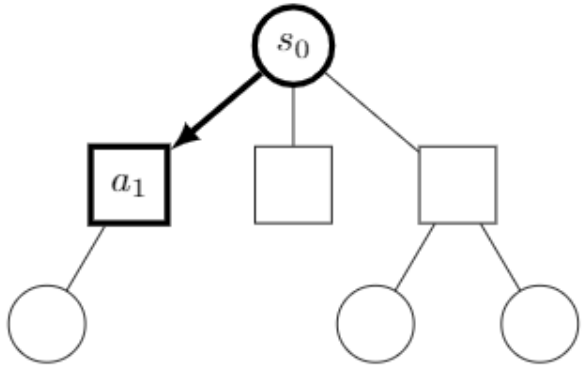
CSE 291 – AI Agents
Deep RL + Search, pre-LLMs

Prithviraj Ammanabrolu

Thanks to David Silver’s DeepMind RL Course and Rich Sutton’s RL Book. Some slides were adapted form there.

Monte Carlo Tree Search

- 4 phases of building out and simulating paths along a search tree
- Various forms of this used in everything from Alpha Zero to modern LLM inference
- For arbitrary problem with start state s_0 and actions a_i
- All states have attributes:
  - Total simulation reward Q(s) and
  - Total no. of visits N(s)



Why Reinforcement Learning?

- Reinforcement Learning:
  - The environment is initially unknown
  - The agent interacts with the environment
  - The agent improves its policy
- Planning:
  - A model of the environment is known
  - The agent performs computations with its model (without any external interaction)
  - The agent improves its policy a.k.a. deliberation, reasoning, introspection, pondering, thought, search

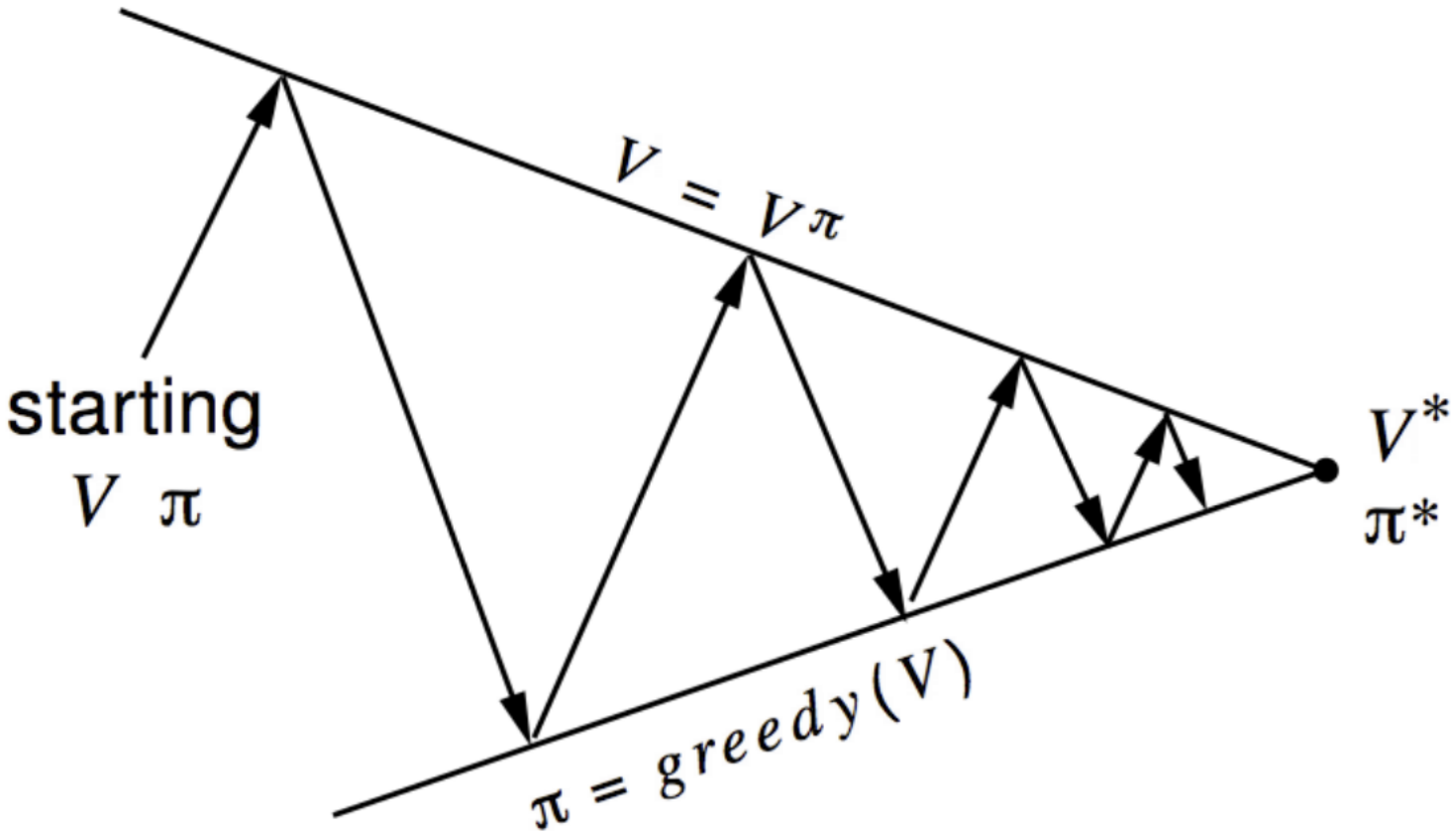
When to use DP

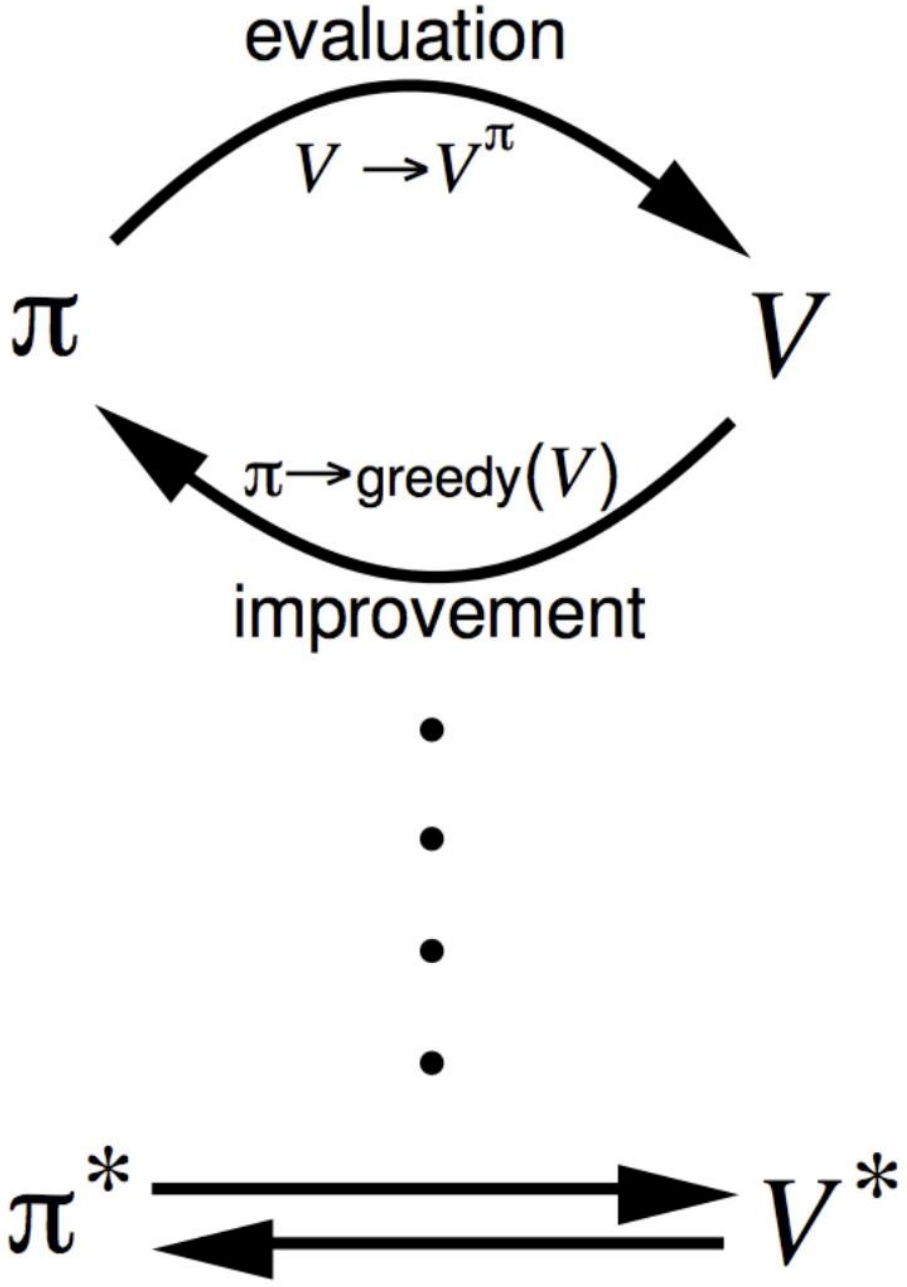
Dynamic Programming is a very general solution method for problems which have two properties:

- Optimal substructure:
  - Principle of optimality applies
  - Optimal solution can be decomposed into subproblems
- Overlapping subproblems:
  - Subproblems recur many times
  - Solutions can be cached and reused
- Markov decision processes satisfy both properties Bellman equation gives recursive decomposition Value function stores and reuses solutions

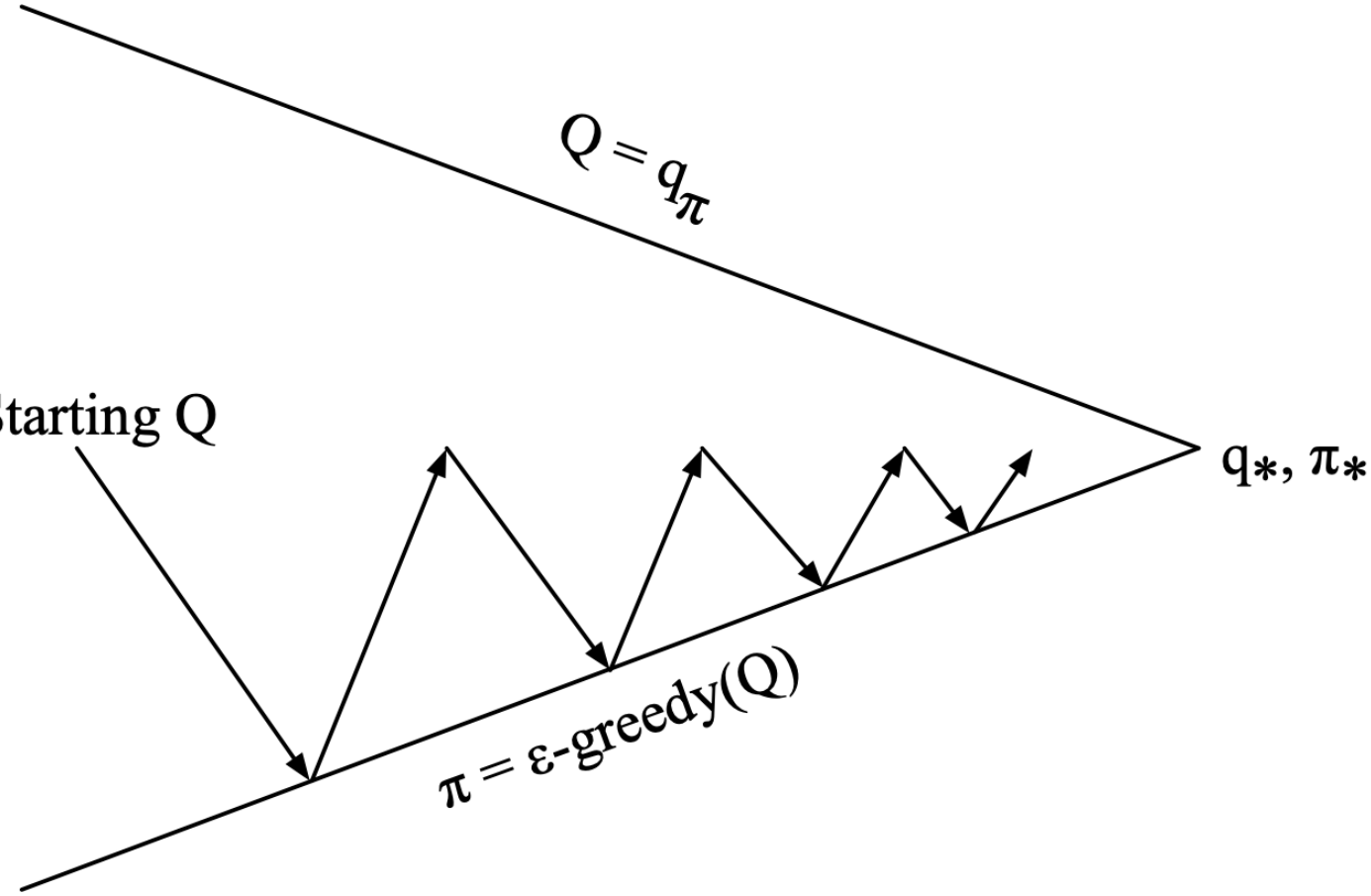
Generalized Policy Iteration

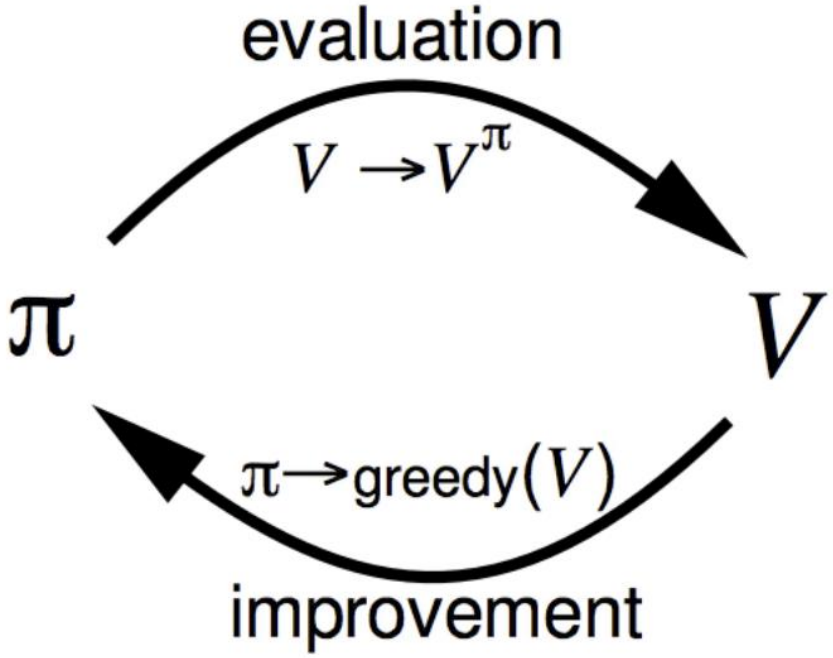
- Both are iterative versions of this

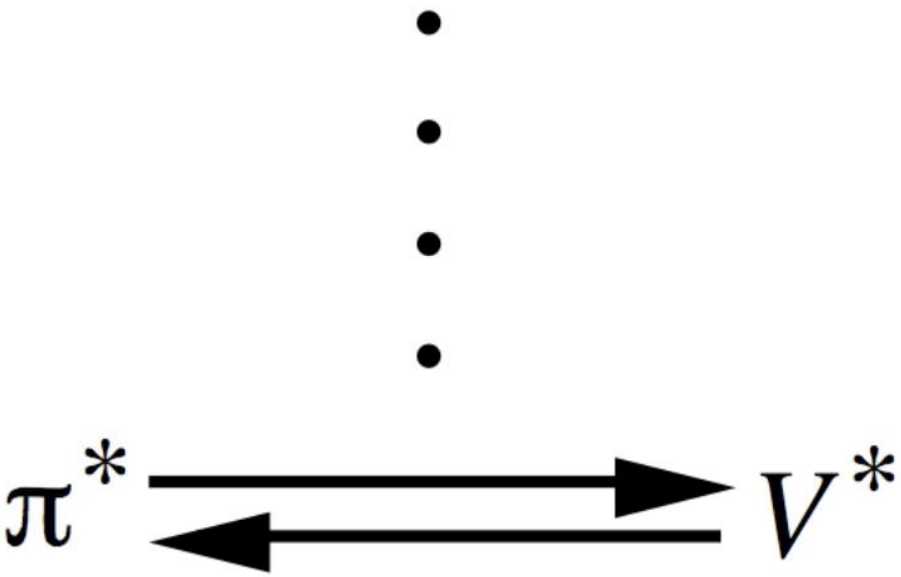




Generalized Policy Iteration with Fn Approximation + Monte Carlo Eval







You can't fully evaluate the entire state space each time

Issues with Monte Carlo estimates

- Need returns for whole trajectory
- The larger the state space is and the longer the horizon, the harder it is to get good estimates
- High variance, very dependent on “getting lucky” and seeing high return trajectories

How to fix? Temporal Difference

- With *Monte Carlo*, we update the value function from a complete episode, and so we **use the actual accurate discounted return of this episode.**

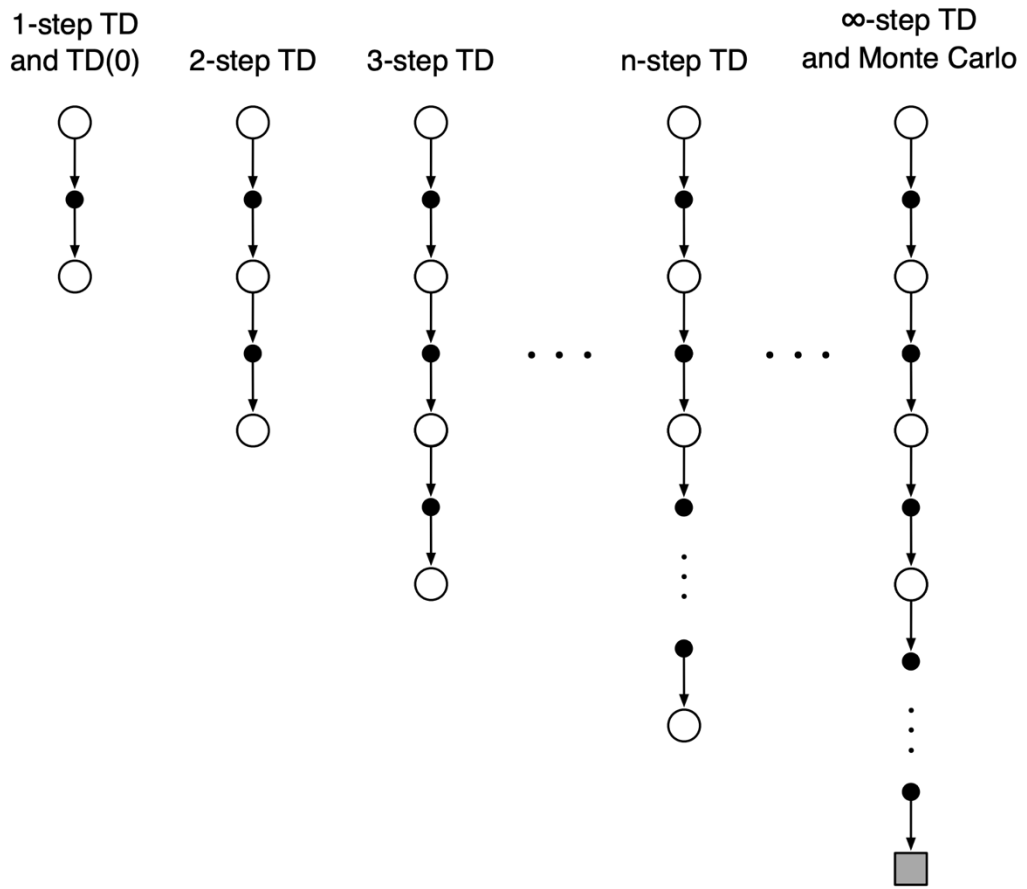
Monte Carlo: $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$

- With *TD Learning*, we update the value function from a step, and we replace G_t, which we don't know, with **an estimated return called the TD target – a bootstrapping method similar to DP**

TD Learning: $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

TD(0) → TD(∞)

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$



TD Advantages

- Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
  - Lower variance
  - Online
  - Incomplete sequences
- Natural idea: use TD instead of MC in our control loop
  - Apply TD to Q(S, A)
  - Use ε-greedy policy improvement
  - Update every time-step

TD Disadvantages

- Bootstrapping means you are chasing a moving target, stability of training very dependent on initialization

How to fix state space is very large

1. Learn from prior experiences

2. Function approximation

On Policy TD Learning - SARSA

- On Policy = learning the policy you are evaluating
- Will not cover SARSA as it is not really used anymore but will cover On Policy later on

Off-policy Learning

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- While following behavior policy $\mu(a|s)$

$$\{S_1, A_1, R_2, \ldots, S_T\} \sim \mu$$

Why is this important?

- Learn from observing humans or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, \ldots, \pi_{t-1}$
- Learn about optimal policy while following exploratory policy
- Learn about multiple policies while following one policy

Q-Learning

- We now consider off-policy learning of action-values Q(s, a)
- Next action is chosen using behavior policy $A_{t+1} \sim \mu(\cdot|S_t)$
- But we consider alternative successor action $A' \sim \pi(\cdot|S_t)$
- And update $Q(S_t , A_t)$ towards value of alternative action from policy you're actually evaluating

$$Q(S_t , A_t) \leftarrow Q(S_t , A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t , A_t))$$

Q-Learning

- We now allow both behavior and target policies to improve
- The target policy π is greedy w.r.t. Q(s, a)
- $\pi(S_{t+1}) = \text{argmax}_{a'} Q(S_{t+1}, a')$
- The behavior policy μ is e.g. e-greedy w.r.t. Q(s, a)
- The Q-learning target then simplifies:

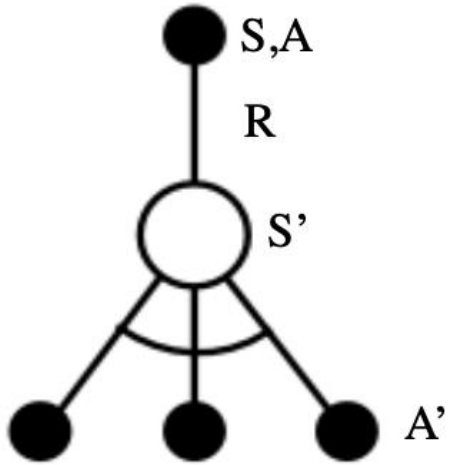
$$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

$$= R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}_{a'} Q(S_{t+1}, a'))$$

$$= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a')$$

Q-Learning

- $Q(S, A) \leftarrow Q(S, A) + \alpha\,(R + \gamma \max_{a'} Q(S', a') - Q(S, A))$
- Q-learning control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$



Q-Learning Full Algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
        $S \leftarrow S'$;
    until S is terminal

Kinda Large Scale RL

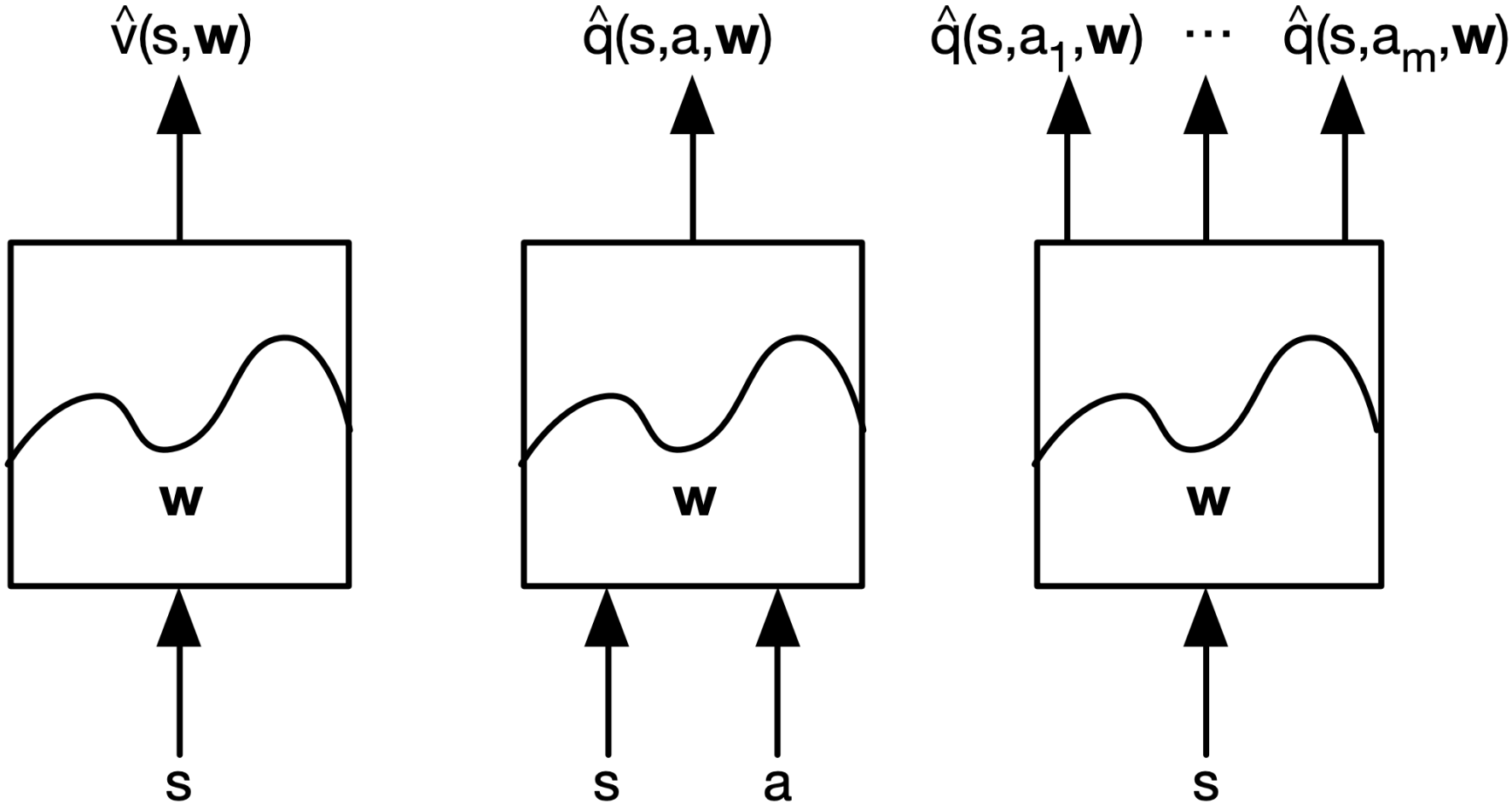
- Reinforcement learning can be used to solve large problems, e.g.
  - Backgammon: 10^{20} states
  - Computer Go: 10^{170} states
  - Helicopter: continuous state space
- How can we scale up the model-free methods for prediction and control from the last two lectures?

Value Function Approximation

- So far we have represented value function by a lookup table
- Every state s has an entry V(s)
- Or every state-action pair s, a has an entry Q(s, a)
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with function approximation

    $$\hat{v}(s, w) \approx v_\pi(s) \text{ or } \hat{q}(s, a, w) \approx q_\pi(s, a)$$
  - Generalize from seen states to unseen states
  - Update parameter w using MC or TD learning

Types of Value Function Approximators



Action-value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, w) \approx q_\pi(S, A)$$

- Minimize mean-squared error between approximate action-value fn $\hat{q}(S, A, w)$ and true action-value fn $q_\pi(S, A)$

$$J(w) = E_\pi [(q_\pi(S, A) - \hat{q}(S, A, w))^2]$$

- Use stochastic gradient descent to find a local minimum

$$-1/2 \nabla_w J(w) = (q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)$$

$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)$$

Deep Neural Nets as function approx.

- Need a Neural Net that is actually able to effectively encode observations and actions
- For the original Atari, this was CNNs
- These days, it is transformers
- Note that you generally need hundreds of k to millions of steps for most environments. The bigger your policy the slower this is

Deep Q Network - DQN

- You actually know all the pieces now
- You put Q-learning together with the function approximation

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

General loop

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Where are we now?

- GPU go brrrr as solution to large state space RL
- Algorithms still not particularly efficient
- No guarantees on anything

Can you do better if you have a Model?

- Everything so far was Model Free
  - No model
  - Learn value function (and/or policy) from experience
- If you know how the world will change in response to your action before you do it, can you use that somehow to influence your actions?
- This is the problem of “given a world model” how to use it.

Model Free vs Model Based 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**

Sample Based Planning

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

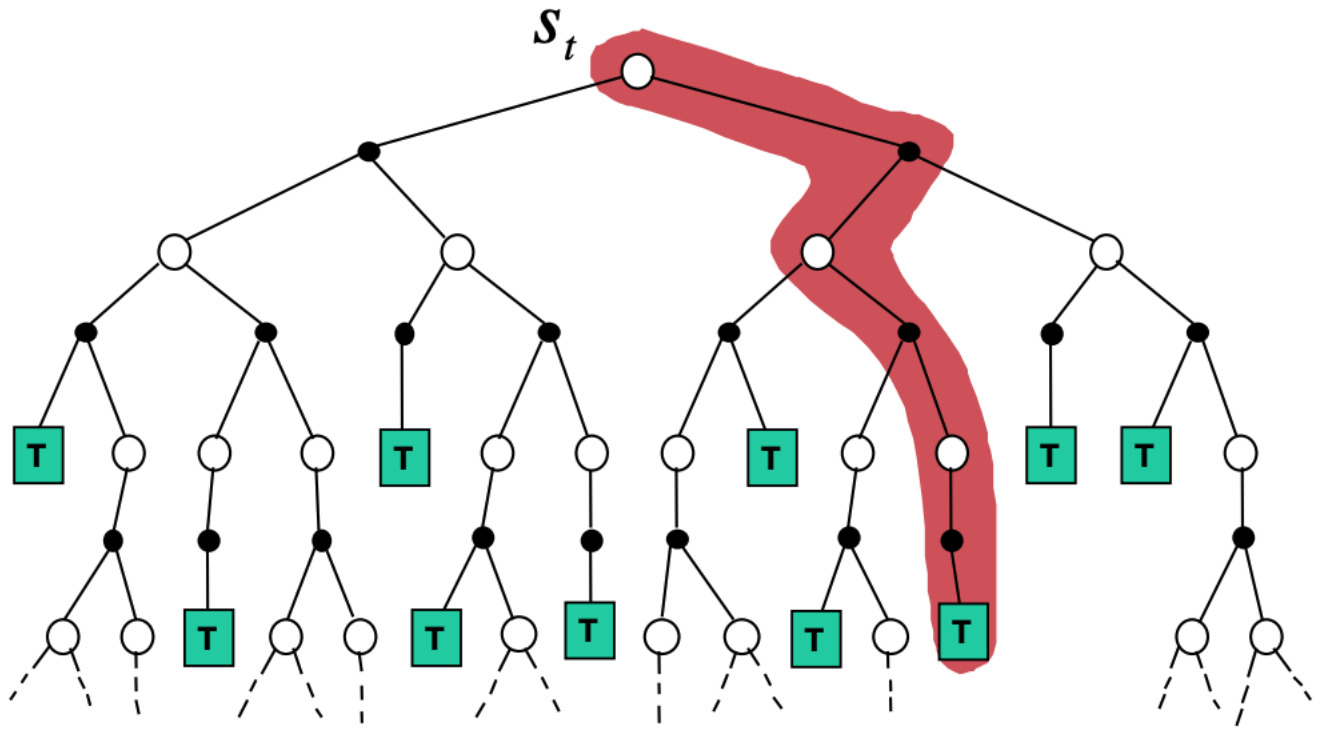
$$S_{t+1} \sim T_\eta(S_{t+1} \mid S_t , A_t)$$

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

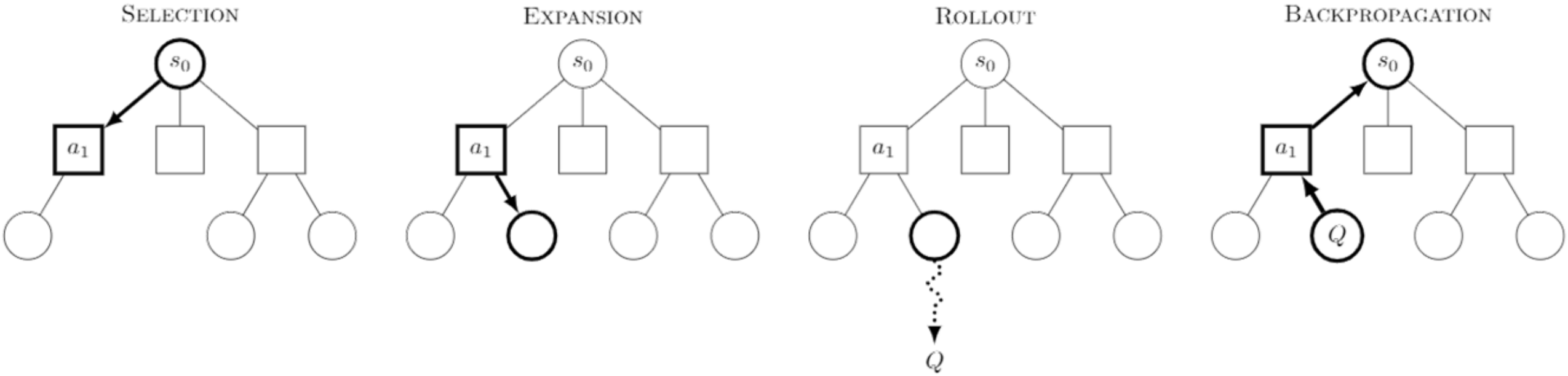
- Apply model-free RL to samples, e.g.: Monte-Carlo control Sarsa Q-learning
- Sample-based planning methods are often more efficient

Simulation Search

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



Revisit MCTS



MCTS (contd)

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

    $s_t \{s_t, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, \ldots, S^k_T\}^K_{k=1} \sim M_{v,\pi}$
  - Evaluate actions by mean return (Monte-Carlo evaluation)

    $Q(s_t, a) = 1/K \sum_{k=1}^{K} G_t \rightarrow q_\pi(s_t, a)$
- Select current (real) action with maximum value

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

MCTS Evaluation

- Given a model M_v
- Simulate K episodes from current state st using current simulation policy π $\{s^t, A^k_t, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, \ldots, S^k_T\}^K_{k=1} \sim M_{v,\pi}$
- Build a search tree containing visited states and actions
- Evaluate states Q(s, a) by mean return of episodes from s, a

$$Q(s, a) = 1 / N(s, a) \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(Su, Au = s, a) Gu \rightarrow q_\pi(s, a)$$

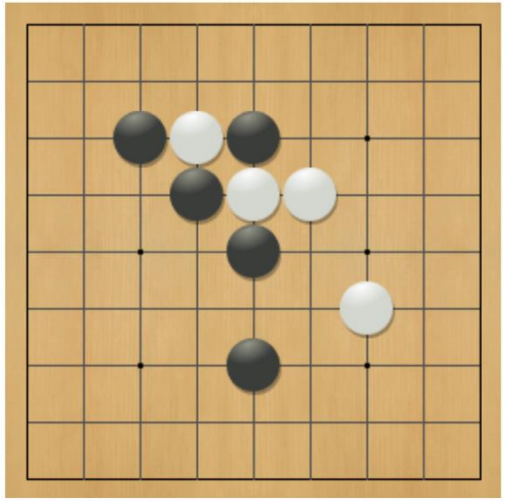
- After search is finished, select current (real) action with maximum value in search tree $a_t = \text{argmax}_{a \in A} Q(s_t, a)$

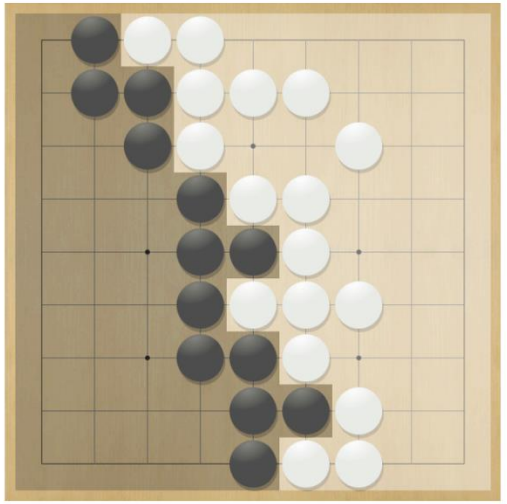
MCTS Simulation

- 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)
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states Q(S, A) by Monte-Carlo evaluation
  - Improve tree policy, e.g. by – greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, Q(S, A) → q*(S, A)

Go Case Study

- 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





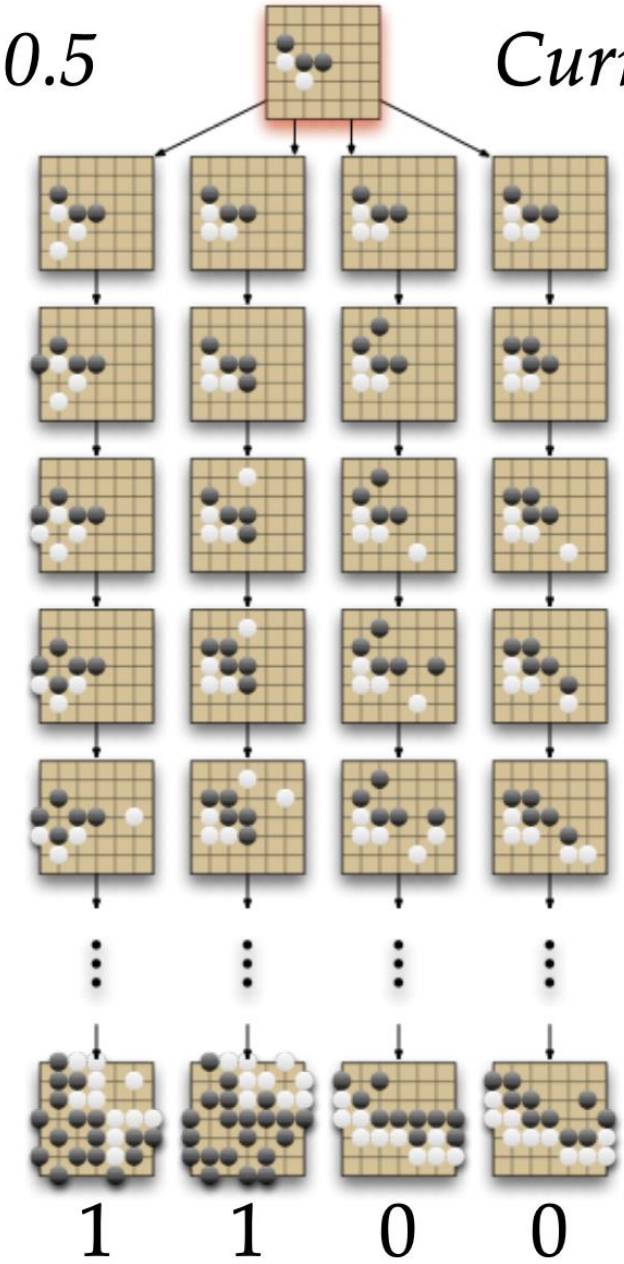
Go Case Study

- How good is a position s?
- Reward function (undiscounted):
  - $R_t = 0$ for all non-terminal steps $t < T$
  - $R_T = 1$ if Black wins
  - $R_T = 0$ if White wins
- Policy $\pi = \langle \pi_B , \pi_W \rangle$ selects moves for both players, Self Play
- Value function (how good is position s):

$$v_\pi(s) = E_\pi [R_T \mid S = s] = P [\text{Black wins} \mid S = s]$$

$$v^*(s) = \max_{\pi B} \min_{\pi W} v_\pi(s)$$

$V(s) = 2/4 = 0.5$

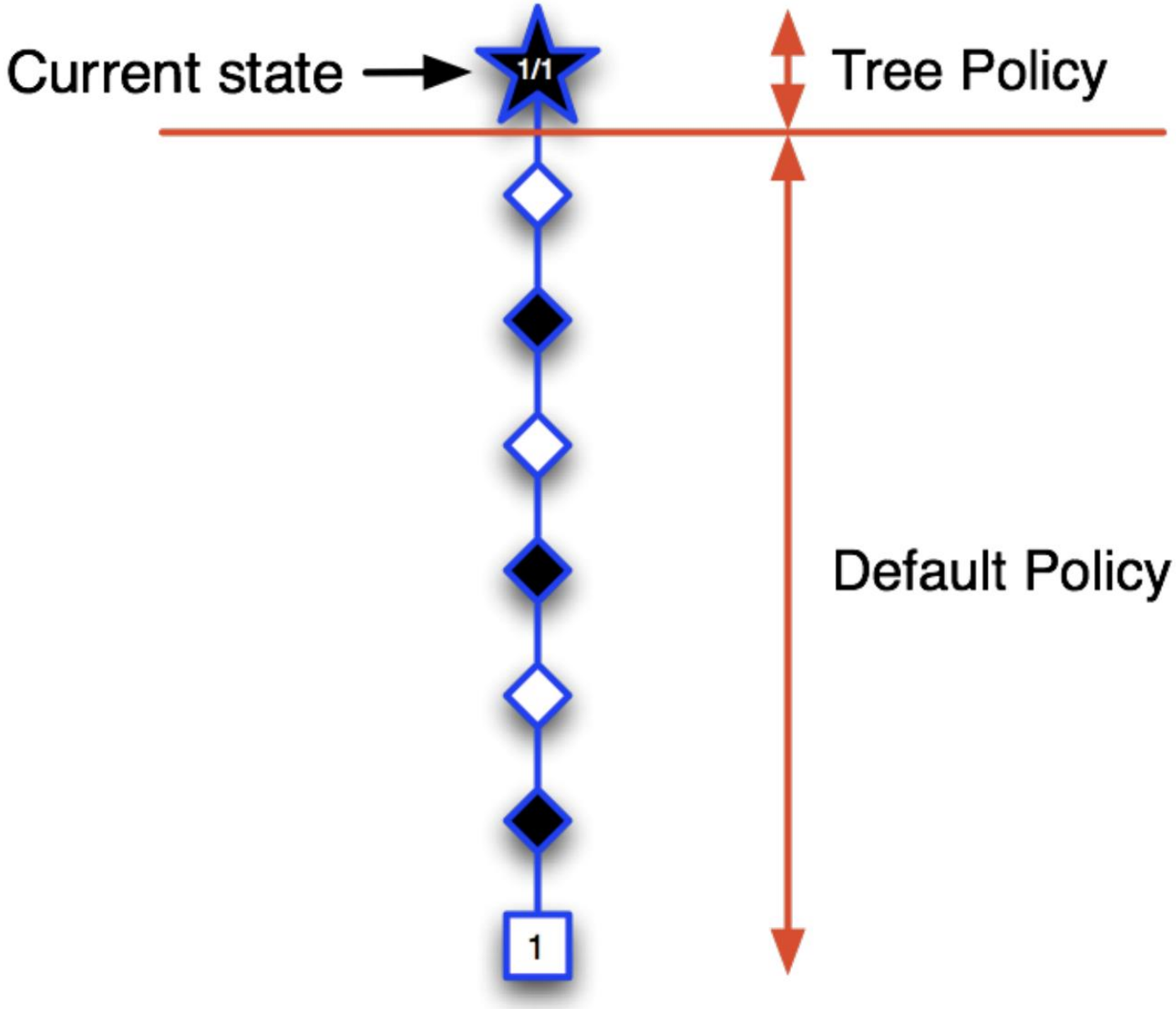


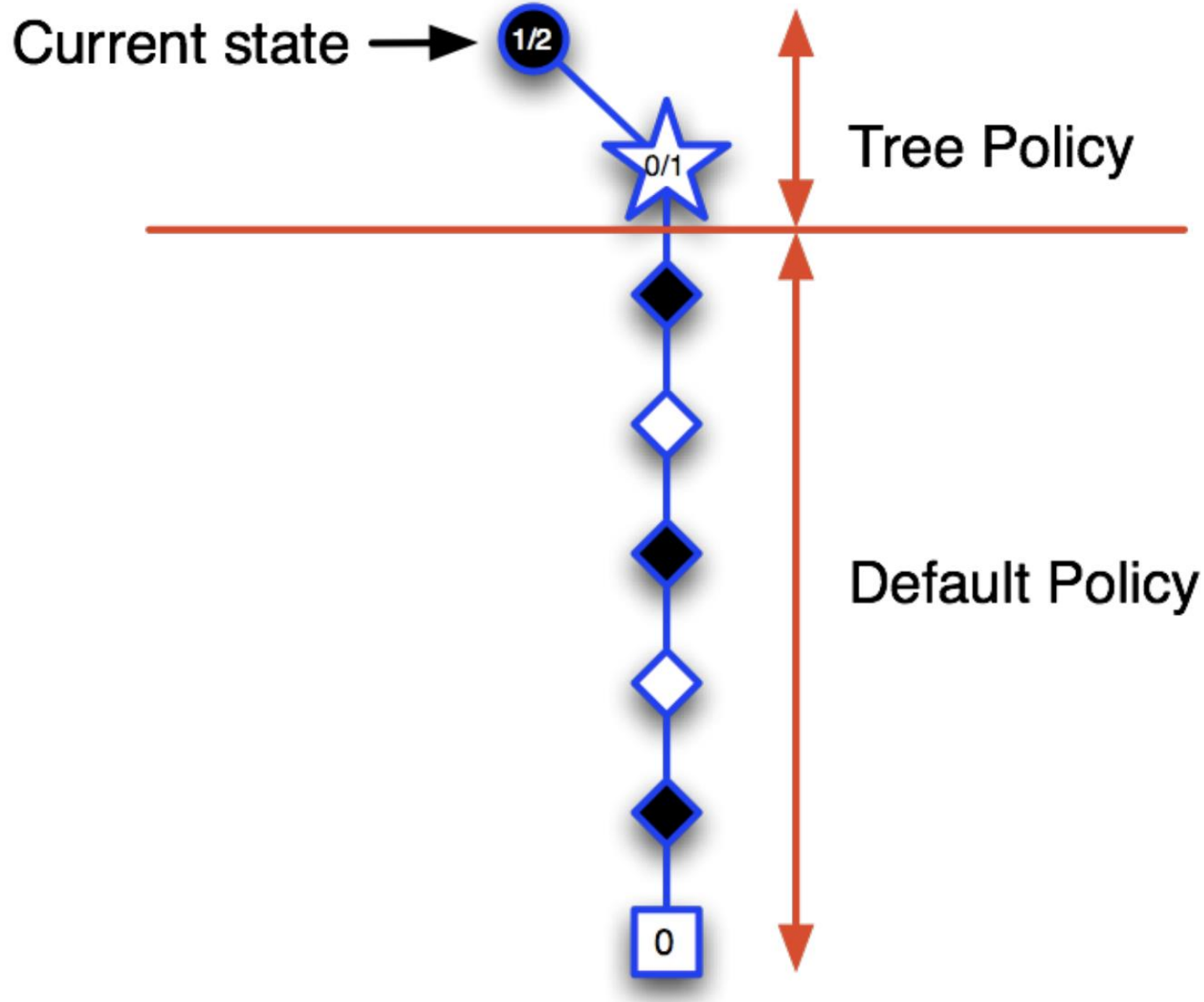
Current position s

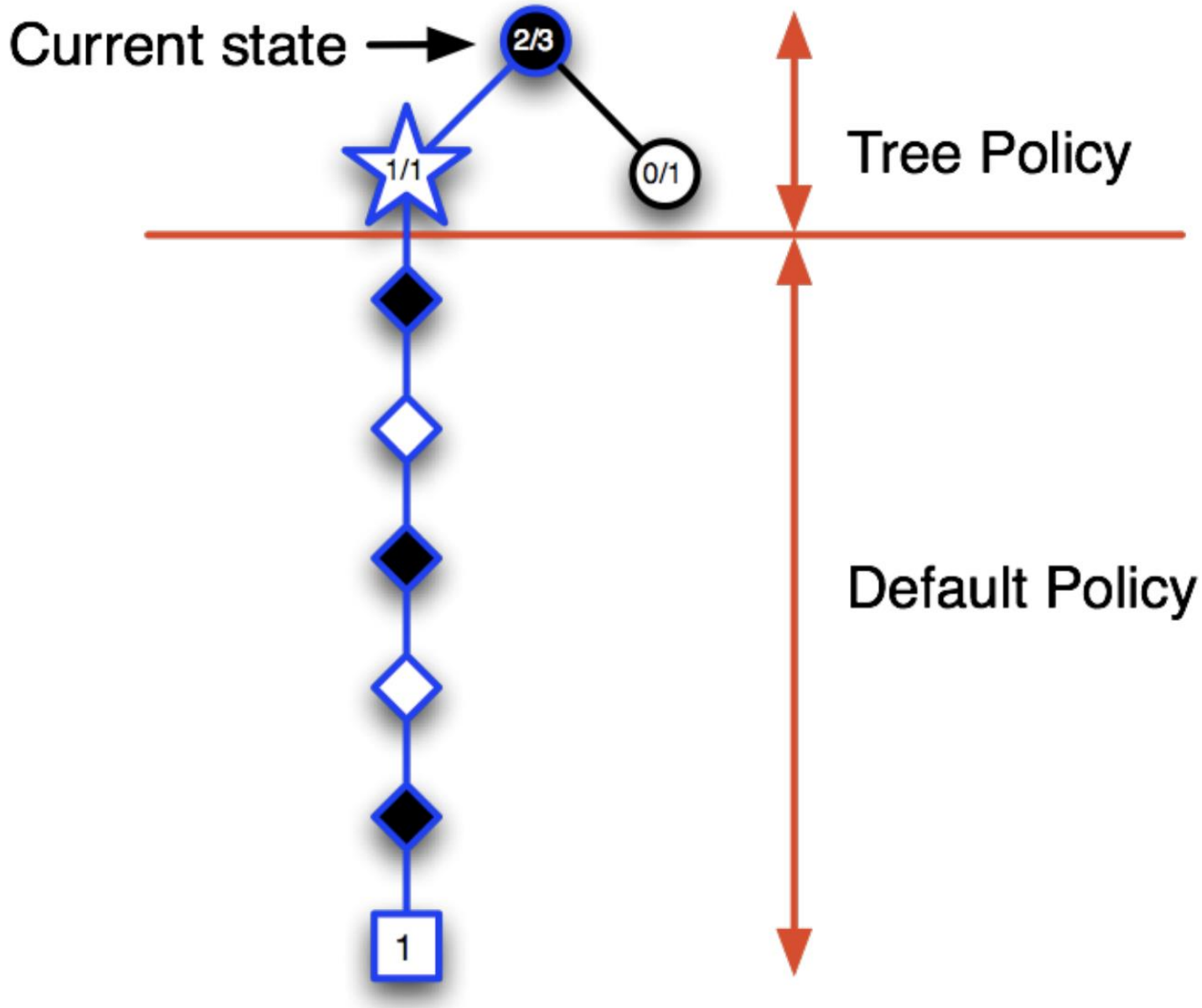
Simulation

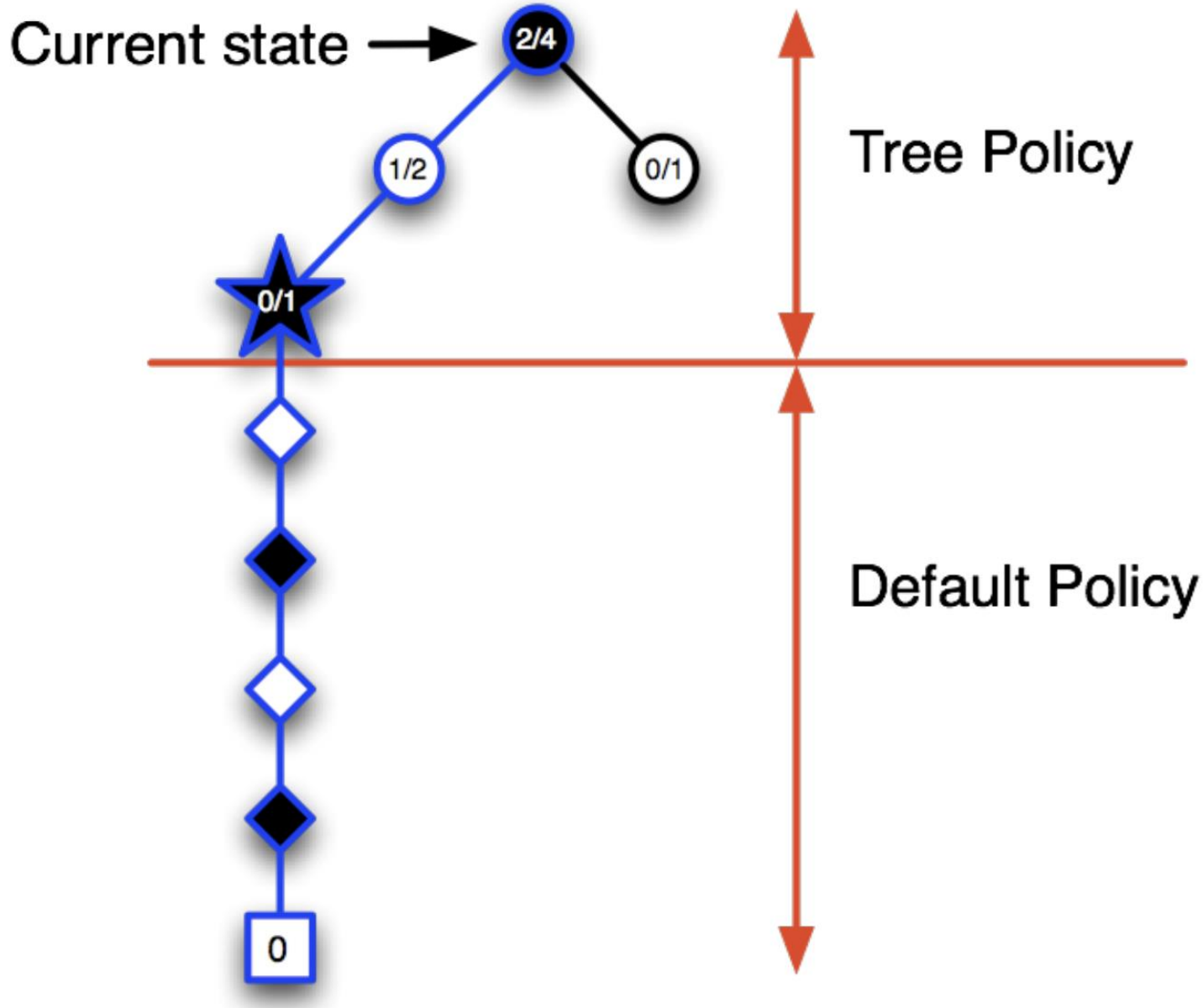
1 1 0 0

Outcomes


Current state
1/1
Tree Policy
Default Policy
1


Current state
1/2
0/1
Tree Policy
Default Policy
0


Current state
2/3
1/1
0/1
Tree Policy
Default Policy
1


Current state
2/4
1/2
0/1
0/1
0
Tree Policy
Default Policy

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

$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

- Select actions based on action-values Q(s, a) e.g. e-greedy
- May also use function approximation for Q

AlphaGo

- Same exact MC method as what we just talked about
- Just use neural nets to learn the probabilities using self play and outcome rewards
- Needed a lot of human games to train the initial value networks
- Also had some hand crafted features to bake in knowledge about the game

AlphaZero

- Relaxed the constraint of requiring a lot of human data and constraints up front by just scaling
- Just do pure online RL

Model Free vs Model Based 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

What is a Model?

- A model M is a representation of an MDP <S, A,T, R>, parametrized by η
- We will assume state space S and action space A are known
- So a model M = $<T_\eta, R_\eta>$ represents state transitions $T_\eta \approx T$ and rewards $R\eta \approx R$

$$S_{t+1} \sim P\eta(S_{t+1} \mid S_t , A_t)$$

$$R_{t+1} = R\eta(R_{t+1} \mid S_t , A_t)$$

- Typically assume conditional independence between state transitions and rewards

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

Learning a Model

- Goal: estimate model M_η from experience $\{S_1, A_1, R_2, \ldots, S_T\}$
- This is a supervised learning problem

  $S_1, A_1 \rightarrow R_2, S_2$

  $S_2, A_2 \rightarrow R_3, S_3$

  $\ldots S_{T-1}, A_{T-1} \rightarrow R_T, S_T$
- Learning s, a → r is a regression problem
- Learning s, a → s' is a density estimation problem
- Pick loss function, e.g. mean-squared error, KL divergence, … Find parameters η that minimizes empirical loss

Model Based RL

- Pick your fav simulation search algo from before and do planning with your model
- Key difference here is that the Model has errors, uncertainty
- What does this mean for how many steps you need to take in an env?

Model Based RL

- Pick your fav simulation search algo from before and do planning with your model
- Key difference here is that the Model has errors, uncertainty
- It will take a lot longer! (Why?)
- This is the overall concept behind MuZero, simultaneously learn both model and policy

Models and Simulation and Reality

- Traditionally we consider two sources of experience
- Real experience: Sampled from environment (true MDP)

  $S' \sim T^a_{s,s'}$

  $R = R^a_s$
- Simulated experience: Sampled from model (approximate MDP)

  $S' \sim T_\eta(S' \mid S, A)$

  $R = R_\eta(R \mid S, A)$
- What's the issue with World Models learned inside a simulation?

Pros and Cons of MBRL

- Pros
  - Can do all the (self, un) supervised learning tricks to learn from large scale data
  - Can reason about uncertainty
- Cons
  - Need model of T first
  - Will build estimate of value from that
  - Two(+) sources of error