CS 188: Artificial Intelligence

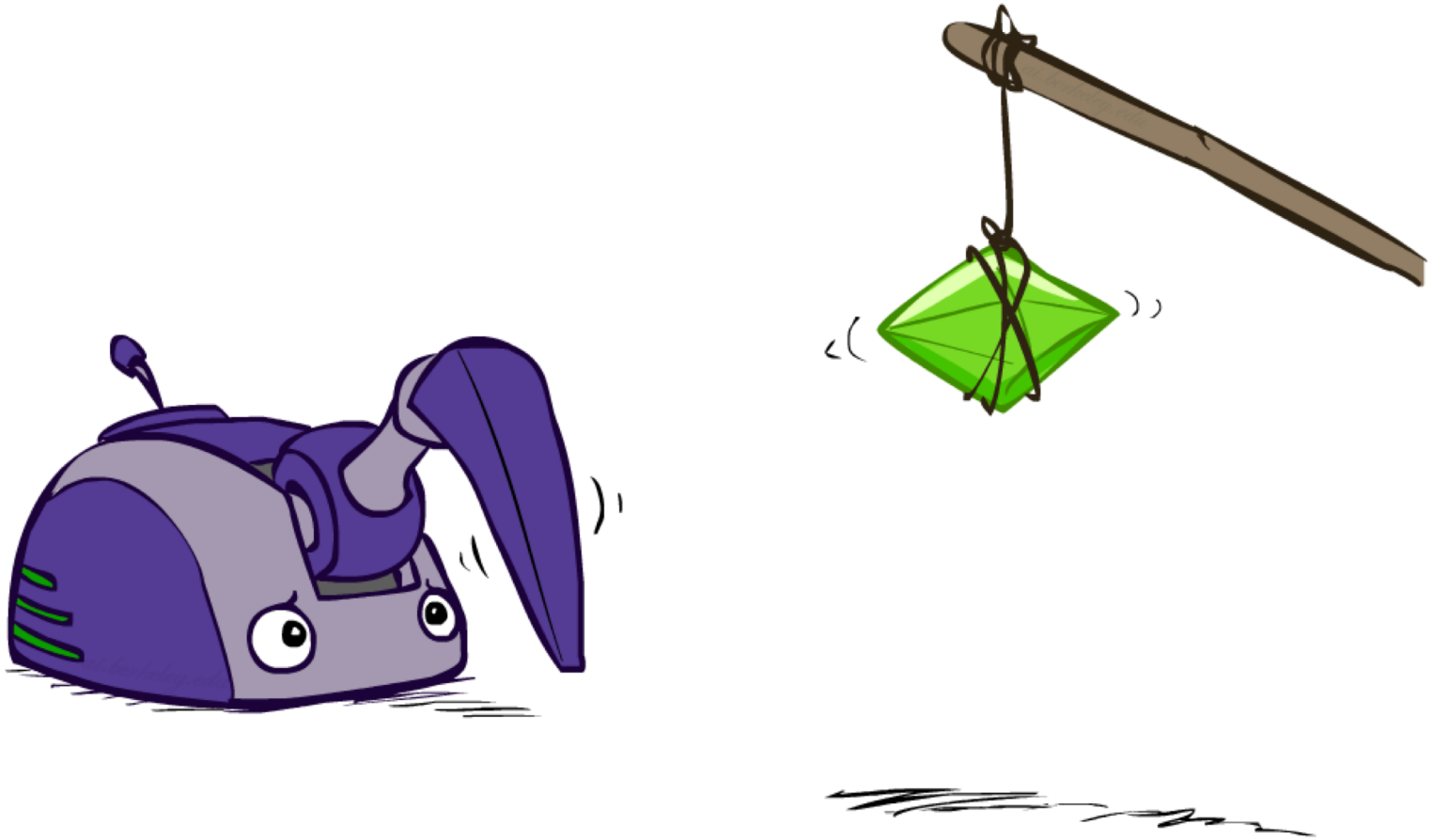
Reinforcement Learning

Instructors: Pieter Abbeel and Dan Klein

University of California, Berkeley

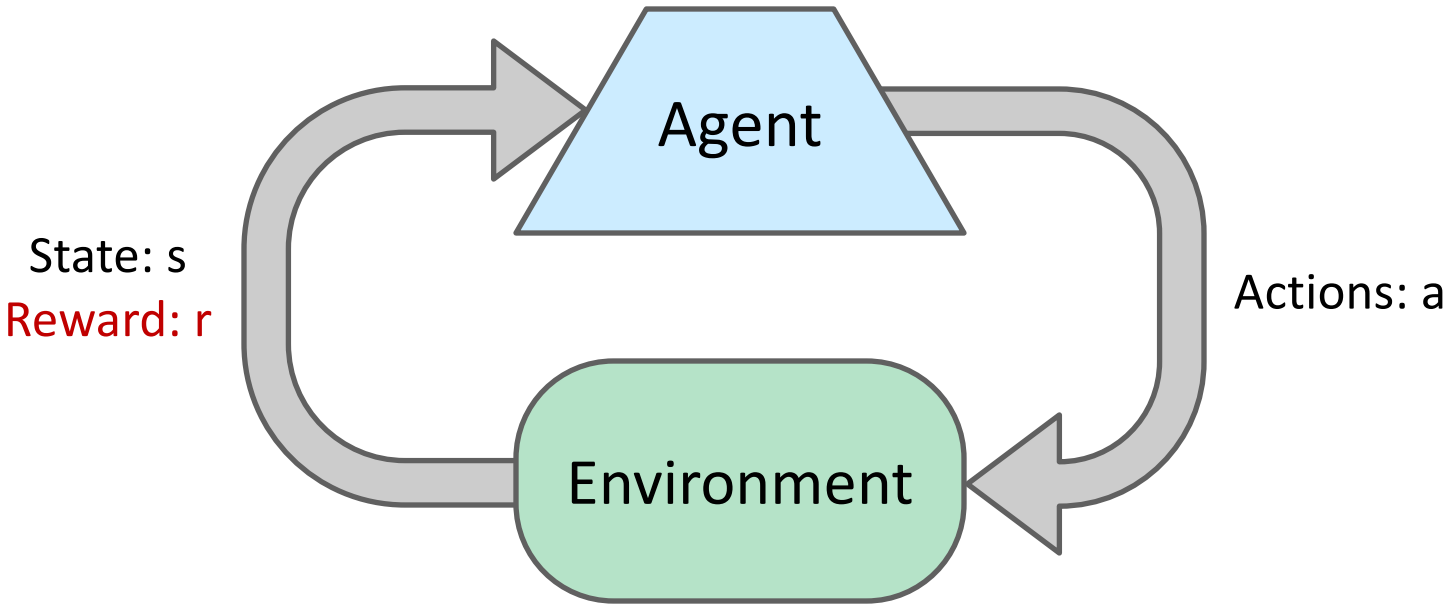
Reinforcement Learning

RL is about how to learn behaviors.



Reinforcement Learning

RL is like MDP but we don't know the model of the world (aka we don't know it's parameters). Instead we have to learn this through interacting with the world.



- Basic idea:
  - Receive feedback in the form of rewards
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to maximize expected rewards
  - All learning is based on observed samples of outcomes!

Example: Learning to Walk

Initial

This was trained exclusively using a simulation. This worked better in the simulation than in real world.

A Learning Trial

Now they run it in the real world and also train it using this new real world data

Main idea here: They trained a RL agent using simulation to get the dog to go as fast as possible. This was pretty good, but the real world is slightly different. It has different friction, etc. So they ran the agent in the real world and kept training. This improved the dog's speed.

After Learning [1K Trials]

Now after training it in the real world (and the simulated one) it goes faster.

[Kohl and Stone, ICRA 2004]

Example: Learning to Walk

Initial

[Kohl and Stone, ICRA 2004]

[Video: AIBO WALK – initial]

Example: Learning to Walk

Training

[Kohl and Stone, ICRA 2004]

[Video: AIBO WALK – training]

Example: Learning to Walk

Finished

[Kohl and Stone, ICRA 2004]

[Video: AIBO WALK – finished]

Example: Sidewinding

The snake learns to get on top of the platform, then shimmy over to the other side of the platform.

It's very difficult to build reliable simulators. For things like this, how do you build a reliable snake simulator? Well if you were to train a MDP on this simulator (and thus "Plan"), it might not work in the real world because the real world is slightly different than the simulation. The takeaway is that you can't always create a plan based on a simulation because the simulation is not exactly equal to the real world.

RL let's it learn on it's own, so avoids this issue of "planning" on a non-reliable simulation.

[Andrew Ng]

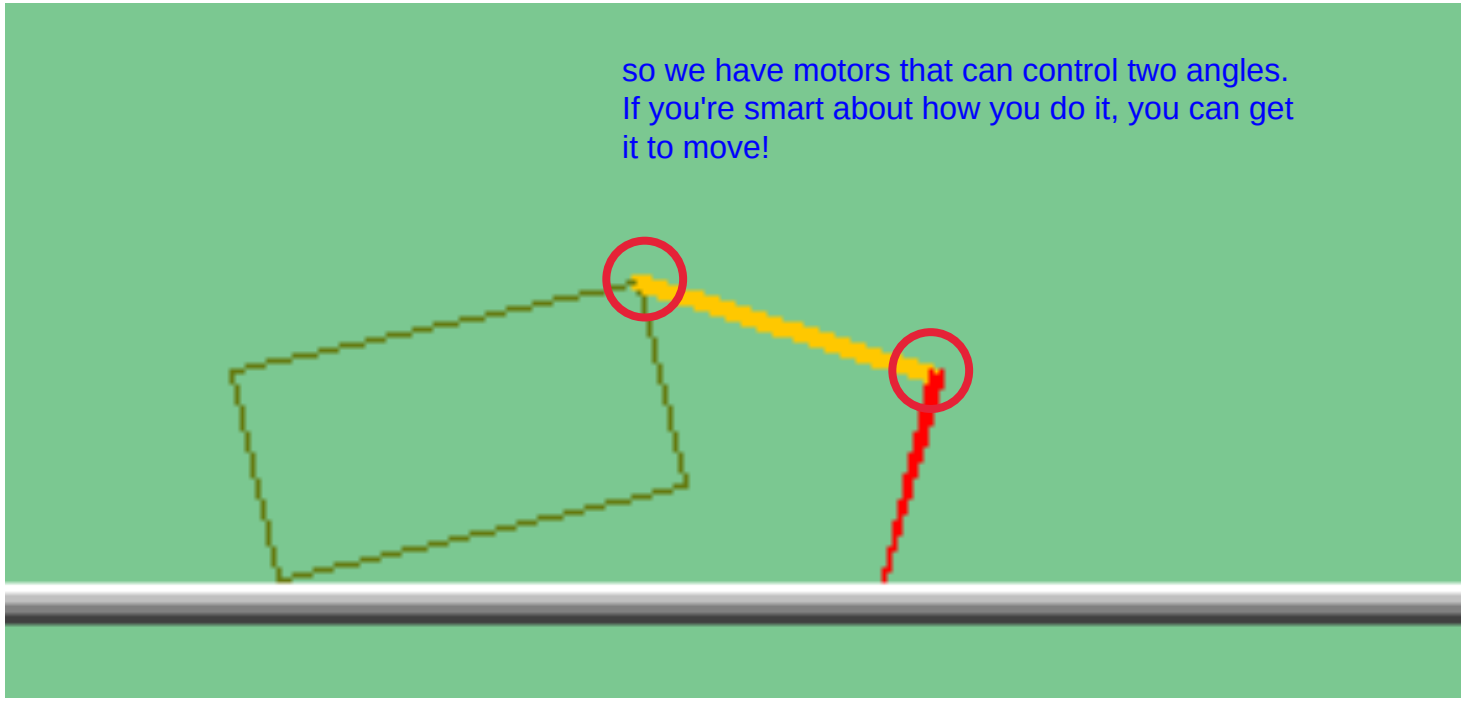
[Video: SNAKE – climbStep+sidewinding]

Example: Toddler Robot

[Tedrake, Zhang and Seung, 2005]

[Video: TODDLER – 40s]

The Crawler!



[Demo: Crawler Bot (L10D1)] [You, in Project 3]

Video of Demo Crawler Bot

Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) A
  - A model T(s,a,s')
  - A reward function R(s,a,s')
- Still looking for a policy $\pi(s)$

so now the picture becomes this. we don't have the transition function or rewards! We just have the states (and the possible actions)

- New twist: don't know T or R

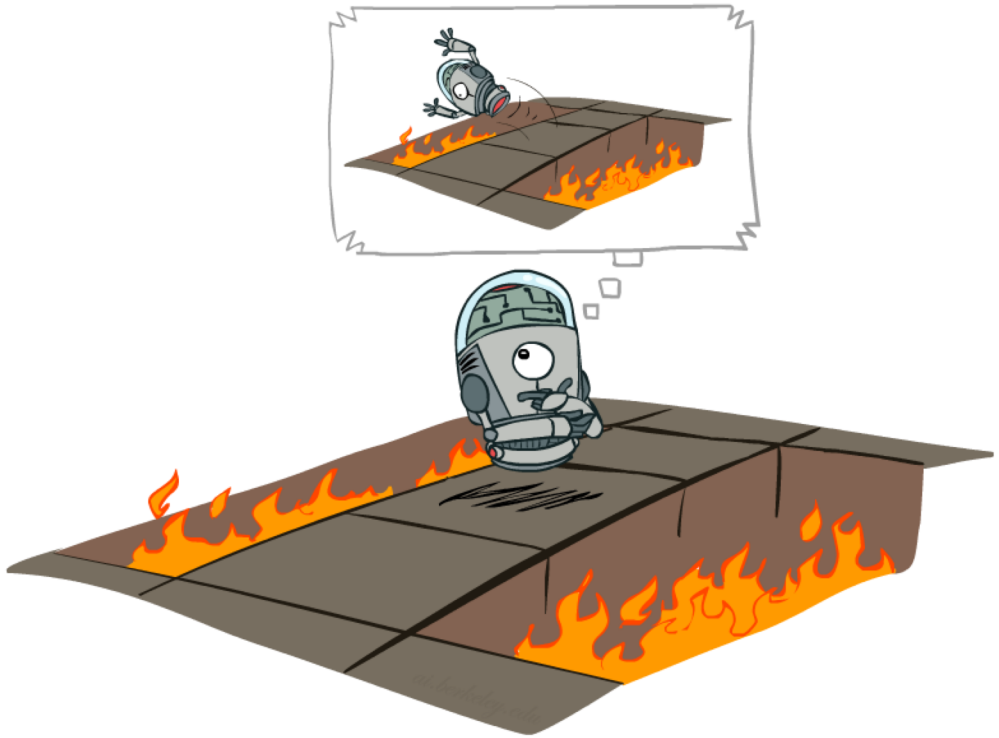
In Policy Iteration and Value Iteration, we need both T and R.

  - I.e. we don't know which states are good or what the actions do
  - Must actually try out actions and states to learn

We need to experiment to approximate T and R!

For most AI agents a person is deciding the reward function. For example, we would decide how many points we get for going Fast and Slow. You need to design it to correctly to get it to do what you intended it to do.

Offline (MDPs) vs. Online (RL)



Offline Solution

Value Iteration & Policy Iteration when we have the full MDP available. We plan in advance

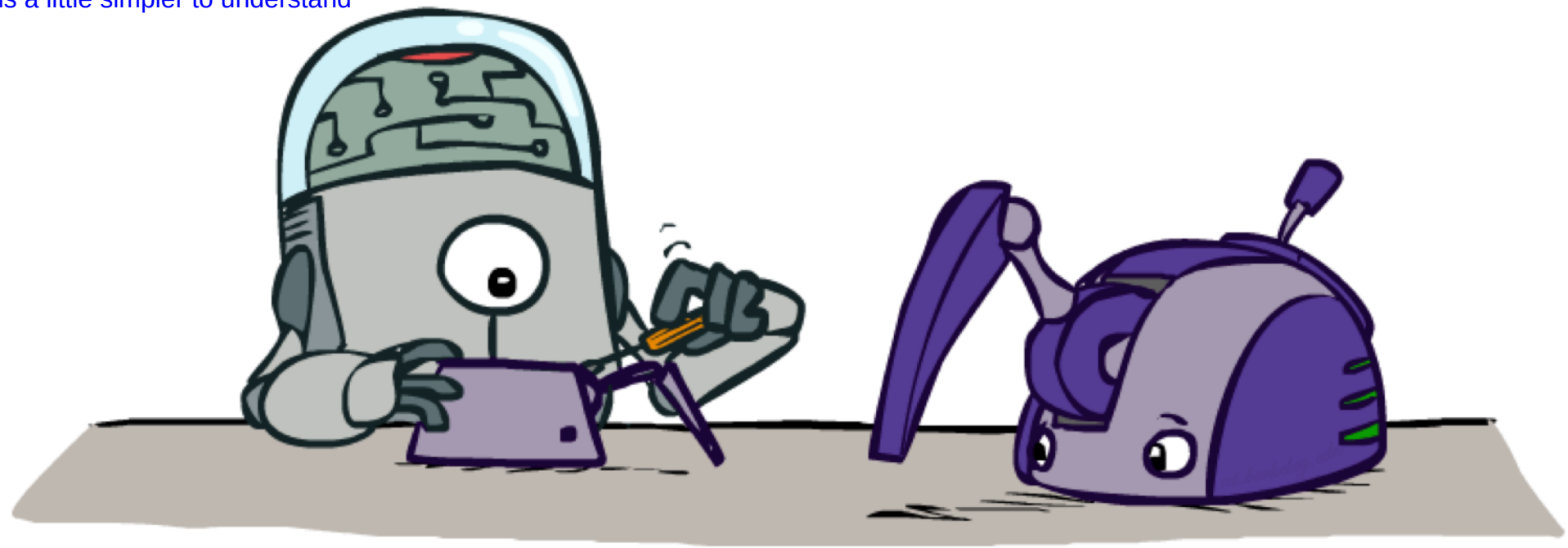
Online Learning

We have to learn a policy by trying things out. We like to do this in simulation because it's expensive in the real world. In RL, we don't know the fire pit is bad. We only know this after trying it, then getting a reward.

Model-Based Learning

Two approaches in RL:
1) Model-Based
2) Model-Free

Both are good approaches. One is not better than the other. Model-based is a little simpler to understand



Model-Based Learning

- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

- Step 1: Learn empirical MDP model
  - Count outcomes s' for each s, a
  - Normalize to give an estimate of $\hat{T}(s, a, s')$
  - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s')

- Step 2: Solve the learned MDP
  - For example, use value iteration, as before

This won't be the true MDP, but will be an approximate MDP.

Then often times after you've learned the MDP and found the "optimal policy" based on that, you'll then test it in the real world, gain more data/experience. If your reward in the real world is what you expected, then your simulator is probably pretty good and you might be done. But if it doesn't work as well that means you're getting new data that can inform you how the world works different from the simulator thought. Then you use that to improve your MDP representation and your policy.

When you were in state s and took action a, you landed in s1' 75% of the time, s2' 20% of the time s3' 5% of the time and 0% in all other states.

You can do the same for the reward.
Then you can build a table to find out the reward for a particular transition

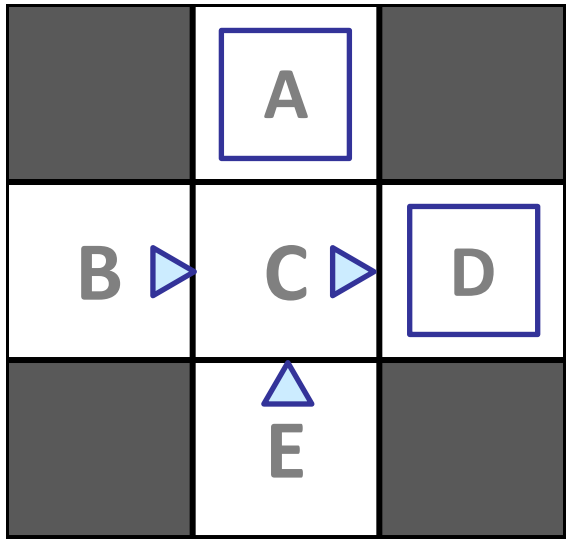
You need to do this for every starting state and every action!

we're finding our transition and reward functions using the frequencies we observe

Example: Model-Based Learning

3/4 times we do (C, east) we end up in D, and 1/4 times we end up in A

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

After we get this model we can run Value or Policy Iteration to get an "optimal" policy based on the model that we just learned..

Learned Model

in all times we did (B,east) we got to C.

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
...

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
...

Model based: You sample people's age. Then for each age, you calculate the probability by dividing the number of times that age came up by the total number of people you sampled. Then you find the expected age by summing together the product of age, and the probability of that age.

Example: Expected Age

On the left there is a weighting by probability, and on the right there is not. On the right we drew samples, asked their age. Their probability is already reflected in how many times they are selected.

Model free: You sample people's age. Then you add up all the ages you sampled, and find the average. This is what you'd probably actually do for finding expected age.

Goal: Compute expected age of cs188 students

Known P(A)

$$E[A] = \sum_a P(a) \cdot a \qquad = 0.35 \times 20 + \ldots$$

Without P(A), instead collect samples $[a_1, a_2, \ldots a_N]$

Unknown P(A): "Model Based"

estimated probability

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$

$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

Why does this work? Because eventually you learn the right model.

Unknown P(A): "Model Free"

$$E[A] \approx \frac{1}{N} \sum_i a_i$$

Why does this work? Because samples appear with the right frequencies.

i is a student from 0 to N-1

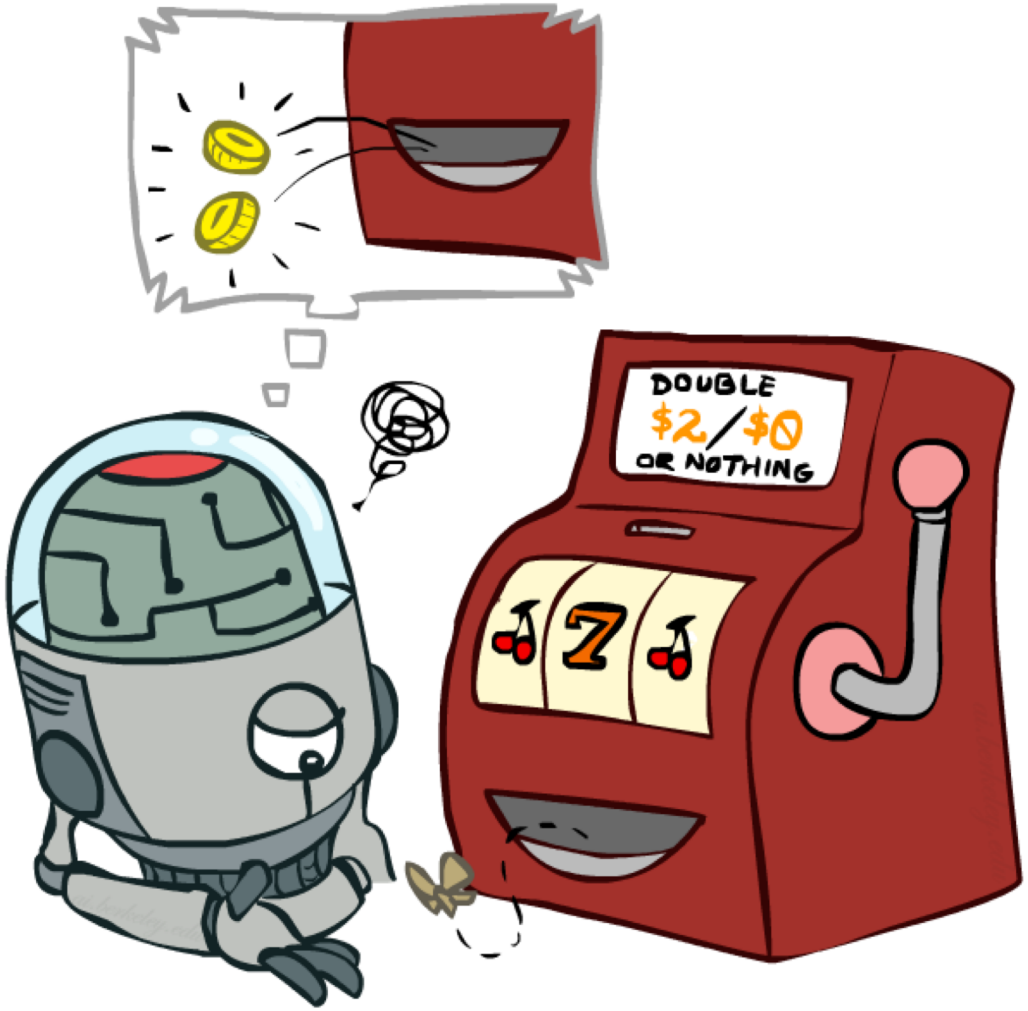
both methods work

Model-Free Learning

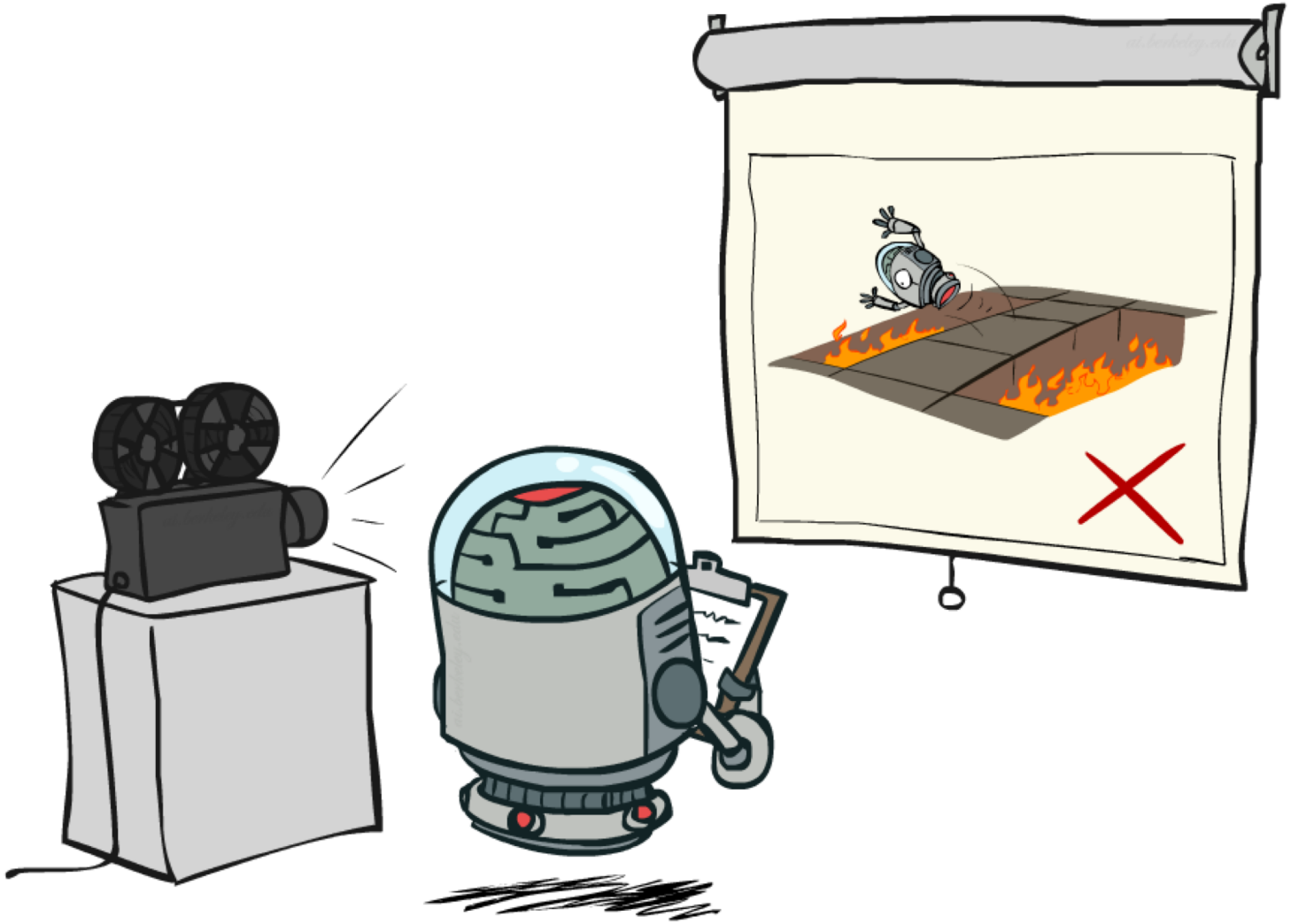
Two types:

1) Passive RL: We just try to estimate quantities/values we care about by watching what's happening in the real world. We don't care about acting in the world and we don't get to choose actions.

2) Active RL: we're also worried about how you collect that data



Passive Reinforcement Learning



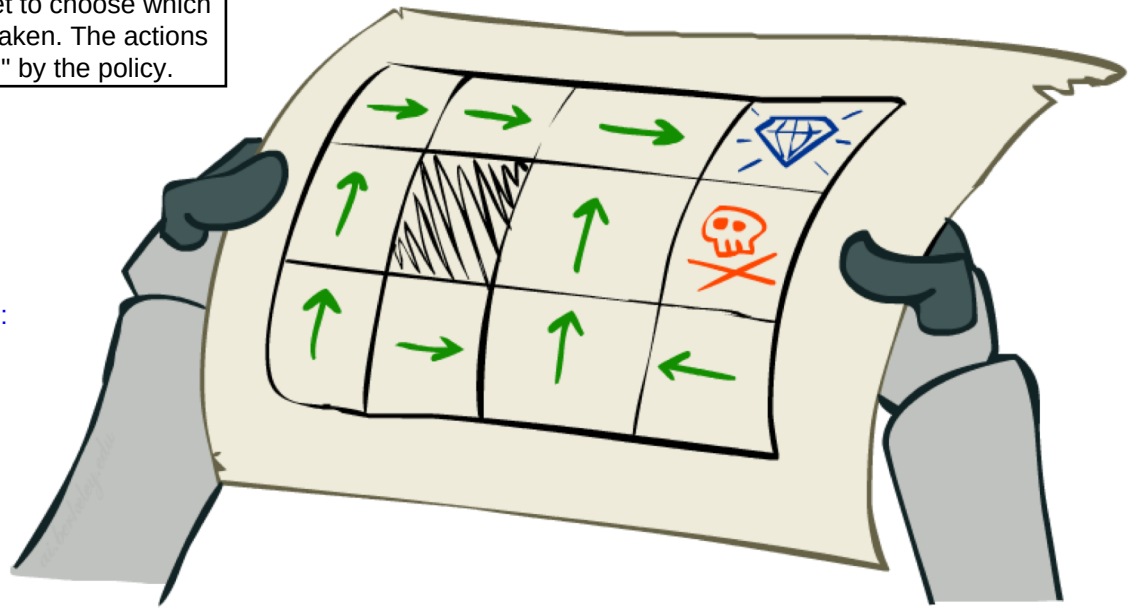
Passive Reinforcement Learning

- Simplified task: policy evaluation
  - Input: a fixed policy $\pi(s)$
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - Goal: learn the state values

we observe this policy in action, but we don't make any actions or get to choose which actions are taken. The actions are "decided" by the policy.

Two types of policy evaluation: Direct and Indirect Evaluation

We get to evaluate the quality of the policy (using the value or expected reward achieved)



- In this case:
  - Learner is "along for the ride"
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - This is NOT offline planning! You actually take actions in the world.

Direct Evaluation

- Goal: Compute values for each state under π
- Idea: Average together observed sample values
  - Act according to π
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples
- This is called direct evaluation

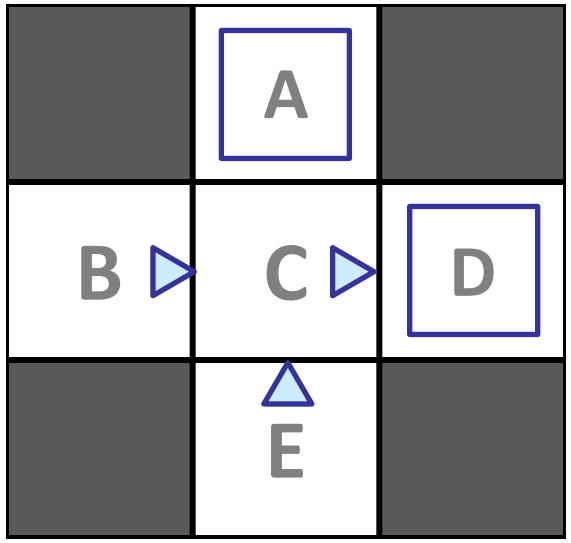
Example: Direct Evaluation

We were in B twice (ep 1 & 2)

In both cases, we got -1, -1 + 10 = +8

we only visited A once and we got -10, then the episode was over so we never got any more rewards

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

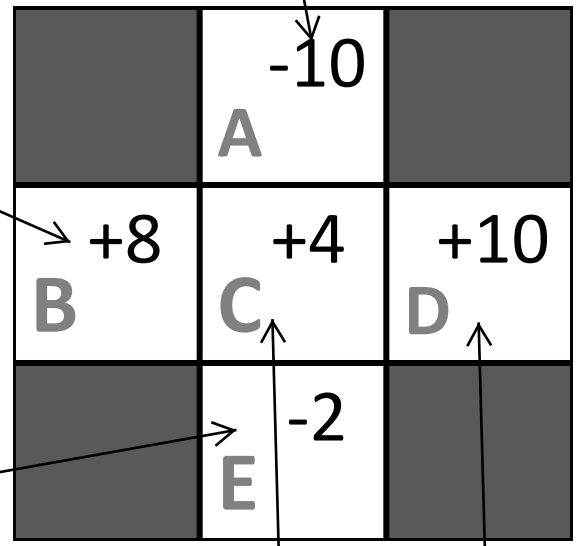
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



We experience E two times:
-1 + (-1) + 10 = 8
-1 + (-1) + (-10) = - 12
(8-12)/2 = -2

We experience C 4 times
-1 + 10 = 9
-1 + 10 = 9
-1 + 10 = 9
-1 + (-10) = -11
(9+9+9-11)/ 4 = 4

We were in D 3 times, and every time we acted we got +10 then the episode was over, so on average we got +10

Problems with Direct Evaluation

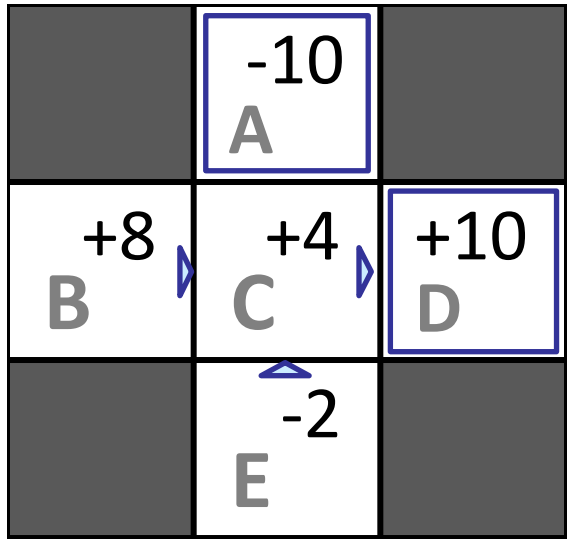
- What's good about direct evaluation?
  - It's easy to understand
  - It doesn't require any knowledge of T, R
  - It eventually computes the correct average values, using just sample transitions

- What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

never really considers correlations between states, just looking at averages.

Not super precise unless you calculate infinite data.
But we don't want to have to learn after collecting infinite data!

Output Values



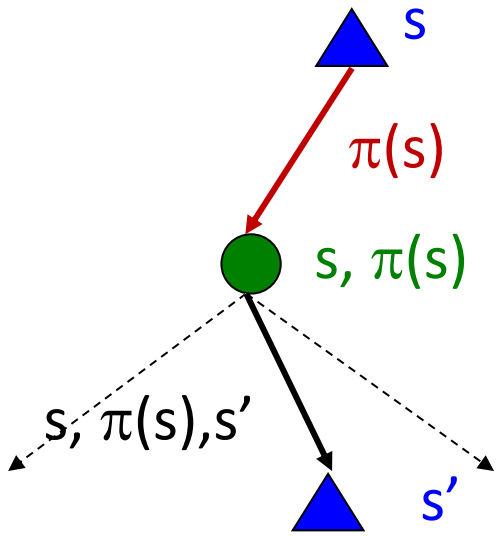
If B and E both go to C under this policy, how can their values be different?

Why Not Use Policy Evaluation?

- Simplified Bellman updates calculate V for a fixed policy:
  - Each round, replace V with a one-step-look-ahead layer over V

$$V_0^{\pi}(s) = 0$$

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$



  - This approach fully exploited the connections between the states
  - Unfortunately, we need T and R to do it!

- Key question: how can we do this update to V without knowing T and R?
  - In other words, how to we take a weighted average without knowing the weights?

This is the question we'll need to answer!

So we can use an average of samples instead of a weighted expectation!

Sample-Based Policy Evaluation?

- We want to improve our estimate of V by computing these averages:

$$V^{\pi}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{k}(s')]$$

we can't rewind time, making this strategy difficult

- Idea: Take samples of outcomes s' (by doing the action!) and average

starting from state s, let's use our policy from there and see what happens

then we can average them

the tricky thing is that you often cannot make the agent start an a specific state, then run it, then reset to that same state.

In our Grid World, this might be easy, but in more realistic environments, we might not.

so what do we do??? ...see next slide.

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V^{\pi}_{k}(s'_1)$$

$$sample_2 = R(s, \pi(s), s'_2) + \gamma V^{\pi}_{k}(s'_2)$$

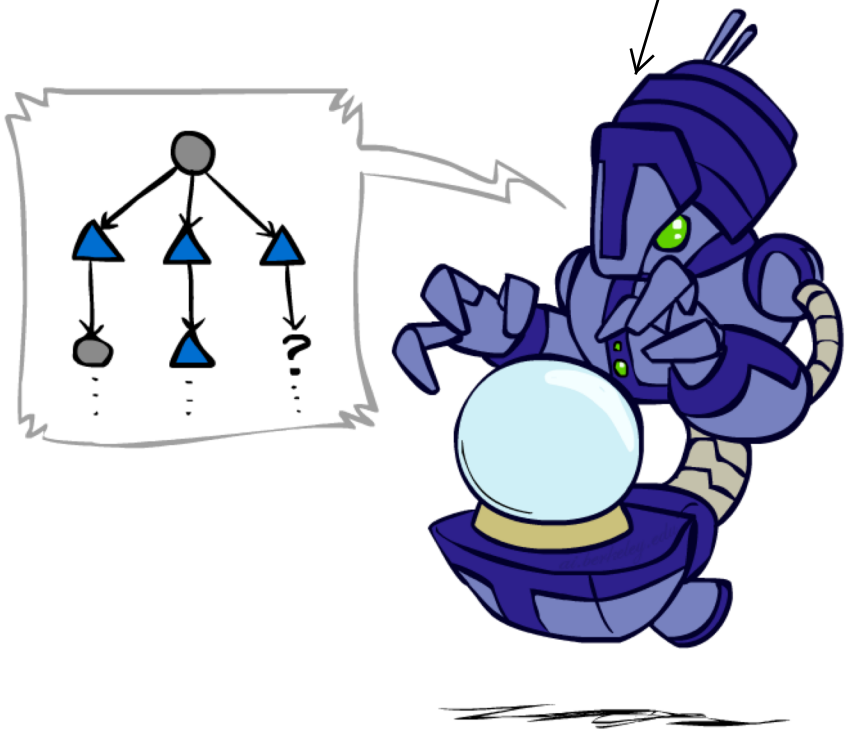
$$\dots$$

$$sample_n = R(s, \pi(s), s'_n) + \gamma V^{\pi}_{k}(s'_n)$$

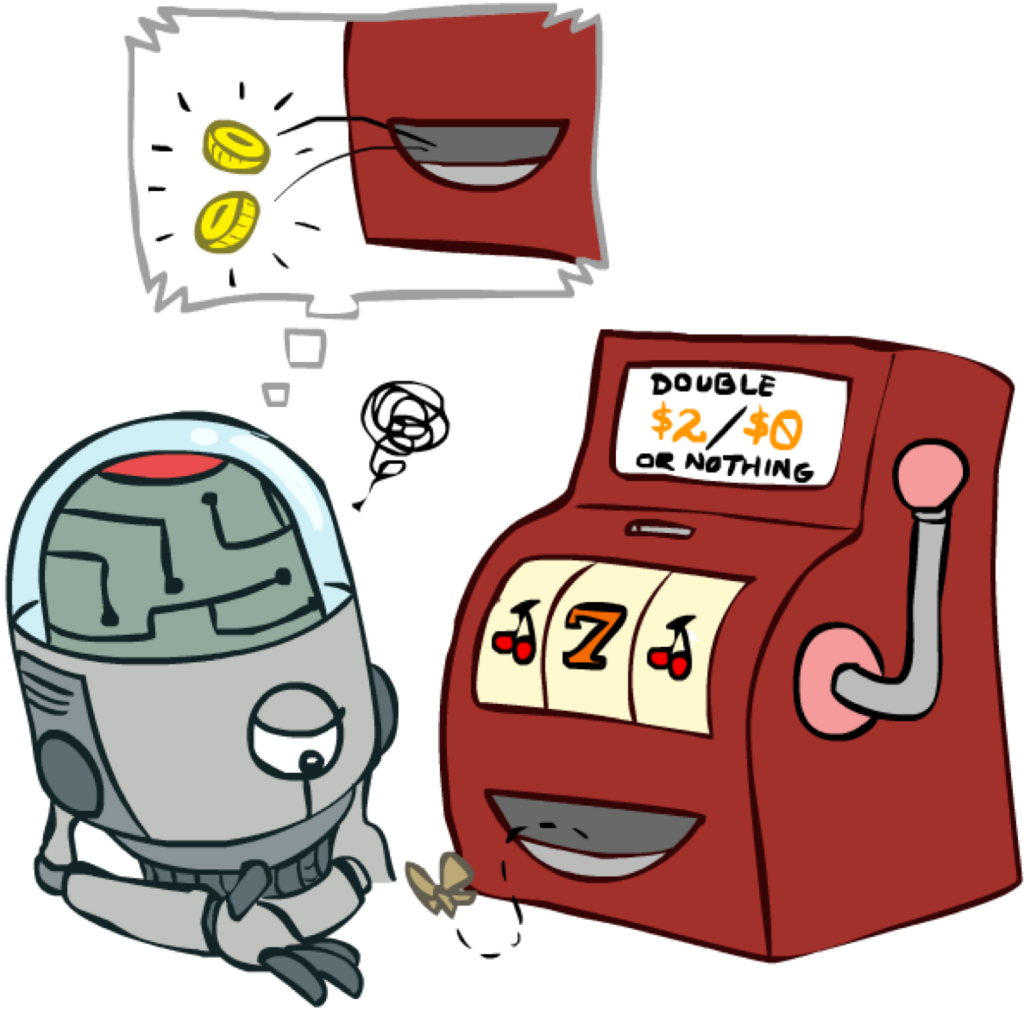
$$V^{\pi}_{k+1}(s) \leftarrow \frac{1}{n} \sum_{i} sample_i$$

let's assume we have this, because we have Vk0 = 0

remember the model-free age estimation used the sample values like this, while model-based used the explicit probabilities!

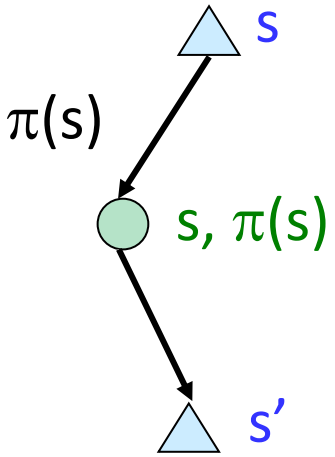


Temporal Difference Learning



Temporal Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

in the beginning our samples are very noisy/ random because V(s') is not correct. As time goes by, V(s') gets more accurate, so our samples get more accurate

how do you take an average of only one sample? Well we can assume we have a running average V(s), then we keep that mostly (say alpha is 0.1), and we correct it using the new sample value! We know V(s) isn't exactly optimal/true, but that's okay. We are improving it as we go.

We know that our sample V(s) is noisy because it's only from one sample. That's okay.

Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

same has the sample in the previous slide, except now we have only one sample per state s, instead of several

Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

so we adjust our best estimate, V(s), based on our recent sample.

Exponential Moving Average

- Exponential moving average
  - The running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$

this is the running average at n-1

  - Makes recent samples more important:

it's not the same as computing the actual average, it's a little different

$$\bar{x}_n = \frac{x_n + (1-\alpha) \cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

  - Forgets about the past (distant past values were wrong anyway)

- Decreasing learning rate (alpha) can give converging averages

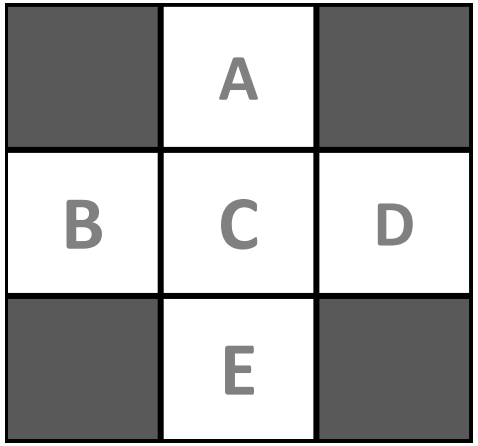
Example: Temporal Difference Learning

V(s) only changes for the state you LEFT, because the state you left is the one where you get a new estimate of the value of that state

V(s) = r +gamma * V(s')
originally V(D) = 8
sample V(C) = -2 + 1 * 8= 6

update
V(C) <-- (1-0.5)*0 + 0.5(6)
V(C) <-- 3

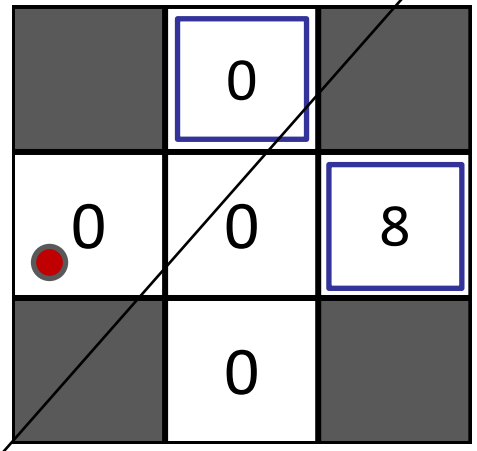
States

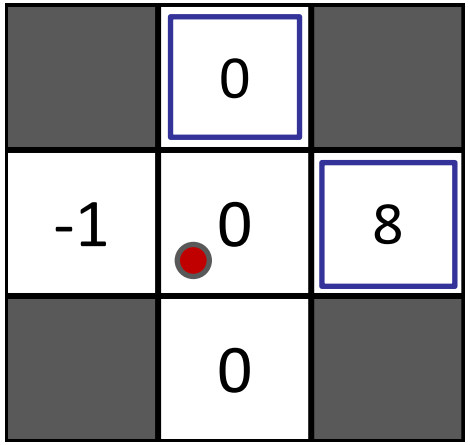


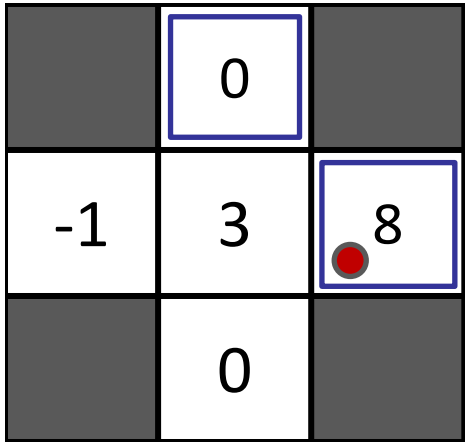
Observed Transitions

B, east, C, -2

C, east, D, -2







Assume: $\gamma = 1$, $\alpha = 1/2$

V(s) = r + gamma * V(s')
originally V(C) = 0
sample V(B) = -2 + 1 * 0 = -2

update
V(B) <-- (1 - 0.5)*0 + 0.5(-2)
V(B) <-- (-1)

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

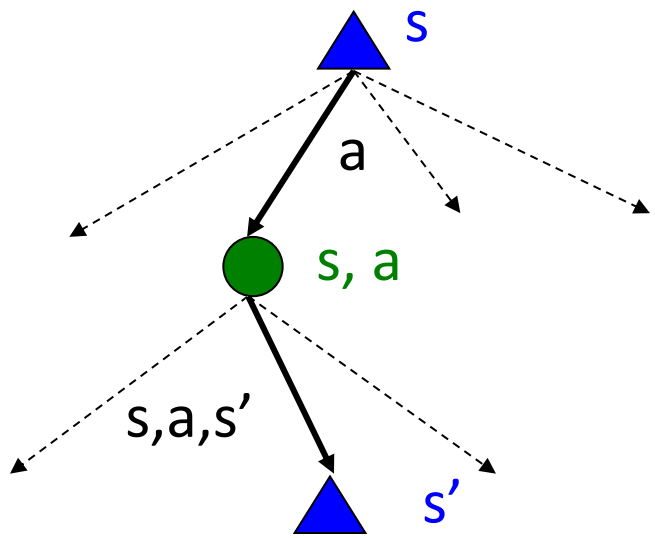
In the MDP lectures, we say Policy Iteration. In Policy Iteration, you need to do Policy Evaluation. But then you need to improve your policy doing policy updates. To improve you policy, you need to have a model... so we're still kind of stuck because we don't know how to update our policy.

Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, if we want to turn values into a (new) policy, we're sunk:

$$\pi(s) = \arg\max_a Q(s,a)$$

$$Q(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right]$$

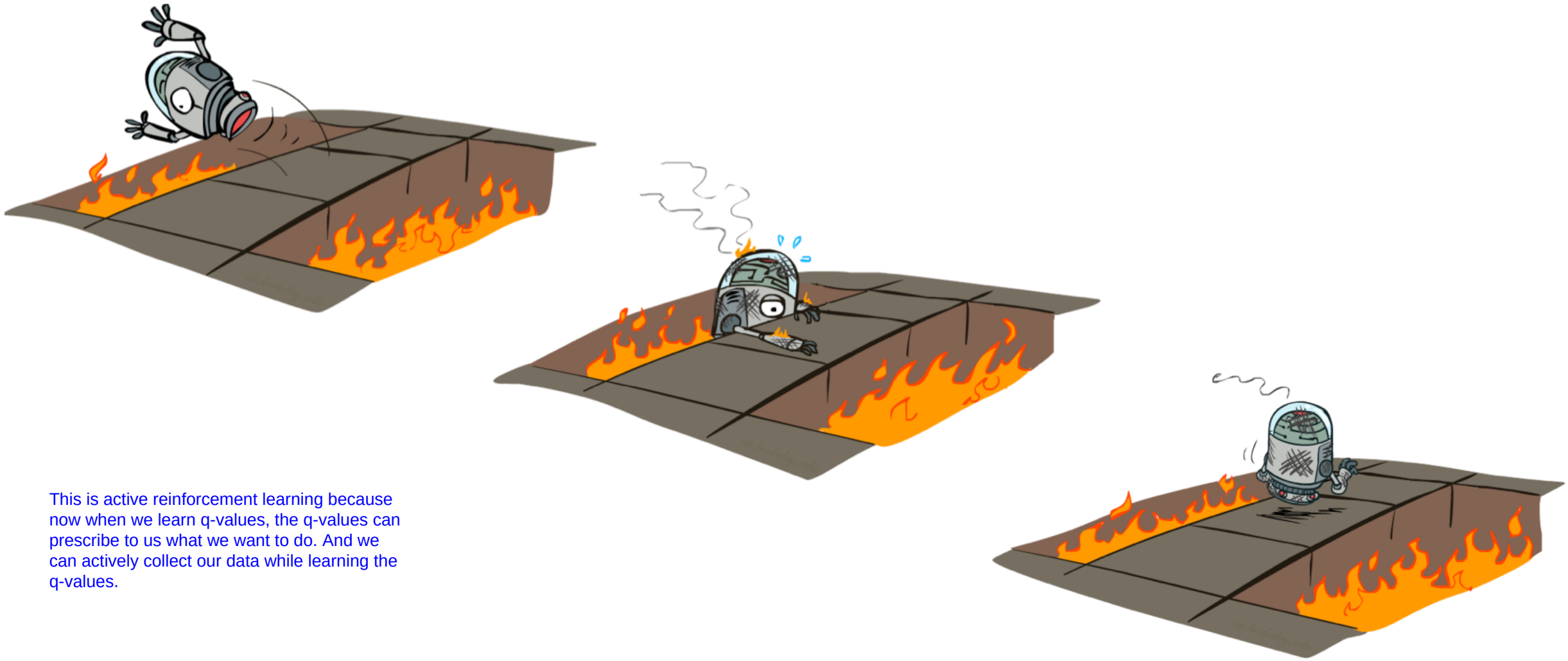


- Idea: learn Q-values, not values
- Makes action selection model-free too!

The key idea for the remainder of this lecture is that maybe we've been doing it all wrong in some sense. Because instead of learning the values of the regular states V(s), why don't we learn the values of the q_states? Then we can easily derive a policy from this by selecting the action with the highest q-value. Thus by learning and improving our q-values, we automatically improve our policy. So let's now switch to learning q-values.

Why didn't we start off learning q-values? The math is simpler for learning V-values and it is still relevant to using V-values. But now by using V-values, we can improve our policy.

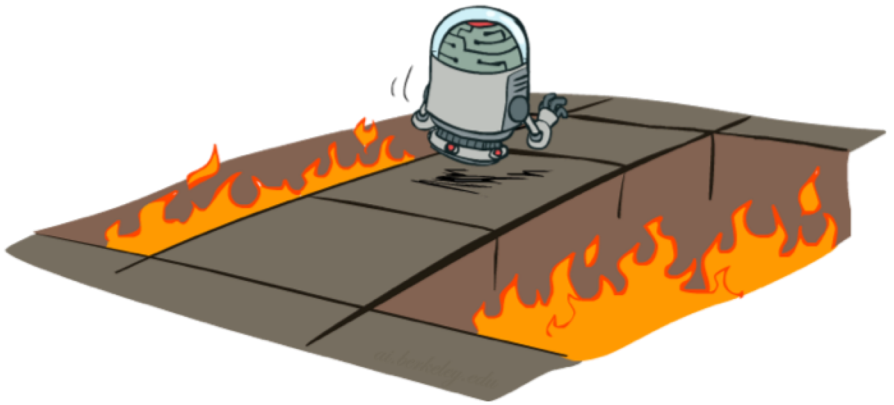
Active Reinforcement Learning



This is active reinforcement learning because now when we learn q-values, the q-values can prescribe to us what we want to do. And we can actively collect our data while learning the q-values.

Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values



- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

Detour: Q-Value Iteration

- Value iteration: find successive (depth-limited) values
  - Start with $V_0(s) = 0$, which we know is right
  - Given V_k, calculate the depth k+1 values for all states:

you can't do sample-based estimation with the max in front. It doesn't really make sense to take the max over a single sample

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$, which we know is right
  - Given Q_k, calculate the depth k+1 q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

here we can just samples and average those

Q-Learning

- Q-Learning: sample-based Q-value iteration

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

- Learn Q(s,a) values as you go

we'll make it sample based so that we don't need the model and we can get rid of the transition function

  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

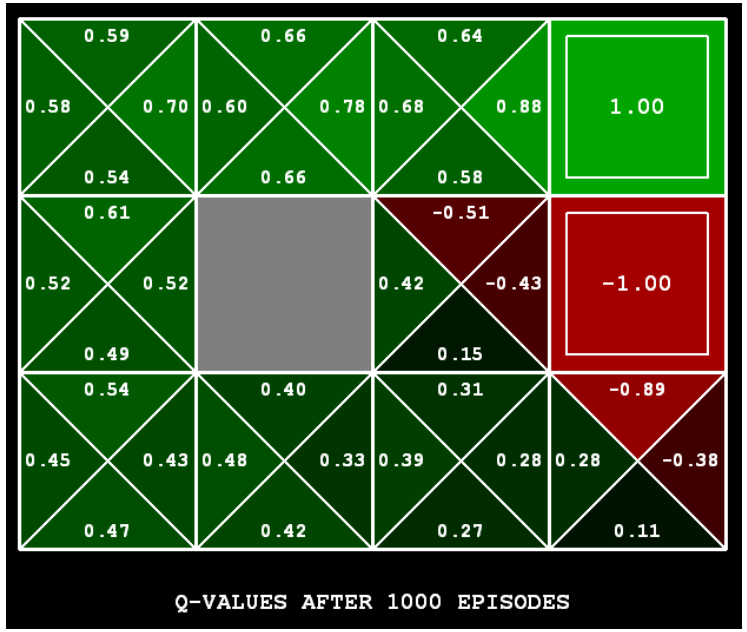
this isn't an accurate estimate at the start, but will improve over time

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

as we continue to update our q-value and take more samples, we get more accurate q-values.



[Demo: Q-learning – gridworld (L10D2)]
[Demo: Q-learning – crawler (L10D3)]

Video of Demo Q-Learning -- Gridworld

Video of Demo Q-Learning -- Crawler

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!

only the good stuff propagates through the "max"

so for q-learning, you can use any policy as long as you visit each state often enough

- This is called off-policy learning

we aren't learning the value of the policy that we're executing, we're learning the optimal policy

- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - … but not decrease it too quickly
  - Basically, in the limit, it doesn't matter how you select actions (!)