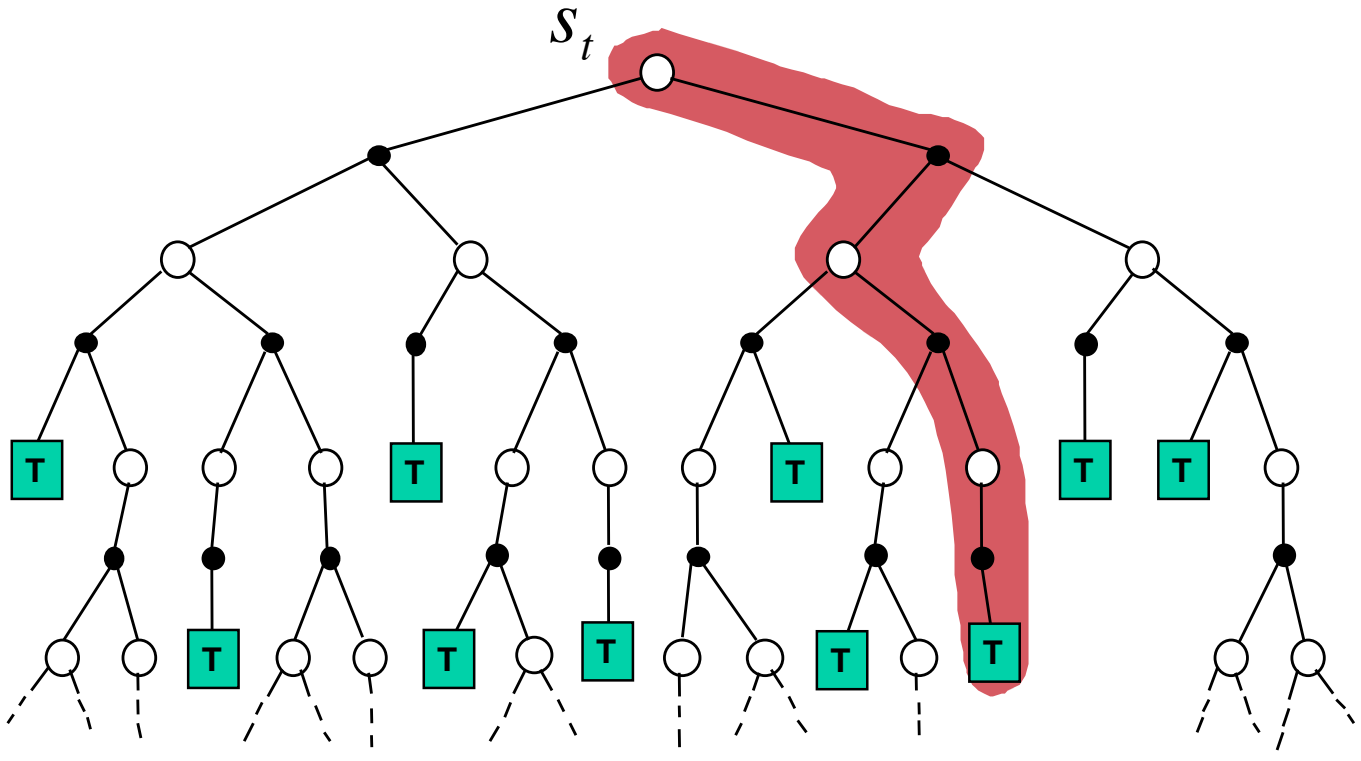
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- 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

$$\{s_t^k, a_t^k, r_{t+1}^k, ..., s_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

- Apply model-free RL to simulated episodes
  - Monte-Carlo control $\rightarrow$ Monte-Carlo search
  - Sarsa $\rightarrow$ TD search

Monte-Carlo Simulation

- Given a model $\mathcal{M}_\nu$ and a simulation policy π
- Simulate K episodes from current state s_t

$$\{s_t, a_t^k, r_{t+1}^k, s_{t+1}^k, ..., s_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

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

$$V(s_t) = \frac{1}{K} \sum_{k=1}^{K} v_t \xrightarrow{P} V^\pi(s_t)$$

Simple Monte-Carlo Search

- Given a model $\mathcal{M}_\nu$ and a policy π
- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s_t

$$\{s_t, a, r^k_{t+1}, s^k_{t+1}, a^k_{t+1}, ..., s^k_T\}^K_{k=1} \sim \mathcal{M}_\nu, \pi$$

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

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

- Select current (real) action with maximum value

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

Monte-Carlo Tree Search (Evaluation)

- Given a model $\mathcal{M}_\nu$
- Simulate K episodes from current state s_t using current simulation policy π

$$\{s_t, a_t^k, r_{t+1}^k, s_{t+1}^k, ..., s_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

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

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(s_u, a_u = s, a) v_u \xrightarrow{P} Q^\pi(s, a)$$

- 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)

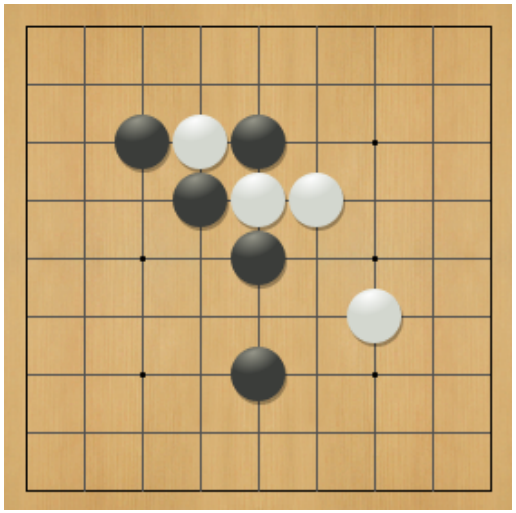
- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximise $Q(s, a)$
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(s, a)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by $\epsilon - \text{greedy}(Q)$
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(s, a) \rightarrow Q^*(s, a)$

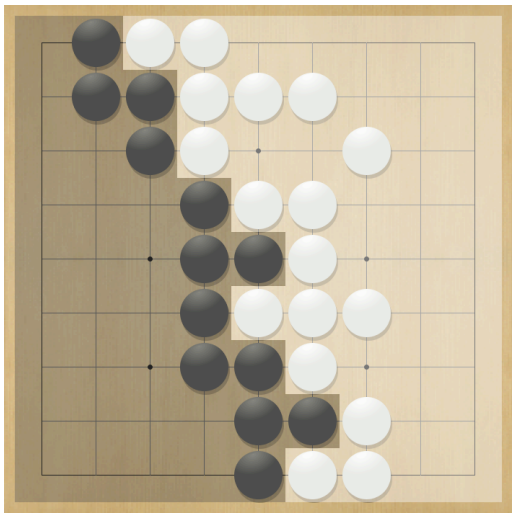
Case Study: the Game of Go

- 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):

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

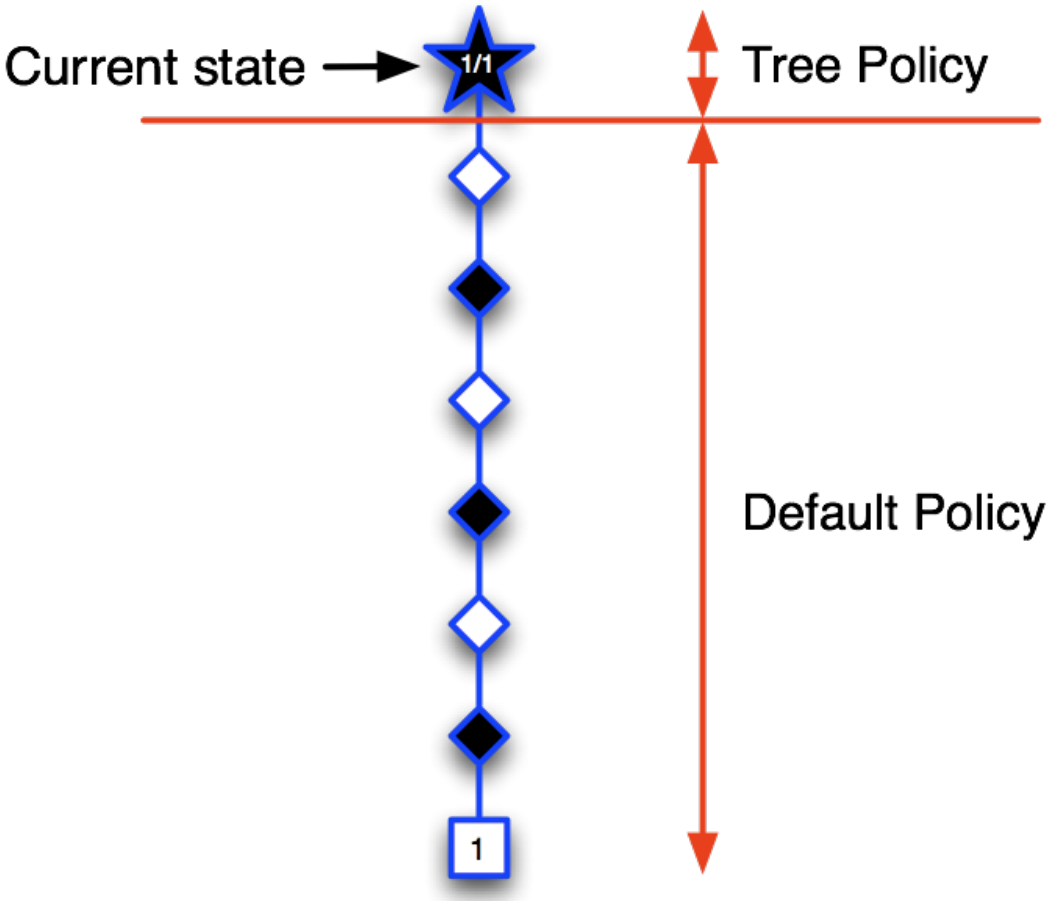
$$r_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

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

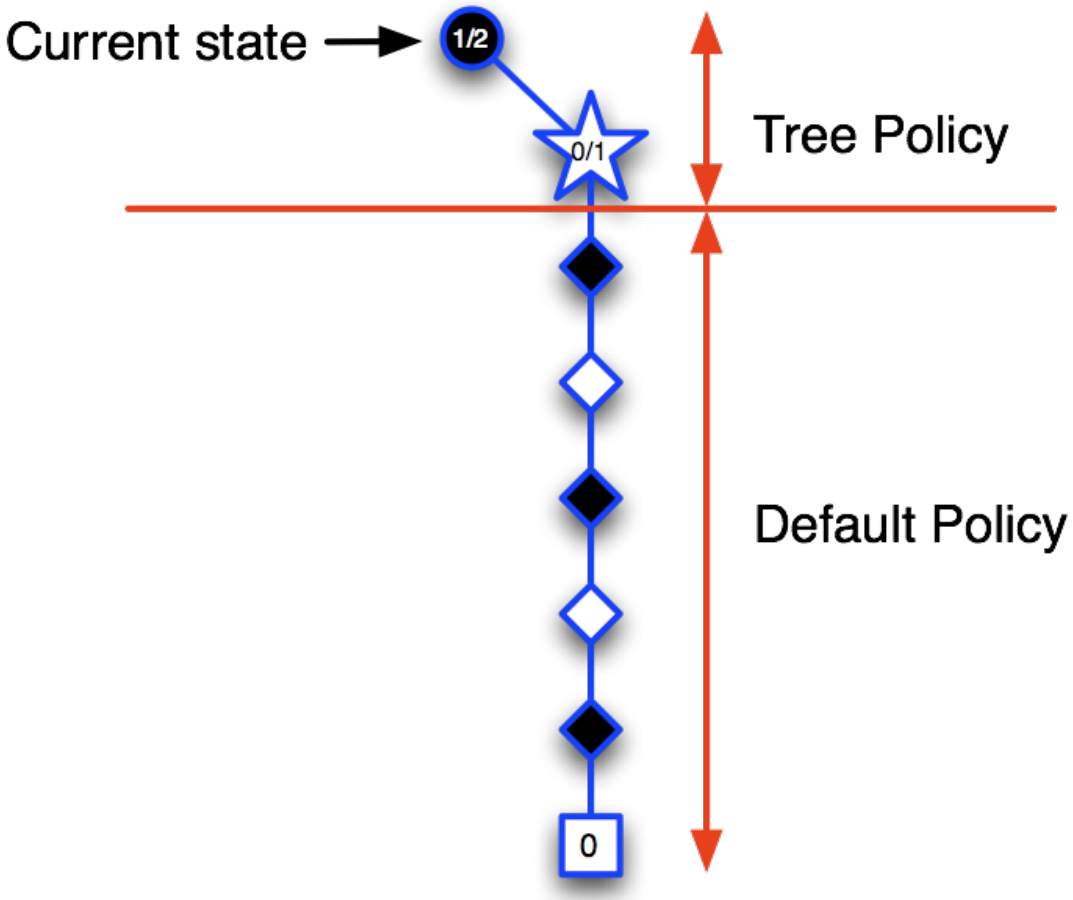
$$V^\pi(s) = \mathbb{E}_\pi \left[r_T \mid s \right] = \mathbb{P} \left[\text{Black wins} \mid s \right]$$

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

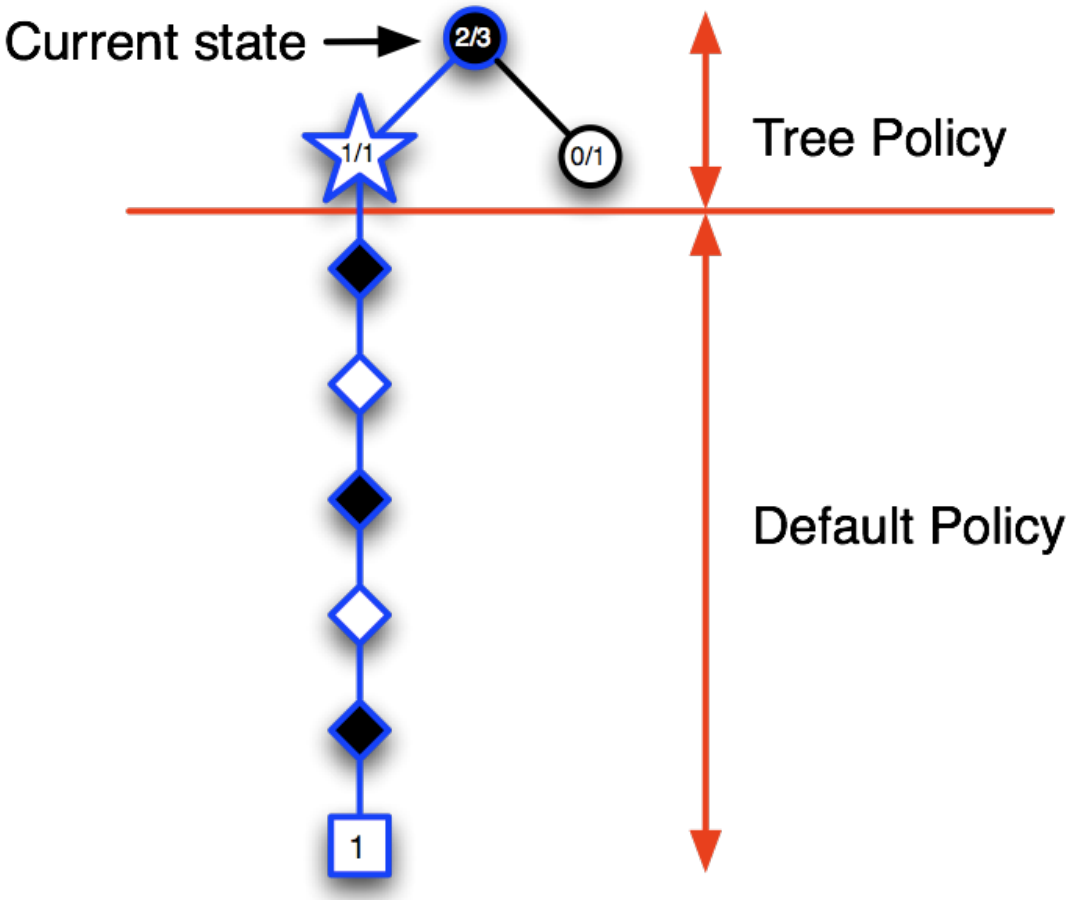
Applying Monte-Carlo Tree Search (1)



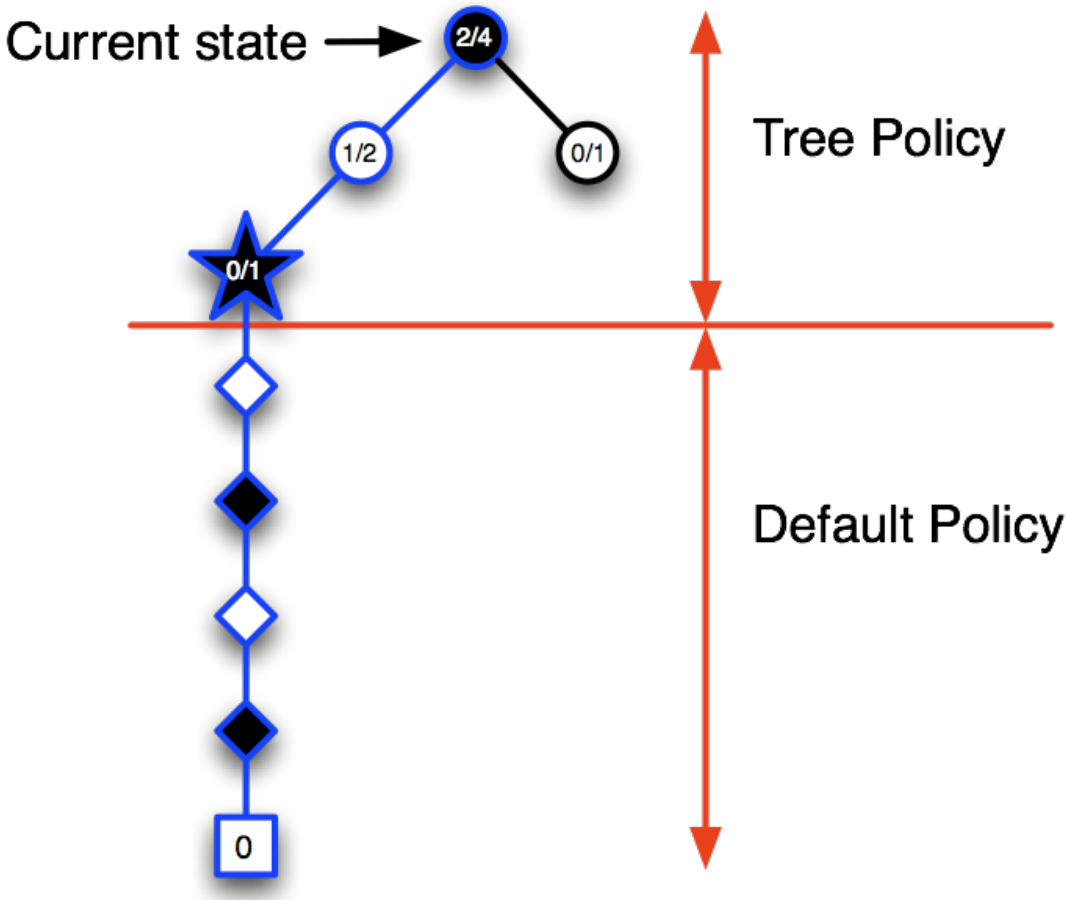
Applying Monte-Carlo Tree Search (2)



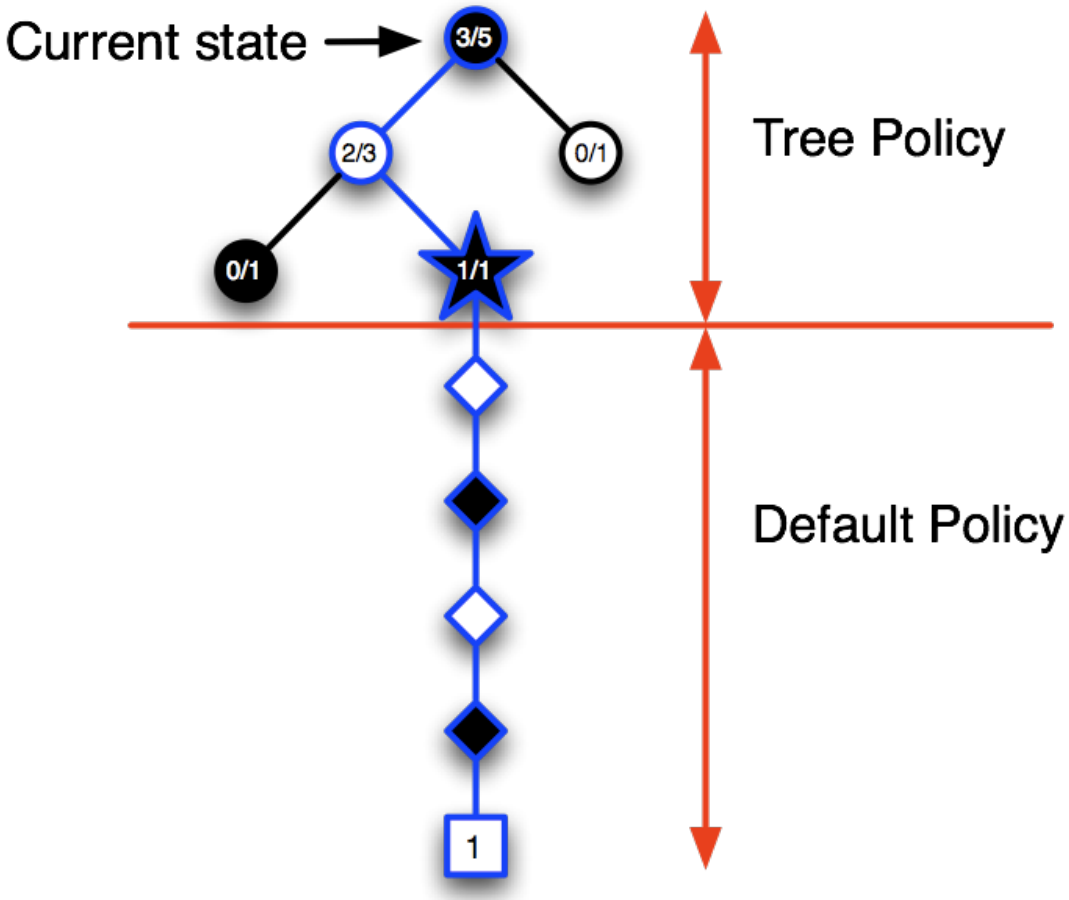
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



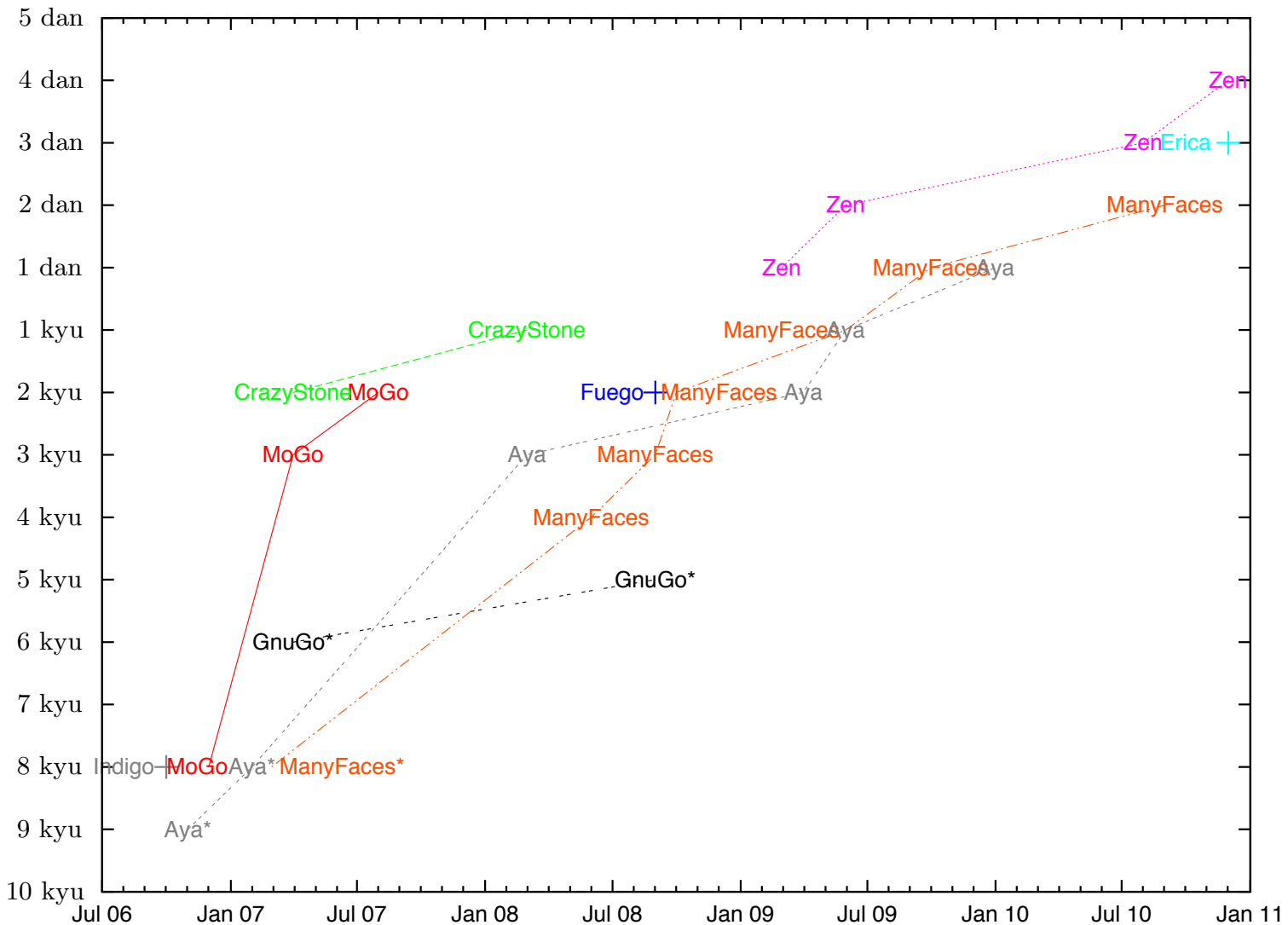
Applying Monte-Carlo Tree Search (5)



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, parallelisable

Example: MC Tree Search in Computer Go



Temporal-Difference Search

- Simulation-based search
- Using TD instead of MC (bootstrapping)
- MC tree search applies MC control to sub-MDP from now
- TD search applies Sarsa to sub-MDP from now

MC vs. TD search

- For model-free reinforcement learning, bootstrapping is helpful
  - TD learning reduces variance but increases bias
  - TD learning is usually more efficient than MC
  - TD(λ) can be much more efficient than MC
- For simulation-based search, bootstrapping is also helpful
  - TD search reduces variance but increases bias
  - TD search is usually more efficient than MC search
  - TD(λ) search can be much more efficient than MC search

TD Search

- Simulate episodes from the current (real) state s_t
- Estimate action-value function $Q(s, a)$
- For each step of simulation, update action-values by Sarsa

$$\Delta Q(s, a) = \alpha(r + \gamma Q(s', a') - Q(s, a))$$

- Select actions based on action-values $Q_\theta(s, a)$
  - e.g. ϵ-greedy

Search tree vs. value function approximation

- Search tree is a table lookup approach
- Based on a *partial* instantiation of the table
- For model-free reinforcement learning, table lookup is naive
  - Can't store value for all states
  - Doesn't generalise between similar states
- For simulation-based search, table lookup is less naive
  - Search tree stores value for easily reachable states
  - But still doesn't generalise between similar states
  - In huge search spaces, value function approximation is helpful

Linear TD Search

- Use linear value function approximation for action-value function $Q_\theta(s, a) \approx Q^\pi(s, a)$

$$Q_\theta(s, a) = \phi(s, a)^T \theta$$

- Simulate episodes from the current (real) state s_t
- At each simulated step u, update action-values by linear Sarsa

$$\Delta\theta = \alpha(r_{u+1} + \gamma Q_\theta(s_{u+1}, a_{u+1}) - Q_\theta(s_u, a_u))\phi(s_u, a_u)$$

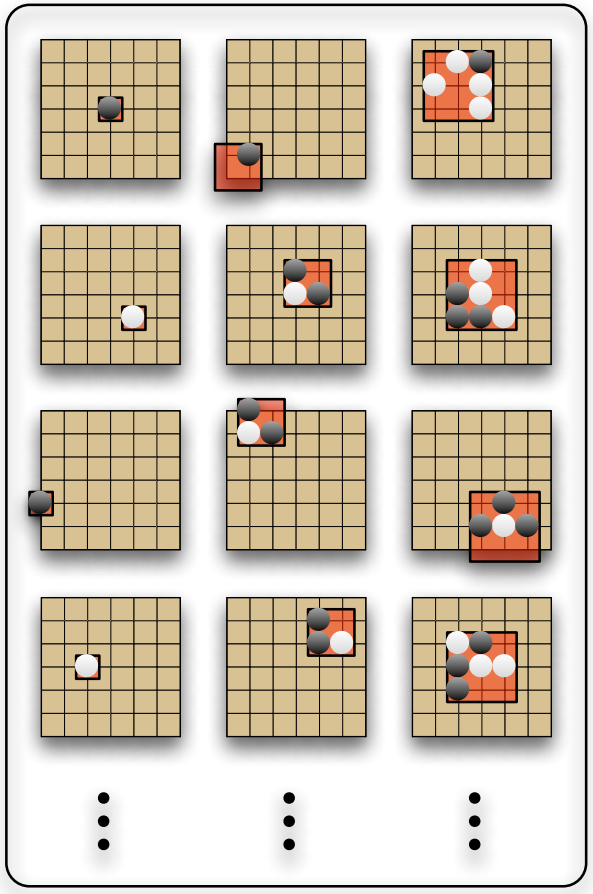
- Select actions based on action-values $Q_\theta(s, a)$
  - e.g. ϵ-greedy

Dyna-2

- In Dyna-2, the agent stores two sets of feature weights
  - Long-term memory
  - Short-term (working) memory
- Long-term memory is updated from real experience using TD learning
  - General domain knowledge that applies to any episode
- Short-term memory is updated from simulated experience using TD search
  - Specific local knowledge about the current situation
- Over value function is sum of long and short-term memories

Local Shape Features in Go

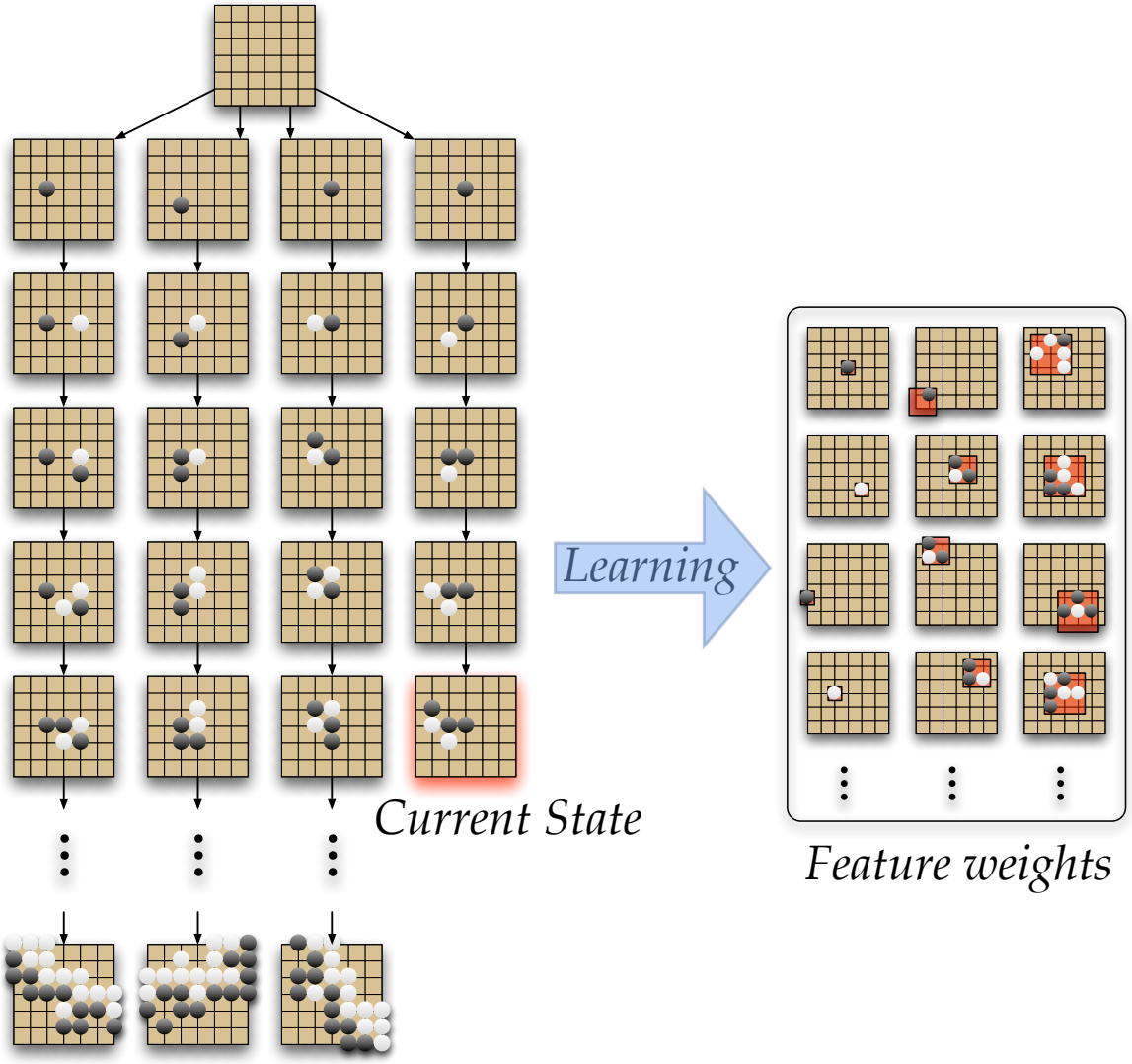
- Binary features matching a local configuration of stones
- All possible locations and configurations from 1x1 to 3x3
- A million features for 9×9 Go



Linear TD Learning in Go

- Features $\phi(s)$
- Value function $V(s) = \phi(s)^\top \theta$
- Play many real games from start to end
- Update feature weights θ after every move
- TD error $\delta = V(s') - V(s)$
- Weight update $\Delta\theta = \alpha\delta\phi(s)$
- Improve policy: ϵ-greedy w.r.t. $V(s')$

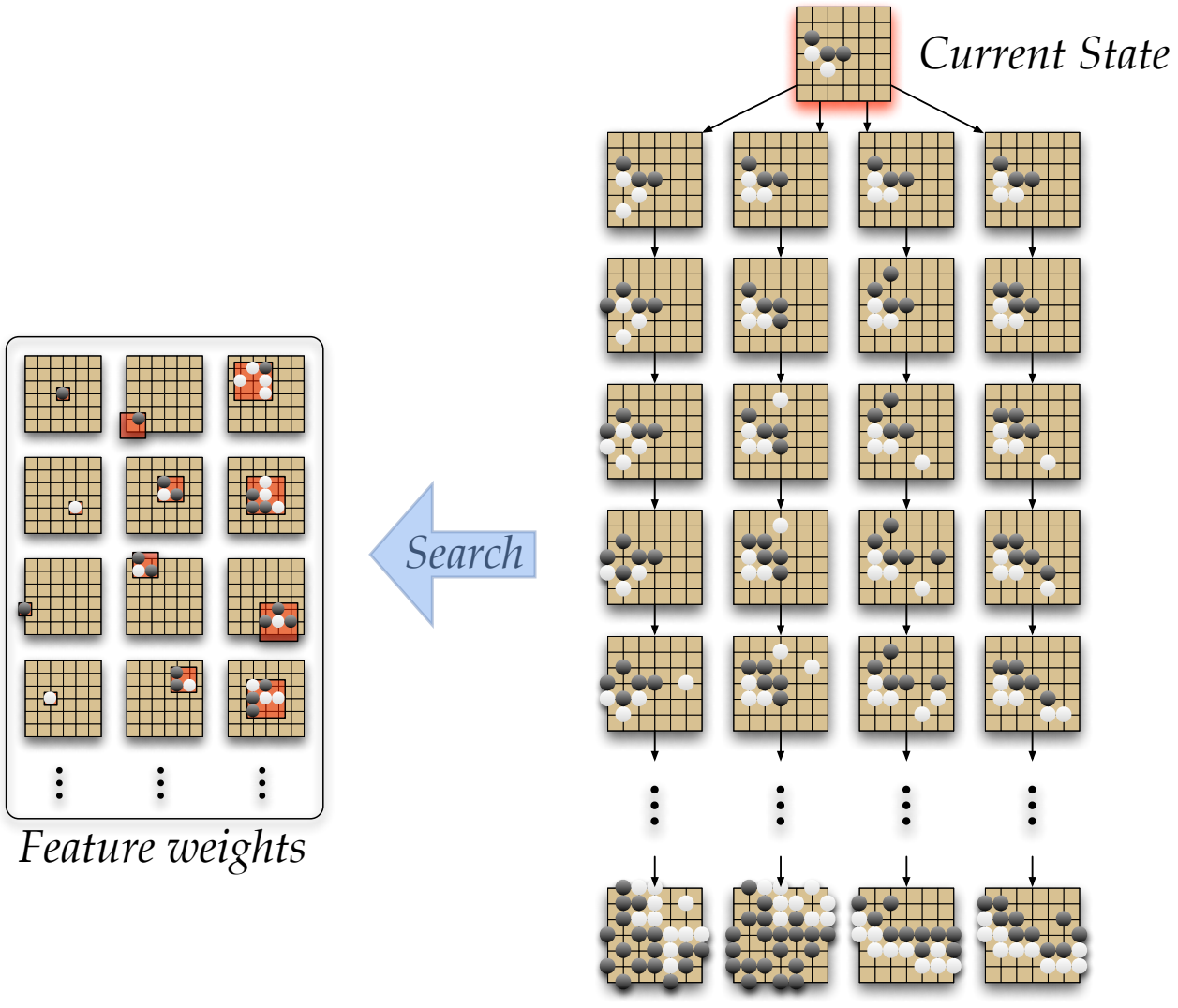
Linear TD Learning in Go (2)



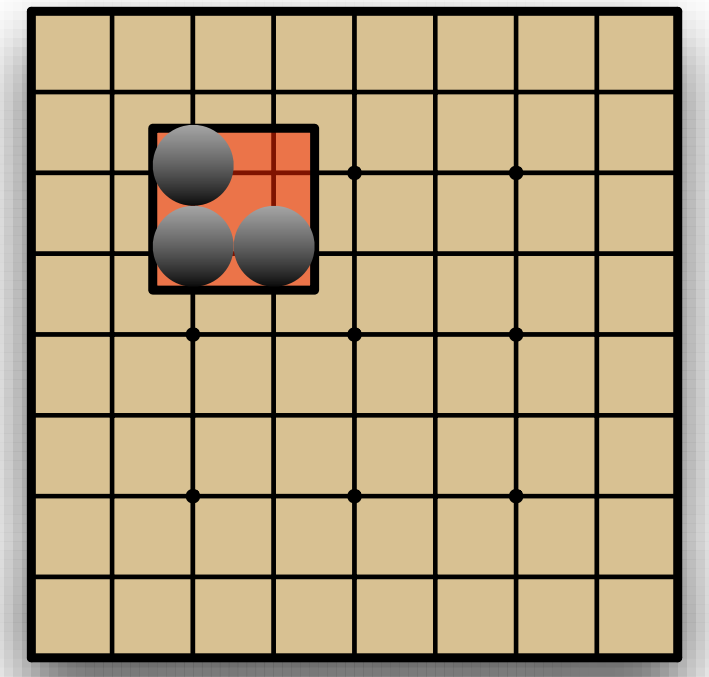
Linear TD Search in Go

- Features $\phi(s)$
- Value function $V(s) = \phi(s)^\top \theta$
- Play many simulated games from current position
- Update feature weights θ after every simulated move
- TD error $\delta = V(s') - V(s)$
- Weight update $\Delta\theta = \alpha\delta\phi(s)$
- Improve simulation policy: ϵ-greedy w.r.t. $V(s')$

Linear TD Search in Go (2)

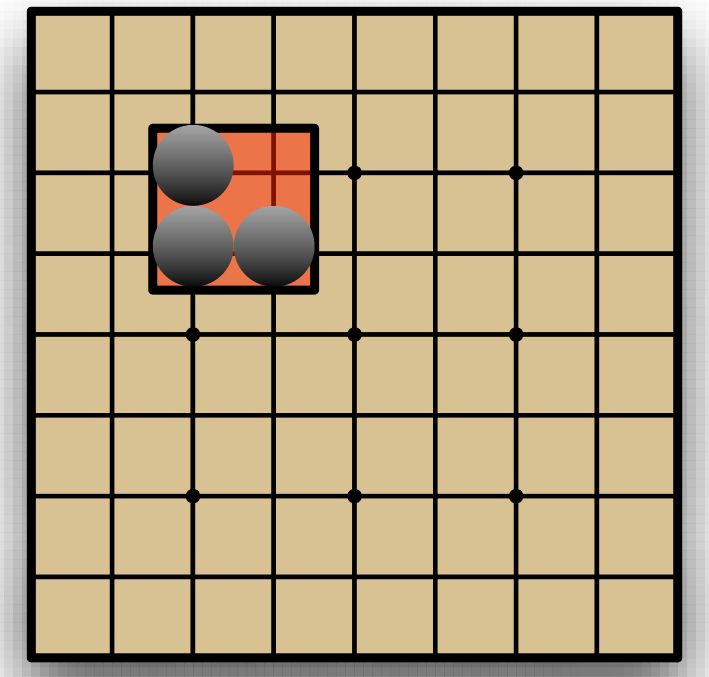


Empty Triangle



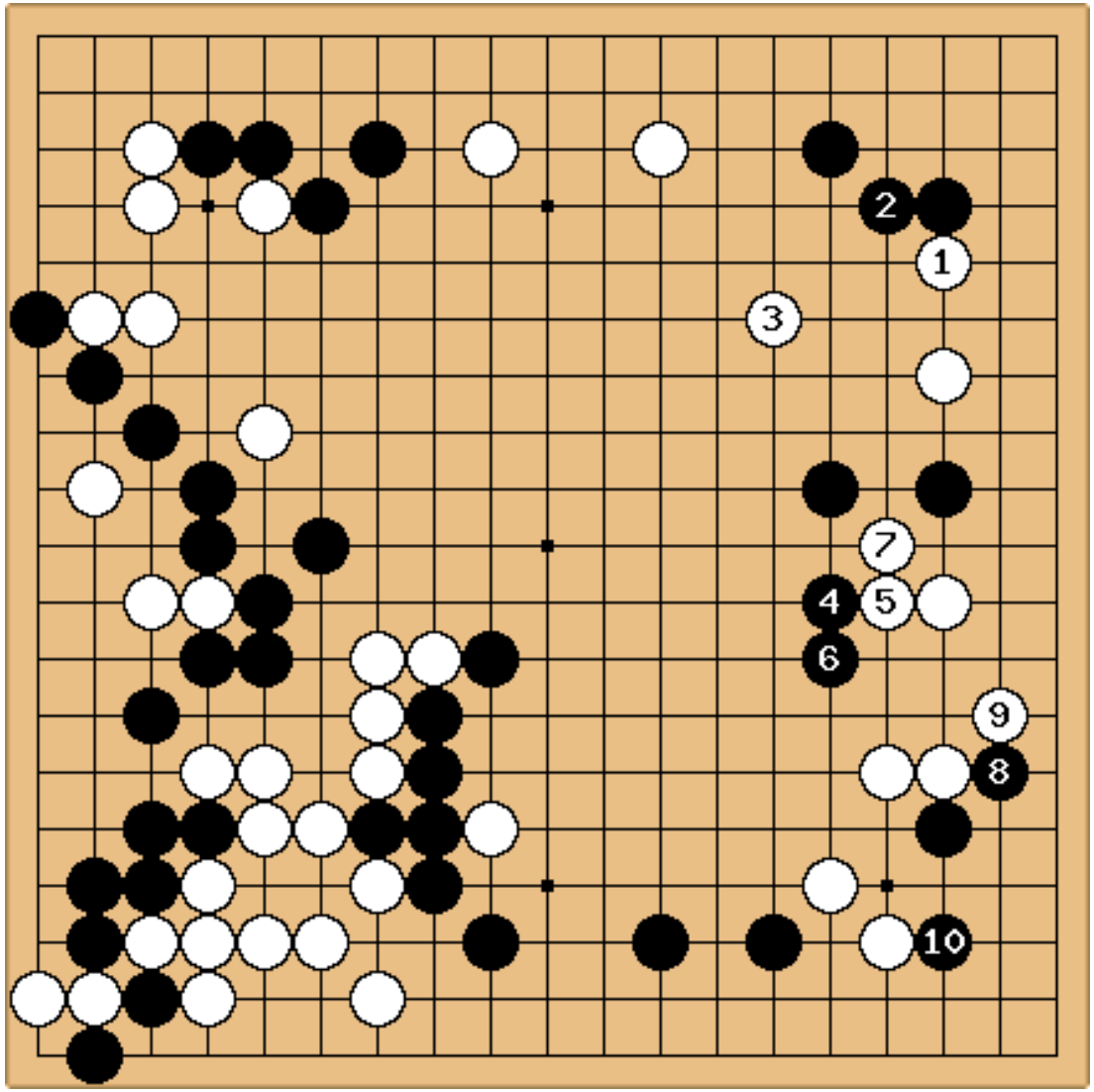
- TD learning: empty triangle shape learns -ve weight
- General knowledge about all games of Go
- Long-term memory

Guzumi

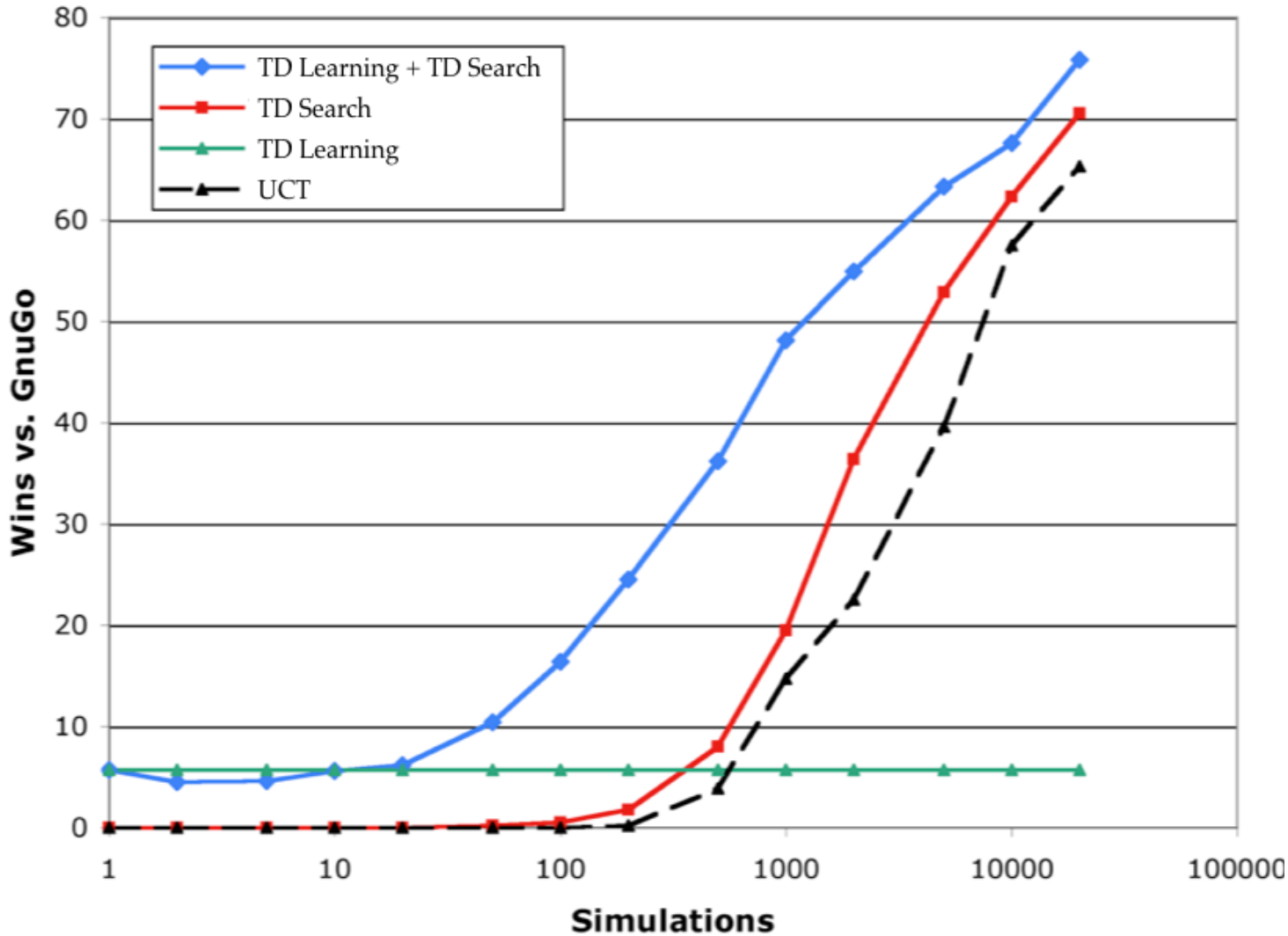


- TD search: empty triangle shape learns +ve weight
- Contextual knowledge specific to current game of Go
- Short-term memory

Blood Vomiting Game



Results of TD search in Go



Bayesian Model-Learning

- Given a prior distribution over models $\mathbb{P}[\eta]$
- and experience $\mathcal{D} = \{s_1, a_1, r_2, ..., s_T\}$
- Compute posterior distribution over models $\mathbb{P}[\eta \mid \mathcal{D}]$
- Using Bayes rule

$$\mathbb{P}[\eta \mid \mathcal{D}] \propto \mathbb{P}[\mathcal{D} \mid \eta] \mathbb{P}[\eta]$$

- Posterior distribution $\mathbb{P}[\eta \mid \mathcal{D}]$ explicitly represents uncertainty

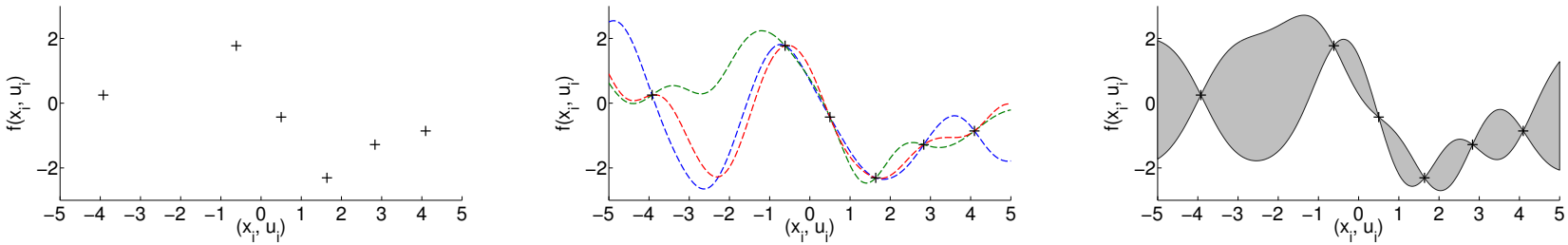
Bayesian Model-Based RL

- Given posterior model distribution $\mathbb{P}[\eta \mid \mathcal{D}]$
- Plan with respect to full distribution
  - Model-based RL: one world model
  - Bayesian model-based RL: many worlds model
- Integrate value over many possible worlds, e.g.

$$V^{\pi}_{bayes}(s) = \mathbb{E}_{\mathbb{P}[\eta|\mathcal{D}]}\left[\mathbb{E}_{\pi}\left[v_t | s_t = s, \mathcal{M}_{\eta}\right]\right]$$
$$V^{*}_{bayes}(s) = \max_{\pi} V^{\pi}_{bayes}(s)$$

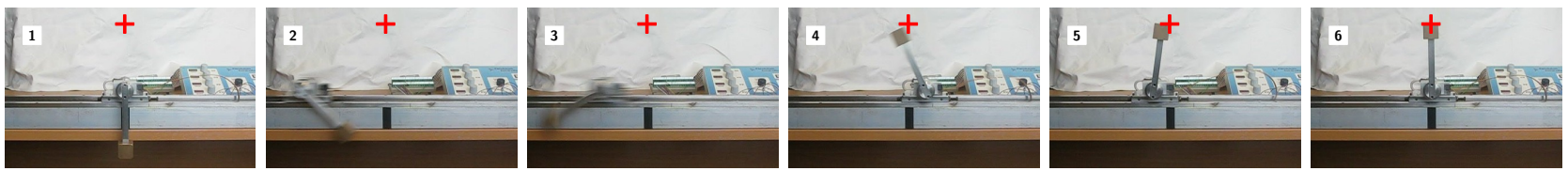
- V^{*}_{bayes} is *Bayes-optimal* w.r.t. prior $\mathbb{P}[\eta]$
  - This finds optimal *stationary* policy π^*
  - Ignoring value-of-information (next lecture)
- But planning problem harder than before

Example: Gaussian Process RL



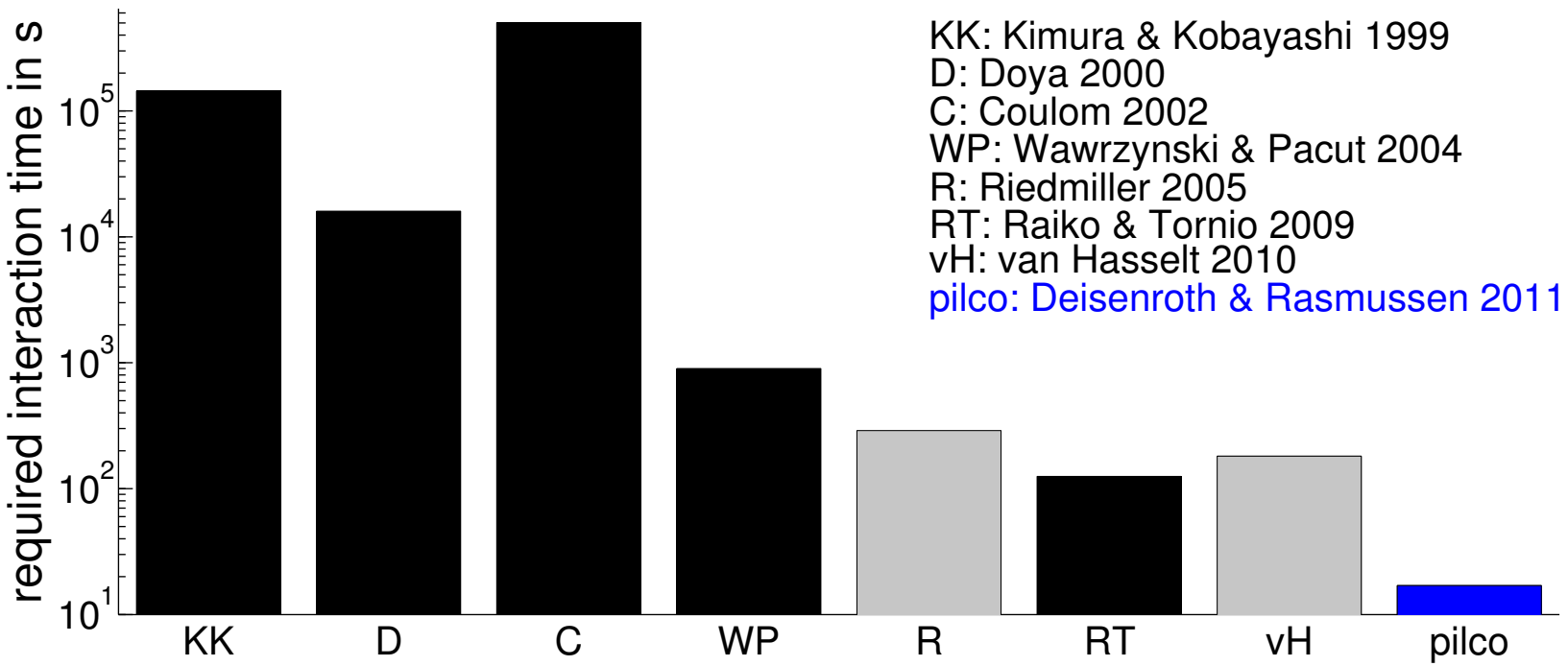
- Many different models are consistent with the data
- Models can be very different
- Planning with respect to one model is brittle
- Posterior distribution over models is estimated by a GP
- Optimal policy is then found with respect to posterior GP
- In this case by following policy gradient

GP-RL on the Cart-Pole

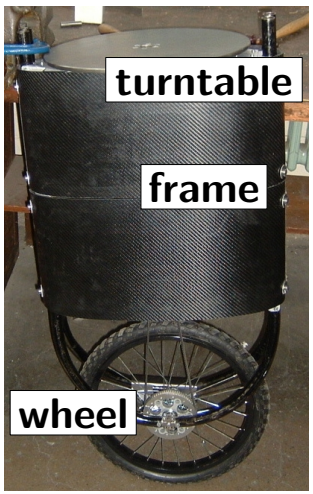


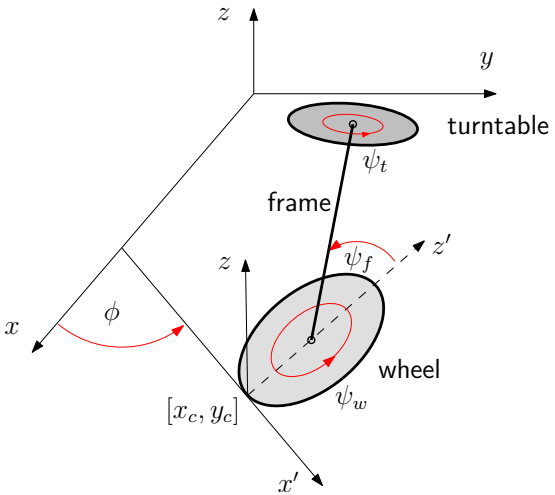
- 4-dimensional state space
- 1-dimensional action space
- Control frequency 10 Hz
- Learns to stay upright after 20 secs of interaction ($<$ 10 trials)

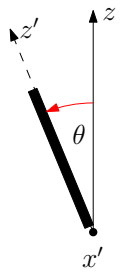
GP-RL on the Cart-Pole (2)



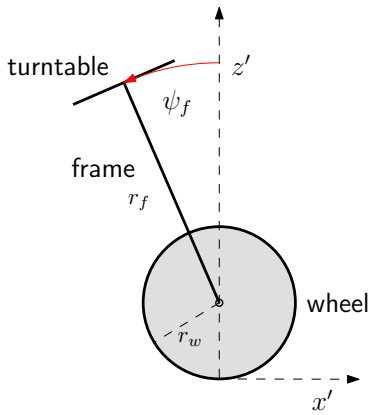
GP-RL on the Unicycle







sideways tilt of wheel



tilt of the frame

- 12-dimensional state space
- 2-dimensional action space (wheel torque, flywheel torque)
- Control frequency 6.66 Hz
- Learns to stay upright after 30 secs of interaction ($<$ 20 trials)

Questions?