CS885 Reinforcement Learning
Lecture 9: May 30, 2018

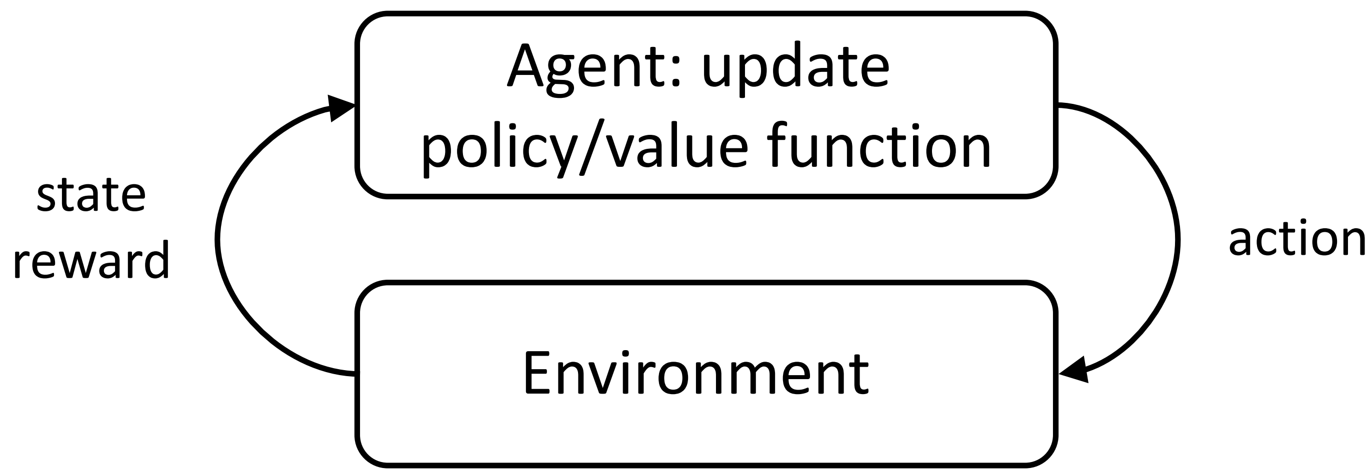
Model-based RL
[SutBar] Chap 8

Outline

- Model-based RL
- Dyna compares model-based with model-free
- Monte-Carlo Tree Search

Model-free RL

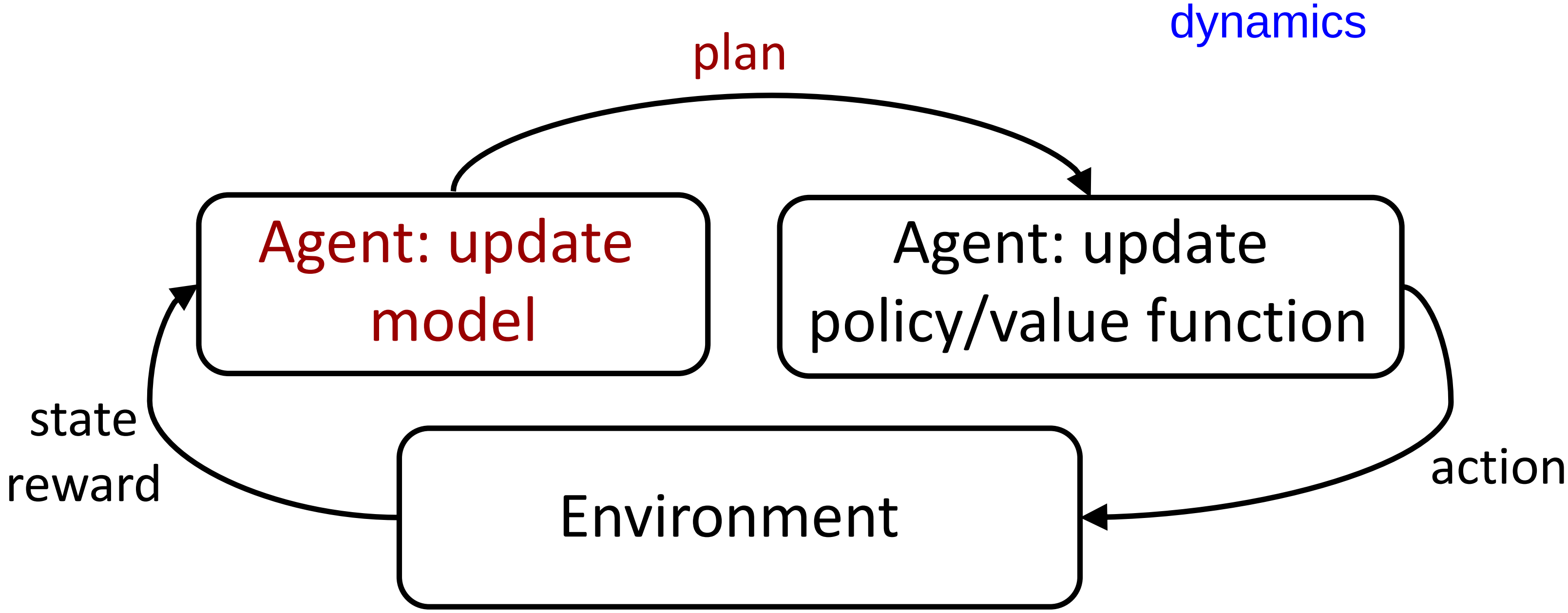
- No explicit transition or reward models
  - Q-learning: **value-based method**
  - Policy gradient: **policy-based method**
  - Actor critic: **policy and value based method**



Model-based RL

- **Learn explicit transition and/or reward model**
  - **Plan based on the model**
  - Benefit: Increased sample efficiency
  - Drawback: Increased complexity

you don't have to actually interact with the env, you can plan "in your head" since you know the dynamics



Maze Example

3	r	r	r	+1
2	u		u	-1
1	u	l	l	l
	1	2	3	4

$\gamma = 1$

Reward is -0.04 for non-terminal states

We need to learn all the transition probabilities!

(1,1)→ (1,2)→ **(1,3)**→ (1,2)→ **(1,3)**→ (2,3)→ (3,3)→ $(4,3)_{+1}$
(1,1)→ (1,2)→ **(1,3)**→ (2,3)→ (3,3)→ (3,2)→ (3,3)→ $(4,3)_{+1}$
(1,1)→ (2,1)→ (3,1)→ (3,2)→ $(4,2)_{-1}$

we learn the transition probabilities by experimenting

P((2,3)|(1,3),r) =2/3
P((1,2)|(1,3),r) =1/3

Use this information in

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} \Pr(s'|s,a)\, V^*(s')$$

we can use this information directly in bellman's equation to do planning!

Model-based RL

- Idea: at each step
  - Execute action
  - Observe resulting state and reward
  - Update transition and/or reward model
  - Update policy and/or value function

Model-based RL (with Value Iteration)

ModelBasedRL(s)

Repeat

Select and execute a

Observe s' and r

Update counts: $n(s,a) \leftarrow n(s,a) + 1$,

$n(s,a,s') \leftarrow n(s,a,s') + 1$

the transition probability of going to s' from (s,a) is the ratio of times we got to s' divided by the total times we were in (s,a)

Update transition: $\Pr(s'|s,a) \leftarrow \frac{n(s,a,s')}{n(s,a)} \; \forall s'$

Update reward: $R(s,a) \leftarrow \frac{r + (n(s,a)-1)R(s,a)}{n(s,a)}$

Solve: $V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} \Pr(s'|s,a)\, V^*(s') \; \forall s$

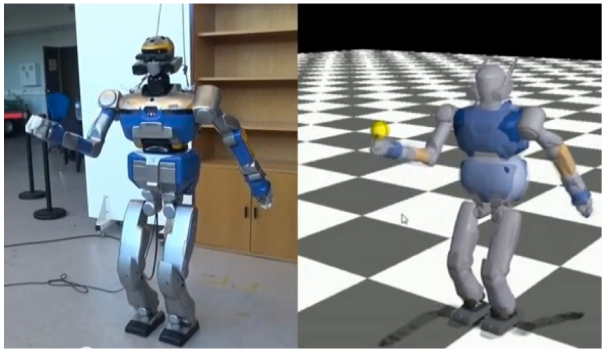
$s \leftarrow s'$

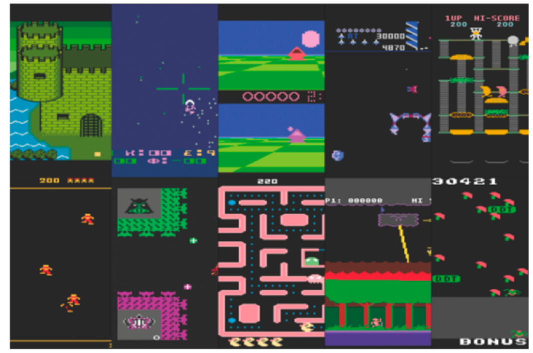
Until convergence of V^*

Return V^*

we plug this into Value iteration and solve Bellman's equation

Complex models





- Use function approx. for transition and reward models
  - Linear model: $pdf(s'|s,a) = N(s'|w^T \begin{bmatrix} s \\ a \end{bmatrix}, \sigma^2 I)$
  - Non-linear models:
    - Stochastic (e.g. Gaussian process):
      $$pdf(s'|s,a) = GP(s|w^T \begin{bmatrix} s \\ a \end{bmatrix}, \sigma^2 I)$$
    - Deterministic (e.g., neural network): $s' = T(s,a) = NN(s,a)$

Partial Planning

- In complex models, fully optimizing the policy or value function at each time step is intractable

- Consider partial planning
  - A few steps of Q-learning
  - A few steps of policy gradient

we only plan for a few steps

these are real states/ actions

we can't count anymore. So we do a gradient step update

Model-based RL (with Q-learning)

error

ModelBasedRL(s)

Repeat

Select and execute a, observe s' and r

Update transition: $w_T \leftarrow w_T - \alpha_T(T(s,a) - s')\nabla_{w_T} T(s,a)$

Update reward: $w_R \leftarrow w_R - \alpha_R(R(s,a) - r)\nabla_{w_R} R(s,a)$

Repeat a few times:

sample $\hat{s}, \hat{a}$ arbitrarily

these are not real states/actions, these are "fake" and are sampled, so we can use our model. We could update Q using both real and simulated experiences (Dyna)

$\delta \leftarrow R(\hat{s},\hat{a}) + \gamma \max_{\hat{a}'} Q(T(\hat{s},\hat{a}),\hat{a}') - Q(\hat{s},\hat{a})$

temporal difference

Update Q: $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q(\hat{s},\hat{a})$

$s \leftarrow s'$

perform a gradient update on our Q-model

so when we update our Q-function, we don't use real experiences, only simulated

Until convergence of Q

Return Q

we could use the Q function to select a

so in model-based, we summarize our previous experiences by building a model. If we have a good model, it will generalize well beyond the actual experiences we gathered over time. If the model isn't good, the generalization won't be good, so we might as well just to model-free using the replay buffer. So it's generally simpler and safer to work with a replay buffer. The problem is that it's unclear how you can genearlize to other state/action pairs that's not part of your experiences. When you have a good model, you can use this to generalize over other state/action pairs. This is beneficial and will allow you to converge faster.

Partial Planning vs Replay Buffer

- Previous algorithm is very similar to Model-free Q-learning with a replay buffer
- Instead of updating Q-function based on samples from replay buffer, generate samples from model

in model-free, we stored our experiences in a replay buffer, then sampled from that buffer to update our Q-function.

- Replay buffer:
  - Simple, real samples, no generalization to other sate-action pairs
- Partial planning with a model

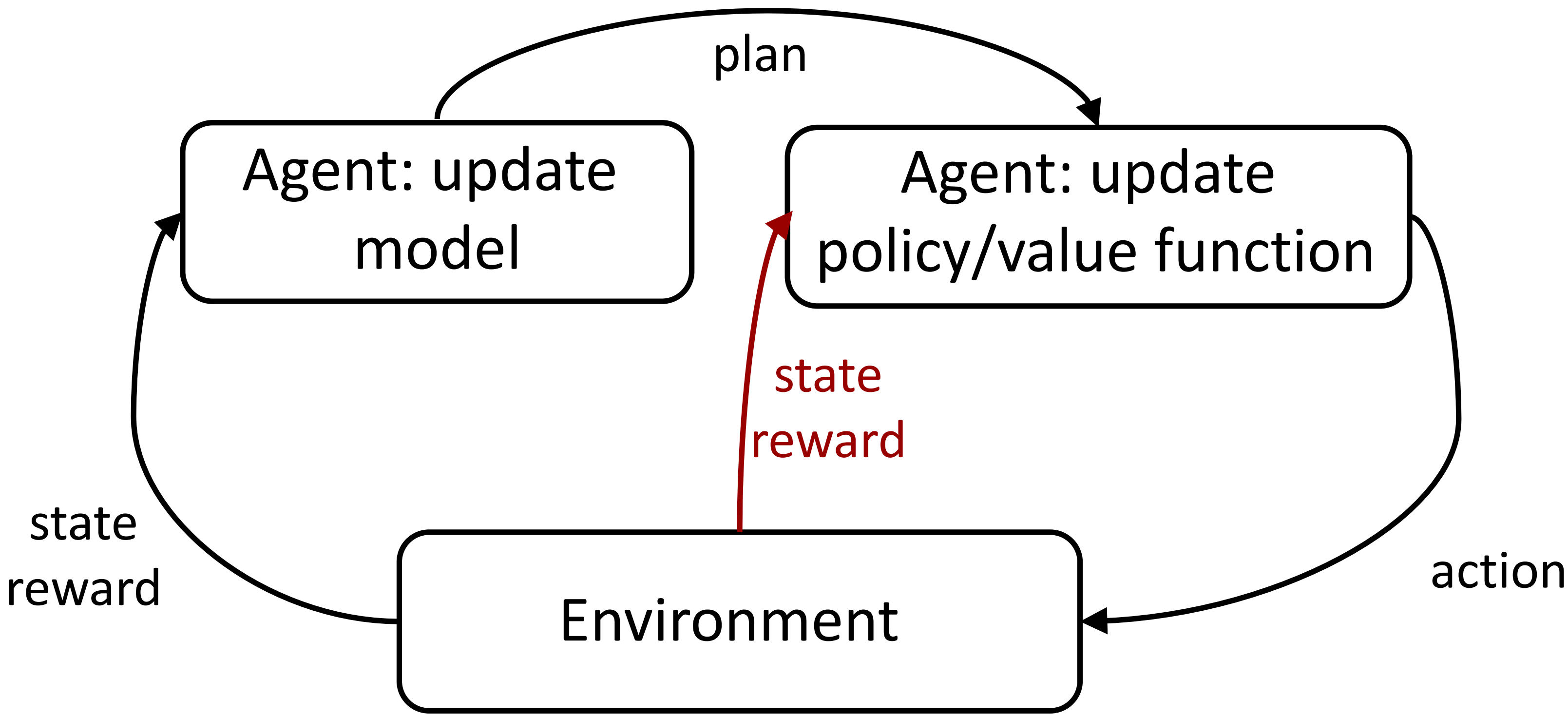
policy updated using simulated samples

  - Complex, simulated samples, generalization to other state-action pairs (can help or hurt)

now, instead of using a replay buffer, we use our previous experiences to learn a model. then we use our model to generate experiences and update our Q-function based on those experiences we can learn a policy using both the sampled and simulated transitions

Dyna

- **Learn explicit transition and/or reward model**
  - Plan based on the model
- **Learn directly from real experience**



Dyna-Q

Dyna-Q(s)

Repeat

using Q-function

Select and execute a, observe s' and r

Update transition: $w_T \leftarrow w_T - \alpha_T(T(s,a) - s')\nabla_{w_T} T(s,a)$

Update reward: $w_R \leftarrow w_R - \alpha_R(R(s,a) - r)\nabla_{w_R} R(s,a)$

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

just like in model-free Q-learning

Update Q: $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q(s,a)$

at first, your model is going to be bad, so you could instead get "enough" real samples first before doing this part

Repeat a few times:

sample $\hat{s}, \hat{a}$ arbitrarily

$\delta \leftarrow R(\hat{s},\hat{a}) + \gamma \max_{\hat{a}'} Q(T(\hat{s},\hat{a}),\hat{a}') - Q(\hat{s},\hat{a})$

Update Q: $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q(\hat{s},\hat{a})$

using simulated data

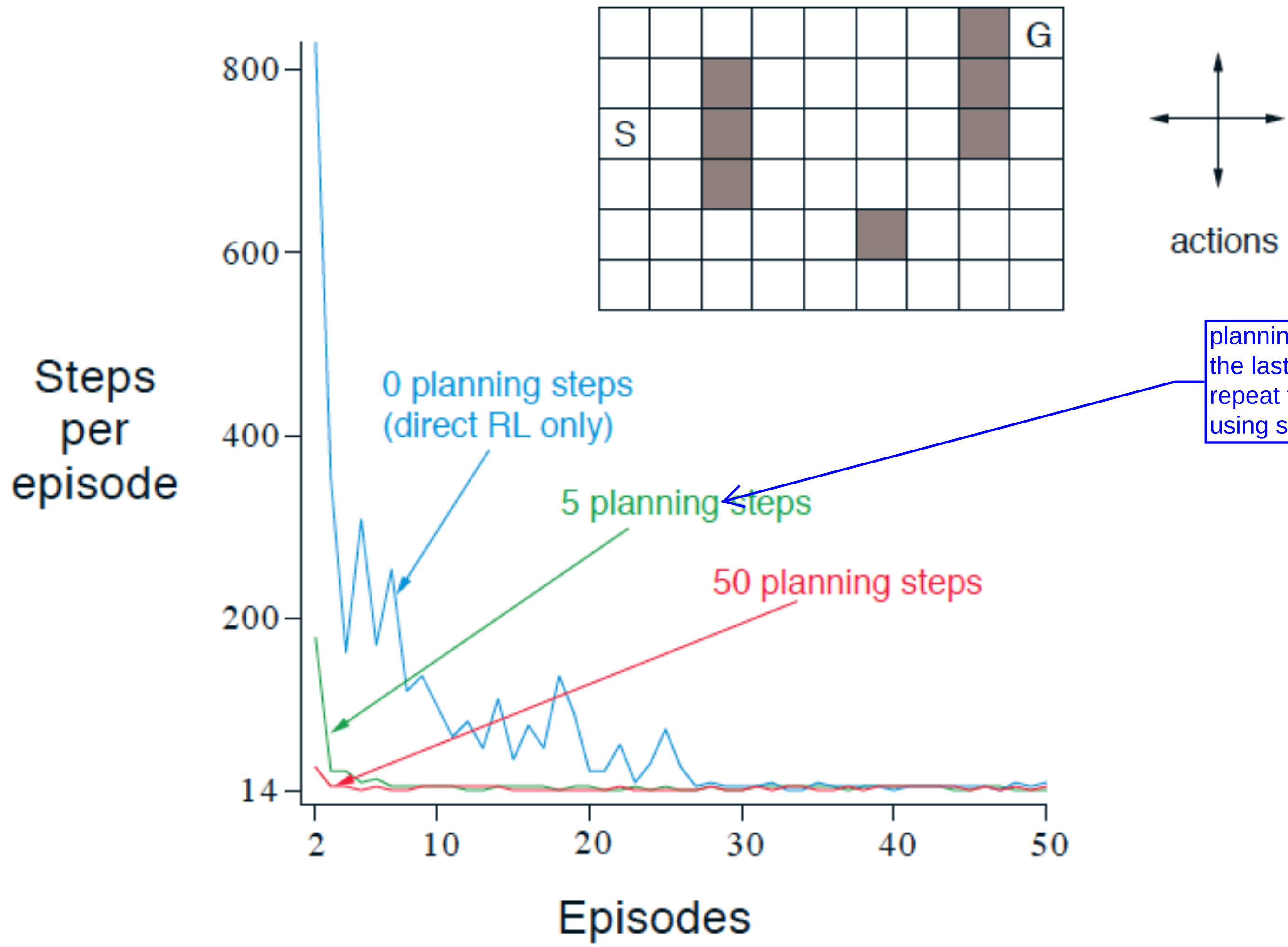
it would make sense that we use the current state for planning

$s \leftarrow s'$

Return Q

Dyna-Q

- Task: reach G from S



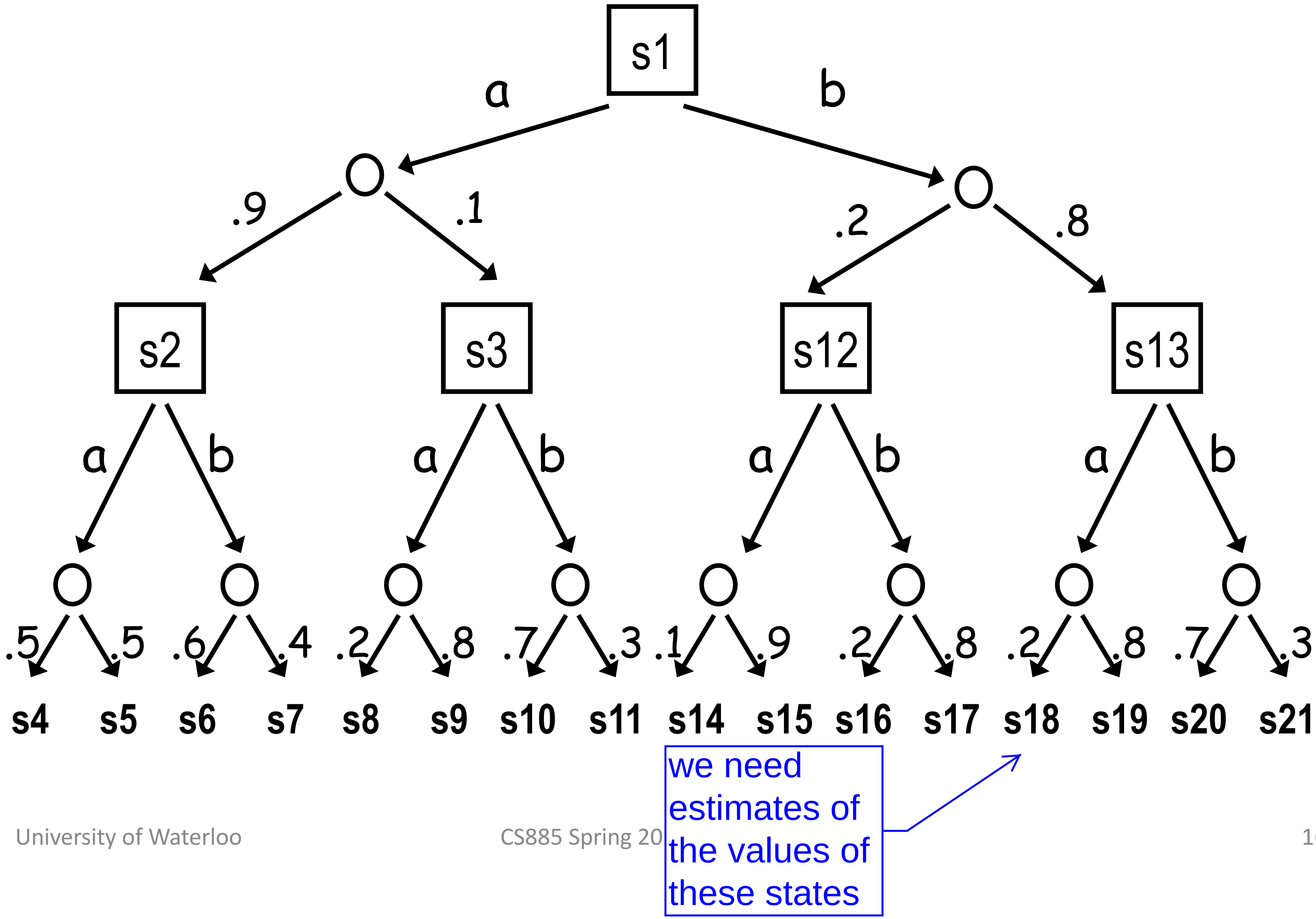
planning steps refers to the last slide where we repeat the Q-updates using simulated data

Planning from current state

- Instead of planning at arbitrary states, plan from the current state
  - This helps improve next action

- Monte Carlo Tree Search

Tree Search



s1
a
b
.9
.1
.2
.8
s2
s3
s12
s13
a
b
a
b
a
b
a
b
.5
.5
.6
.4
.2
.8
.7
.3
.1
.9
.2
.8
.2
.8
.7
.3
s4
s5
s6
s7
s8
s9
s10
s11
s14
s15
s16
s17
s18
s19
s20
s21
we need estimates of the values of these states

Tractable Tree Search

one we get to a leaf, we execute a default policy to do a rollout. We do this N times and then average the total rewards from that state

- Combine 3 ideas:
  - Leaf nodes: approximate leaf values with value of default policy π

this solves problem of going to deep in tree

$$Q^*(s,a) \approx Q^\pi(s,a) \approx \frac{1}{n(s,a)} \sum_{k=1}^{n} G_k$$

simulate transitions from the model from your state

  - Chance nodes: approximate expectation by sampling from transition model

$$Q^*(s,a) \approx R(s,a) + \gamma \frac{1}{n(s,a)} \sum_{s' \sim \Pr(s'|s,a)} V(s')$$

  - Decision nodes: expand only most promising actions

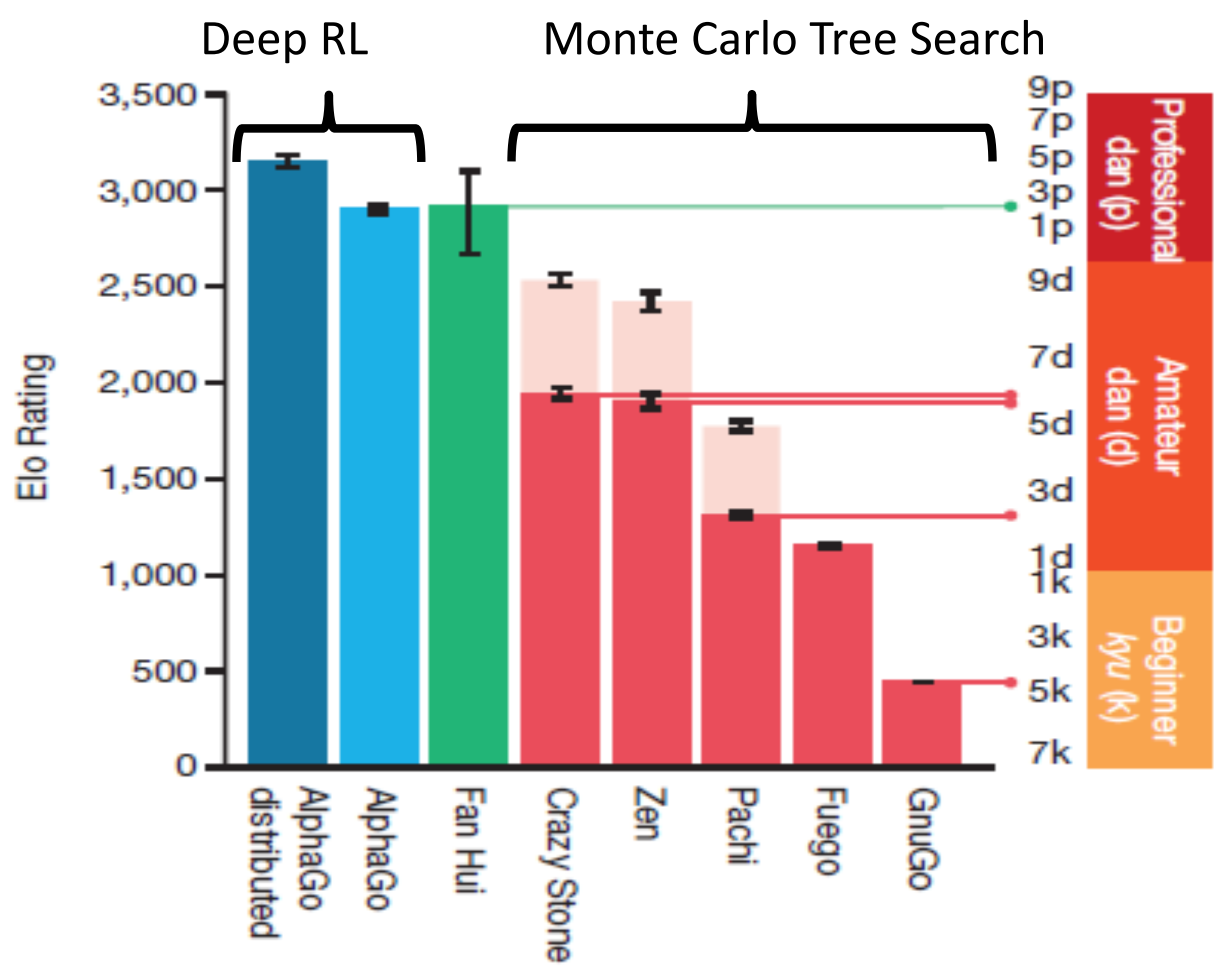
$$a^* = argmax_a Q(s,a) + c\sqrt{\frac{2 \ln n(s)}{n(s,a)}} \quad \text{and} \quad V^*(s) = Q(s,a^*)$$

- Resulting algorithm: Monte Carlo Tree Search

there are many MCTS, but they all use these three ideas

Computer Go

- Oct 2015:



Monte Carlo Tree Search (with upper confidence bound)

UCT(s_0)
create root $node_0$ with state $state(node_0) \leftarrow s_0$
while within computational budget do
$node_l \leftarrow TreePolicy(node_0)$
$value \leftarrow DefaultPolicy(state(node_l))$
$Backup(node_l, value)$
return $action(BestChild(node_0, 0))$

using a Tree policy, select actions until you get to the leaf. Now by leaf, we really mean a node of depth H. We don't want to simulate going all the way down the tree b/c it's computationally inefficient. So instead we pick a depth and traverse down to that depth.

now that we're at the "leaf" we want to estimate its value by doing several rollouts (MC) using a default policy. We calculate the average total returns per rollout

TreePolicy($node$)
while $node$ is nonterminal do
if $node$ is not fully expanded do
return $Expand(node)$
else
$node \leftarrow BestChild(node, C)$
return $node$

so at this point, we have the estimated value of this state. We backup the value from the "leaf" all the way up to the root. So we update the Q-value for each node all the way up to the root.

Monte Carlo Tree Search (continued)

Expand($node$)
choose $a \in$ untried actions of $A(state(node))$
add a new child $node'$ to $node$
with $state(node') \leftarrow T(state(node), a)$
return $node'$

deterministic transition

BestChild($node$,c)

return $\arg\max_{node' \in children(node)} V(node') + c\sqrt{\frac{(2\ln n(node))}{n(node')}}$

DefaultPolicy($node$)
while $node$ is not terminal do
sample $a \sim \pi(a|state(node))$
$s' \leftarrow T(state(node), a)$
return $R(s, a)$

can be any policy, even random

Monte Carlo Tree Search (continued)

Single Player

Backup($node$,$value$)

while $node$ is not null do

$$V(node) \leftarrow \frac{n(node)V(node)+value}{n(node)+1}$$

$$n(node) \leftarrow n(node) + 1$$

$$node \leftarrow parent(node)$$

Two Players (adversarial)

BackupMinMax($node$,$value$)

while $node$ is not null do

$$V(node) \leftarrow \frac{n(node)V(node)+value}{n(node)+1}$$

$$n(node) \leftarrow n(node) + 1$$

$$\boxed{value \leftarrow -value}$$

$$node \leftarrow parent(node)$$

this assumes discrete states and actions

have to adapt this to be for continuous space

AlphaGo

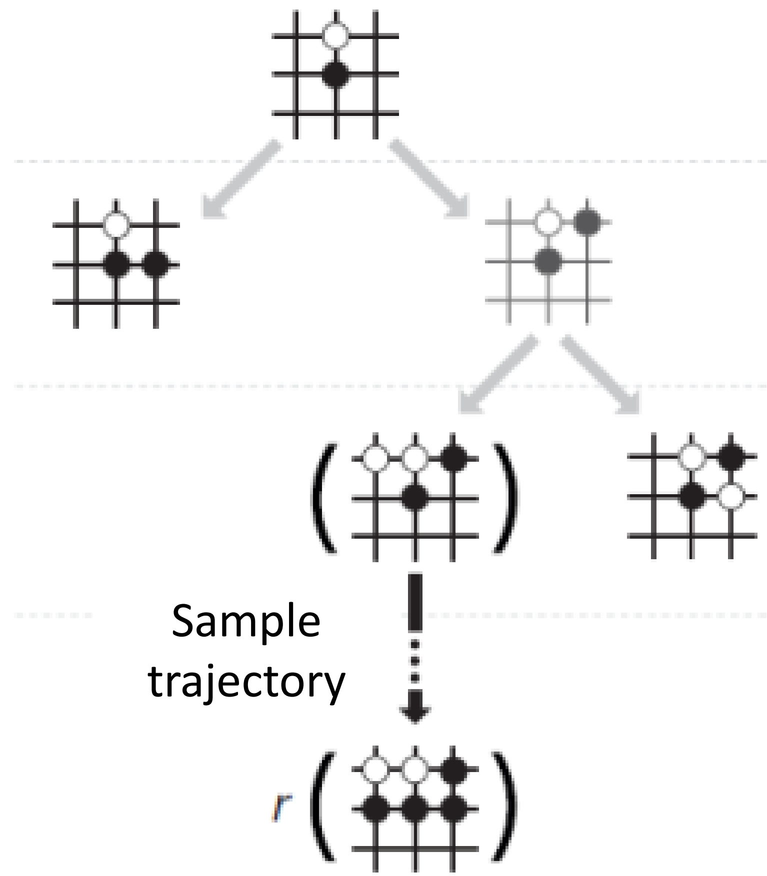
Four steps:

1. Supervised Learning of Policy Networks
2. Policy gradient with Policy Networks
3. Value gradient with Value Networks
4. Searching with Policy and Value Networks
   – Monte Carlo Tree Search variant

Search Tree

- At each edge store
  $Q(s,a), \pi(a|s), n(s,a)$

- Where $n(s,a)$ is the visit count of (s,a)



Simulation

- At each node, select edge a^* that maximizes

$$a^* = argmax_a \, Q(s,a) + u(s,a)$$

value network from NN

rollout by executing default policy

- where $u(s,a) \propto \frac{\pi(a|s)}{1+n(s,a)}$ is an exploration bonus

$$Q(s,a) = \frac{1}{n(s,a)} \sum_i 1_i(s,a) \left[\lambda V_{\boldsymbol{w}}(s) + (1-\lambda) G_i\right]$$

$$1_i(s,a) = \begin{cases} 1 & if \ (s,a) \ was \ visited \ at \ iteration \ i \\ 0 & otherwise \end{cases}$$