CS 4100 // artificial intelligence

instructor: byron wallace

Reinforcement learning

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

A note on the early feedback

- Thank you to everyone who took the time to complete the survey!
- Overall, the feedback was reasonably positive (selection bias?)
- But, I do want to be as responsive as possible, so, a few adjustments…

On the programming HWs

A few folks noted that PacMan HWs felt detached from course content

- These *are* meant to be complementary to the lectures by having you actually work with the models; I realize this is a lot of programming, but.. This *is* an upper-level CS class!

- Some of you *loved* the HWs!

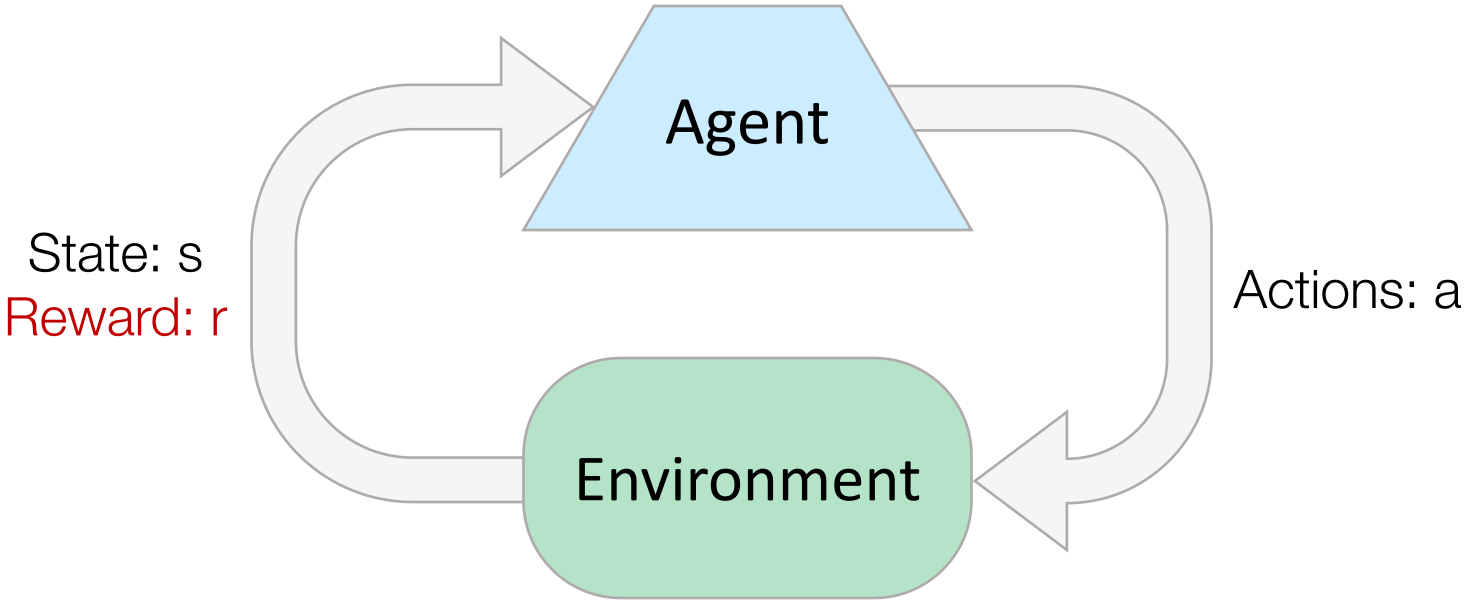
- Still, there was a demand for perhaps more written HWs. Thus: **I am going to scale back the programming components a bit and scale up the written components**. Don't worry, PacMan is not going away entirely!

Other miscalleany

- I will try to post slides immediately before class, so that you may follow along and take notes (this was a request)

- I will also try to post solutions to in-class exercises – another request – and have done so for the value iteration exercise already

Reinforcement learning



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!

Mario

https://www.youtube.com/watch?v=N3L-IZ1Xlfc&list=PL5nBAYUyJTrM48dViibyi68urttMlUv7e&index=19

Reinforcement learning

- *Learn* to map situations to actions
- The fundamental trade-off: *exploration* (what don't we know about our environment?) vs. *exploitation* (how to exploit what we do know)

Reinforcement learning

Still assume an underlying Markov decision process (MDP) – we just don't know the parameters!:

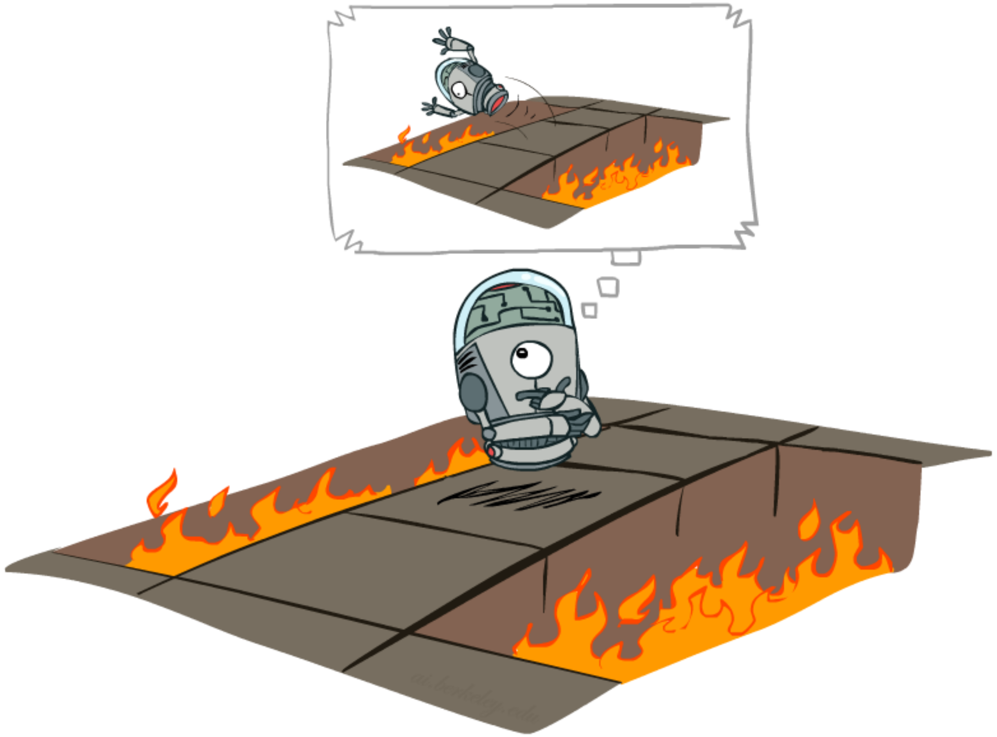
- A set of states $s \in S$
- A set of actions (per state) A
- A model T(s,a,s')
- A reward function R(s,a,s')

And we're still looking for a policy $\pi(s)$

New twist: **don't know T or R**

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

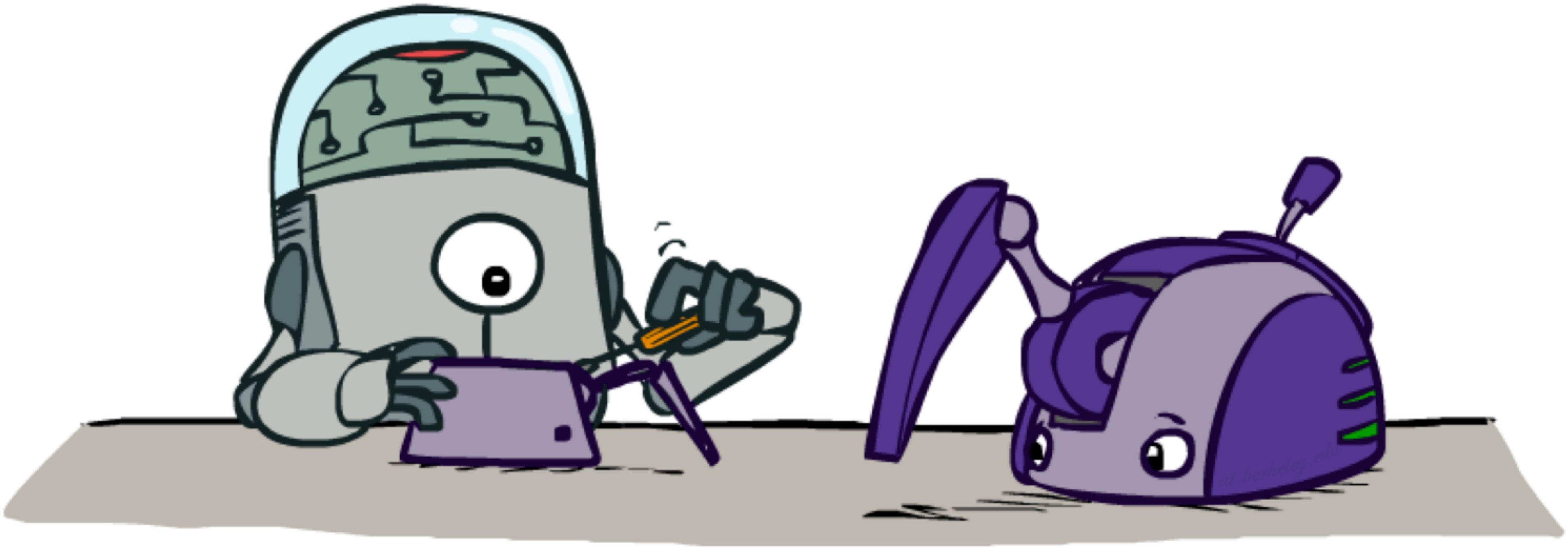
Offline (MDPs) vs. Online (RL)



Offline Solution

Online Learning

Model-based learning



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

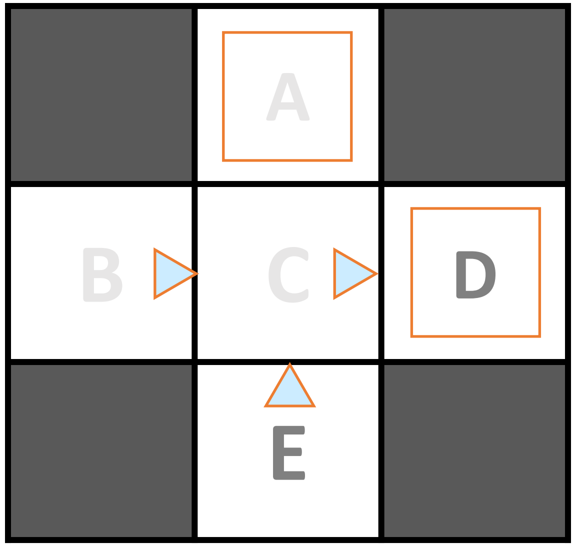
- Count outcomes s' for each s, a
- Normalize to give an estimate of $\widehat{T}(s, a, s')$
- Discover each $\widehat{R}(s, a, s')$ when we experience (s, a, s')

Step 2: Solve the learned MDP

- For example, use value iteration, as before

Example: model-based learning

Input policy π



Assume: γ = 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

Learned model

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

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
…

Example: expected age

Goal: Compute expected age of cs4100 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, … a_N]

Unknown P(A): "Model Based"

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

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

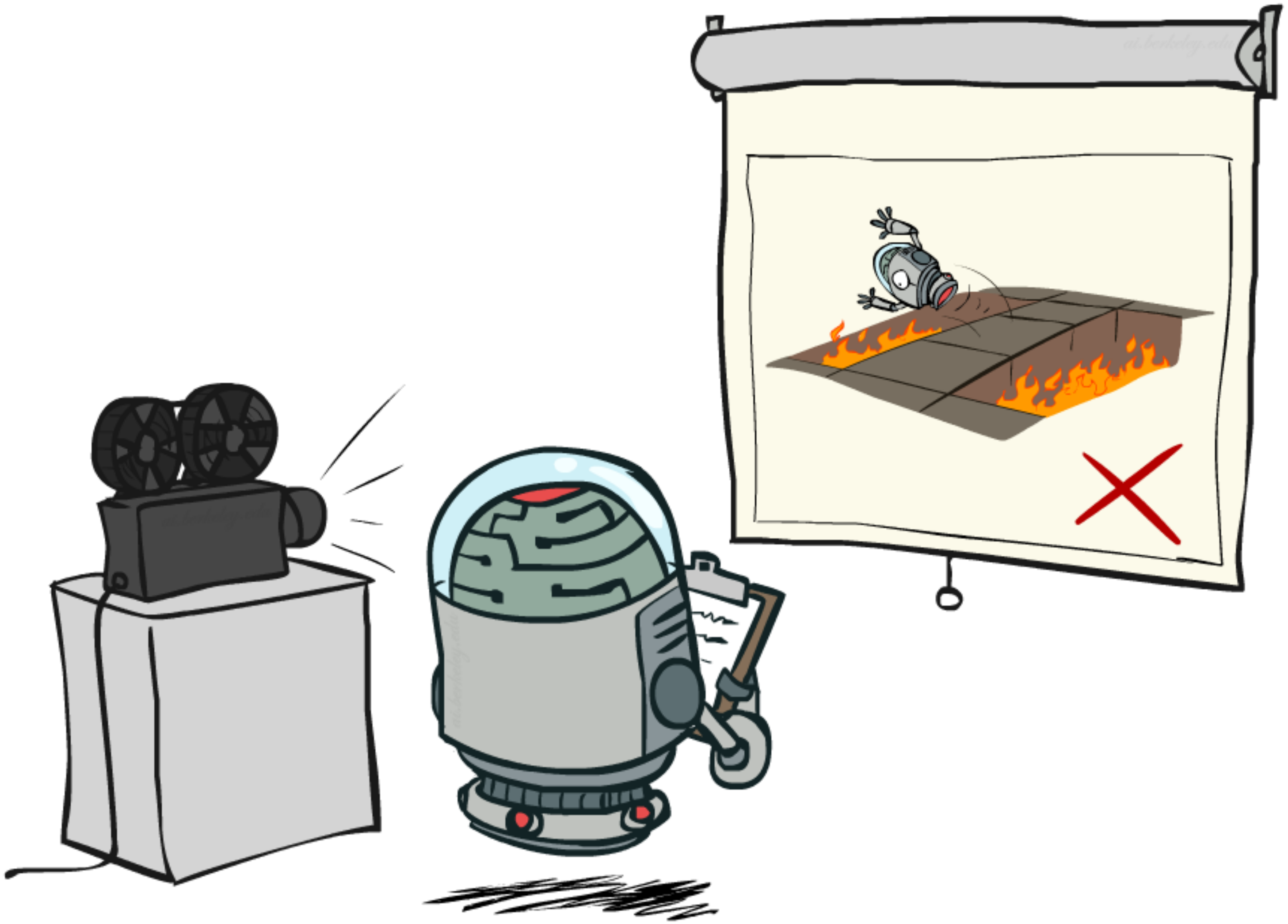
Why does this work? Because eventually you learn the right model.

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.

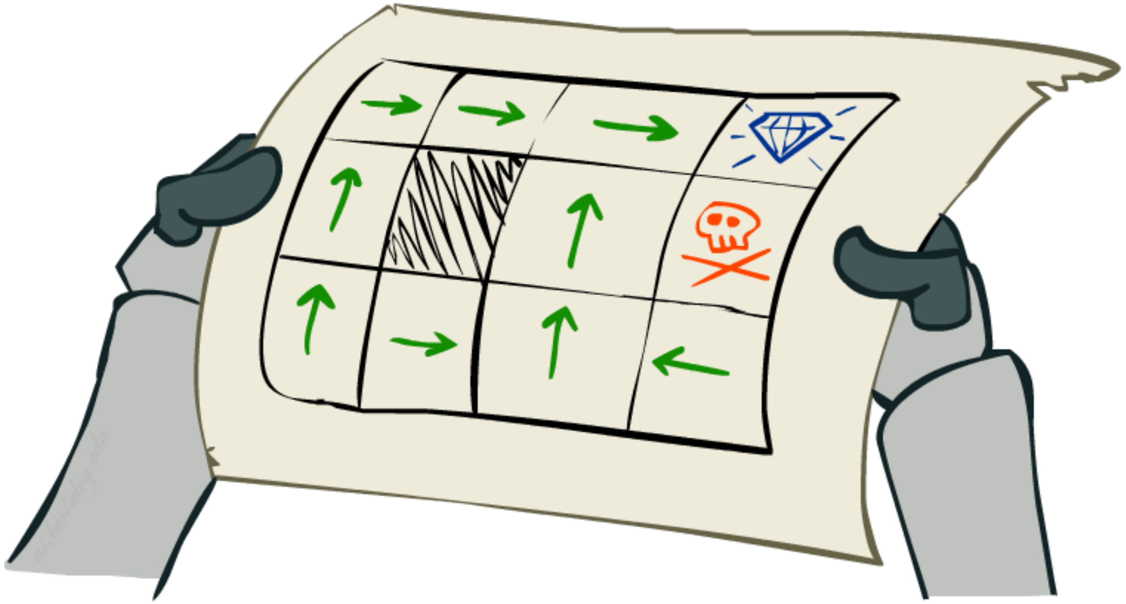
Passive reinforcement learning



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**



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

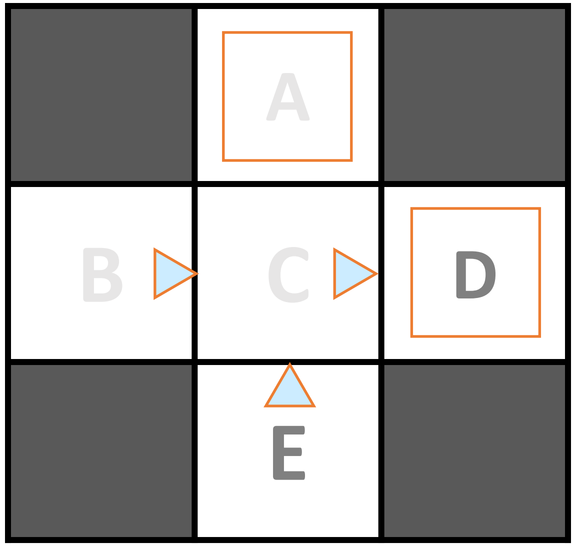
Idea: Average together observed sample values

- 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

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

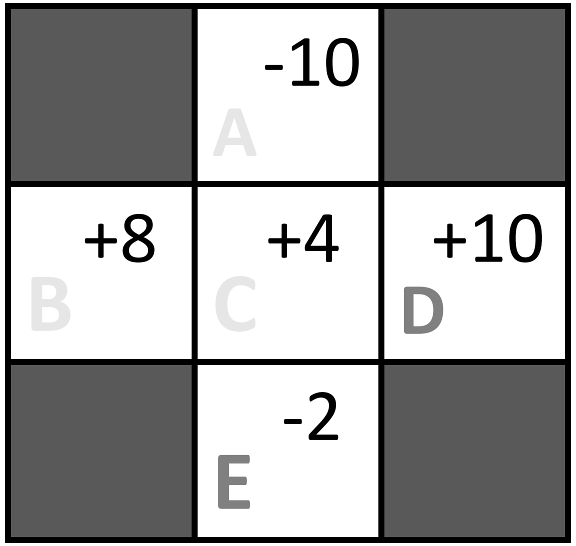
Episode 3

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



Direct evaluation: pros and cons

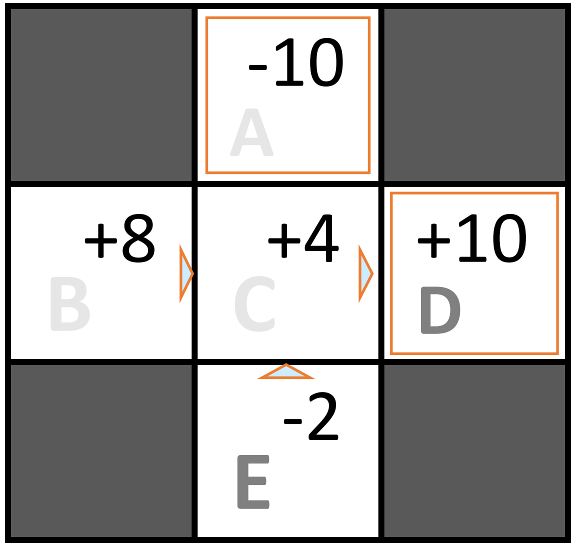
What's good about direct evaluation?

- 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

Output Values



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

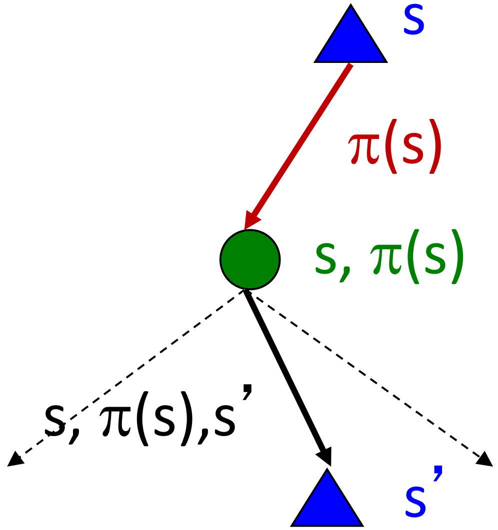
Why not use policy evaluation?

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?

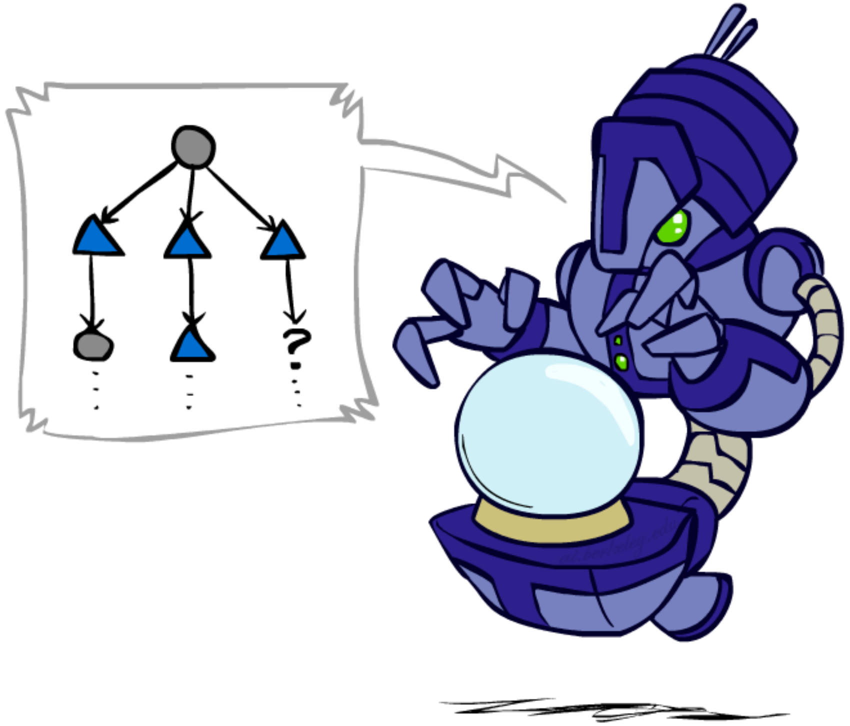
Sample-based policy evaluation?

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

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

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

$$sample_1 = R(s, \pi(s), s_1') + \gamma V_k^{\pi}(s_1')$$
$$sample_2 = R(s, \pi(s), s_2') + \gamma V_k^{\pi}(s_2')$$
$$\ldots$$
$$sample_n = R(s, \pi(s), s_n') + \gamma V_k^{\pi}(s_n')$$



$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n} \sum_i sample_i$$

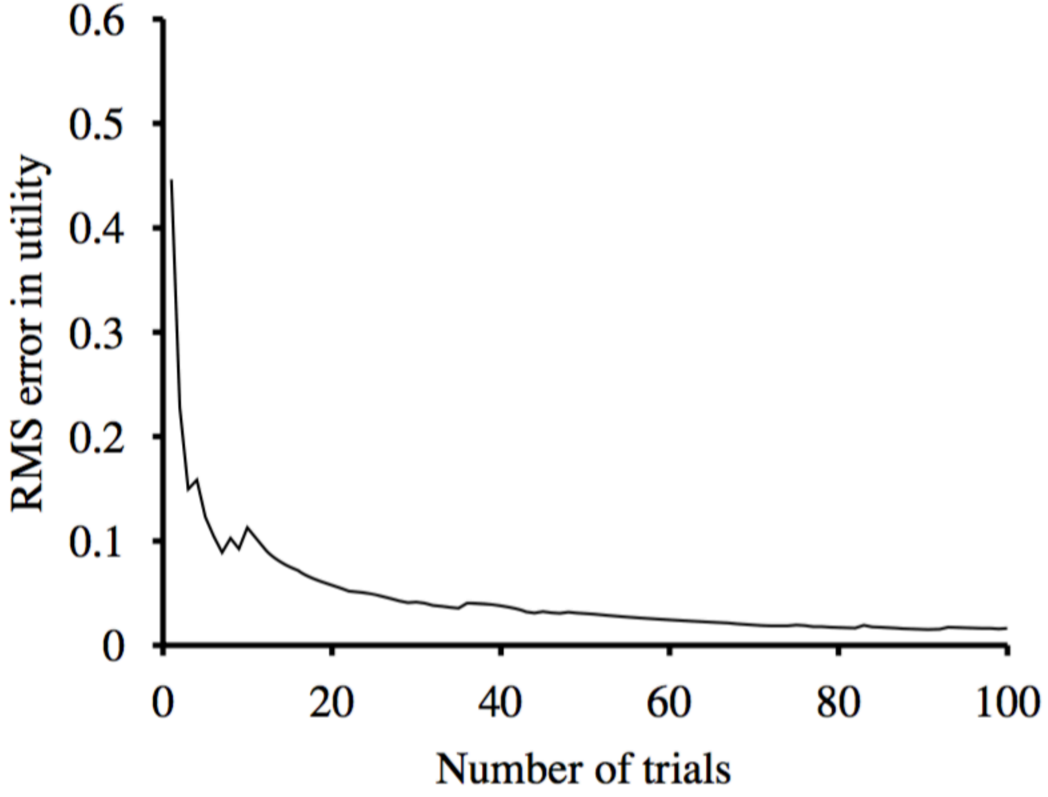
Adaptive dynamic programming (ADP)

Idea: exploit problem constraints between utilities of states by *learning* the transition model that connects them

$$V^{\pi}_{k+1}(s) \leftarrow \sum_{s'} \underbrace{T(s, \pi(s), s')}_{\text{Estimate these transition probabilities}} [\underbrace{R(s, \pi(s), s')}_{\text{Rewards will be observed}} + \gamma V^{\pi}_{k}(s')]$$

```
function PASSIVE-ADP-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s′ and reward signal r′
  persistent: π, a fixed policy
              mdp, an MDP with model P, rewards R, discount γ
              U, a table of utilities, initially empty
              N_sa, a table of frequencies for state–action pairs, initially zero
              N_s′|sa, a table of outcome frequencies given state–action pairs, initially zero
              s, a, the previous state and action, initially null

  if s′ is new then U[s′] ← r′; R[s′] ← r′
  if s is not null then
      increment N_sa[s, a] and N_s′|sa[s′, s, a]
      for each t such that N_s′|sa[t, s, a] is nonzero do
          P(t | s, a) ← N_s′|sa[t, s, a] / N_sa[s, a]
  U ← POLICY-EVALUATION(π, U, mdp)
  if s′.TERMINAL? then s, a ← null else s, a ← s′, π[s′]
  return a
```


0.6
0.5
0.4
0.3
0.2
0.1
0
RMS error in utility
0
20
40
60
80
100
Number of trials

(b)

ADP is one means of incorporating Bellman eq.

- In ADP we (continuously re-)estimate the transition probabilities then estimate policy value using one of the methods from last time
- There's another way: *Temporal Difference Learning (TDL)*
- Idea is to adjust utility estimates at each step to align with Bellman equations

ADP is one means of incorporating Bellman eq.

- In ADP we (continuously re-)estimate the transition probabilities then estimate policy value using one of the methods from last time
- There's another way: *Temporal Difference Learning (TDL)*
- Idea is to adjust utility estimates at each step to align with Bellman equations

$$U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$$

Sample-based (model-free) policy evaluation?

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

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

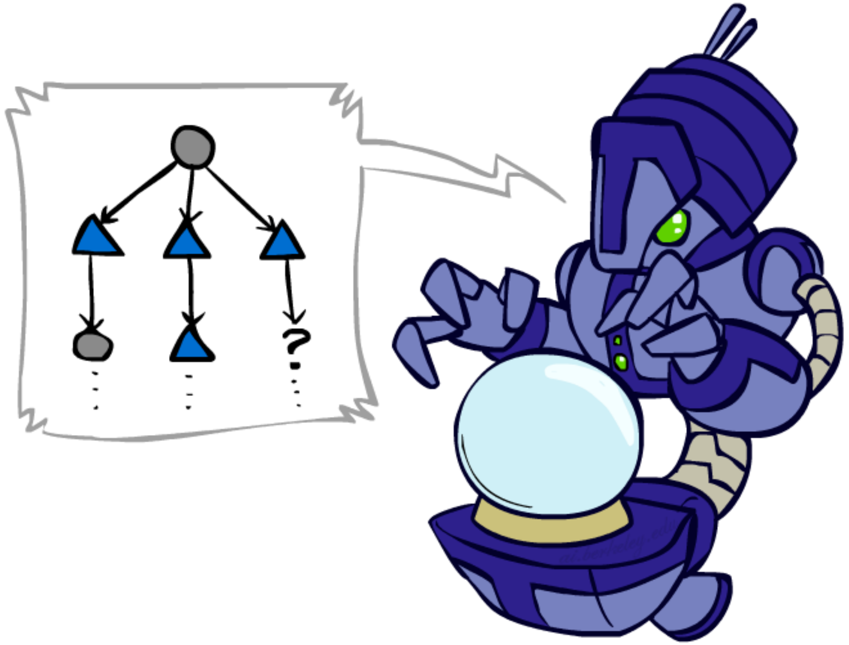
Idea: Take samples of outcomes s' (by doing the action!) and average

$$sample_1 = R(s, \pi(s), s_1') + \gamma V_k^{\pi}(s_1')$$

$$sample_2 = R(s, \pi(s), s_2') + \gamma V_k^{\pi}(s_2')$$

$$\ldots$$

$$sample_n = R(s, \pi(s), s_n') + \gamma V_k^{\pi}(s_n')$$

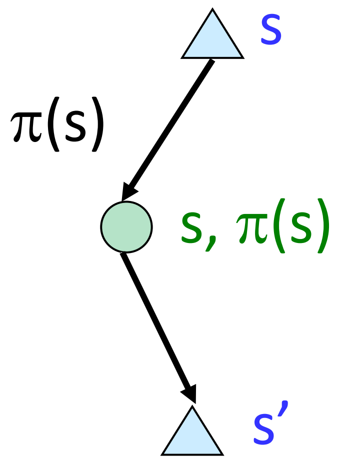


$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n} \sum_i sample_i$$

Temporal Difference Learning (model free!)

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

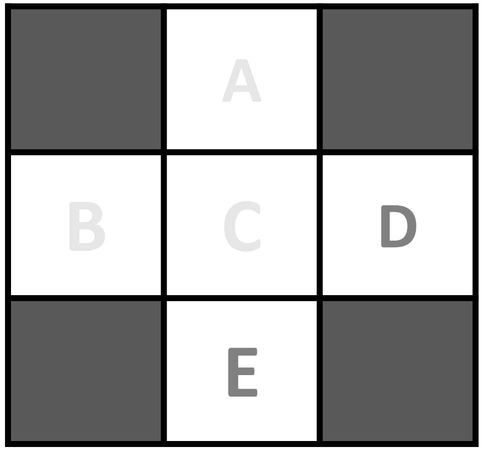
Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

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

Can rewrite as: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Example: temporal difference learning

States

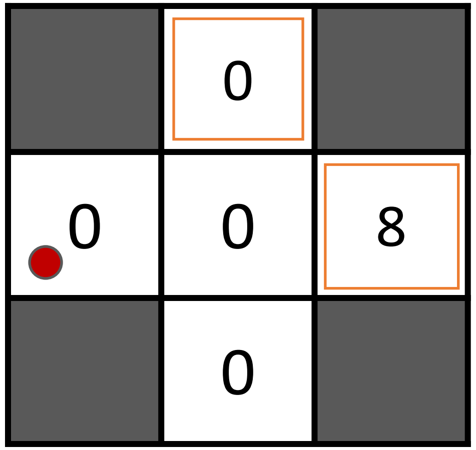


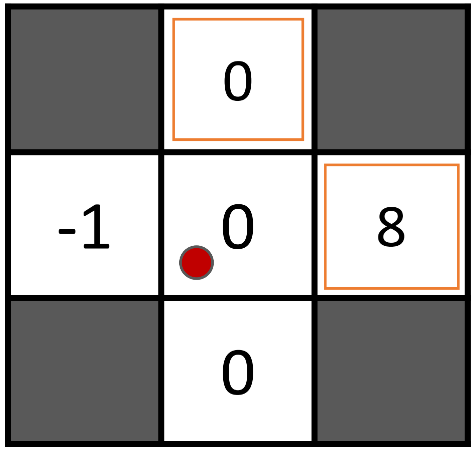
Assume: $\gamma = 1$, $\alpha = 1/2$

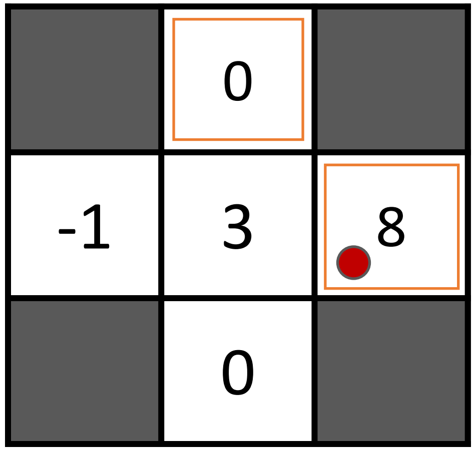
Observed Transitions

B, east, C, -2

C, east, D, -2







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

In class exercise on TDL

Exponential moving average

- The running interpolation update: $\bar{x}_n = (1-\alpha)\cdot\bar{x}_{n-1} + \alpha\cdot x_n$
- Makes recent samples more important:

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

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

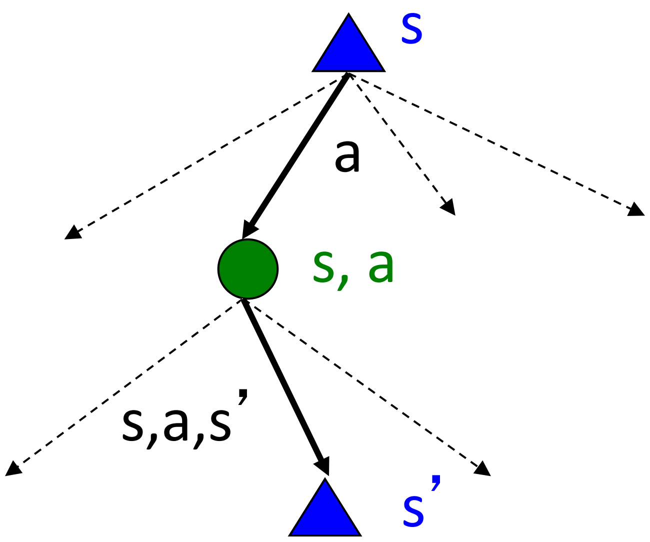
Decreasing learning rate (alpha) can give converging averages

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:

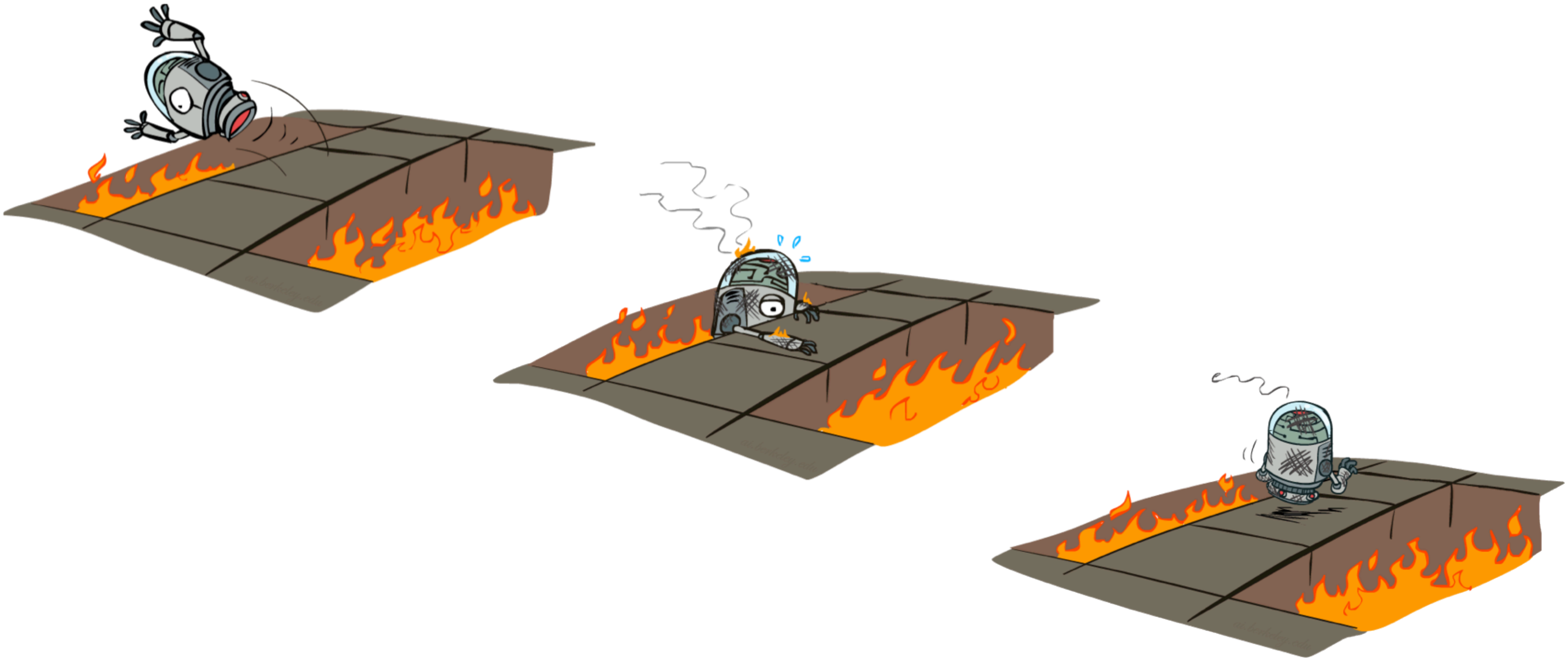
$$\pi(s) = \arg\max_a Q(s,a)$$

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



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

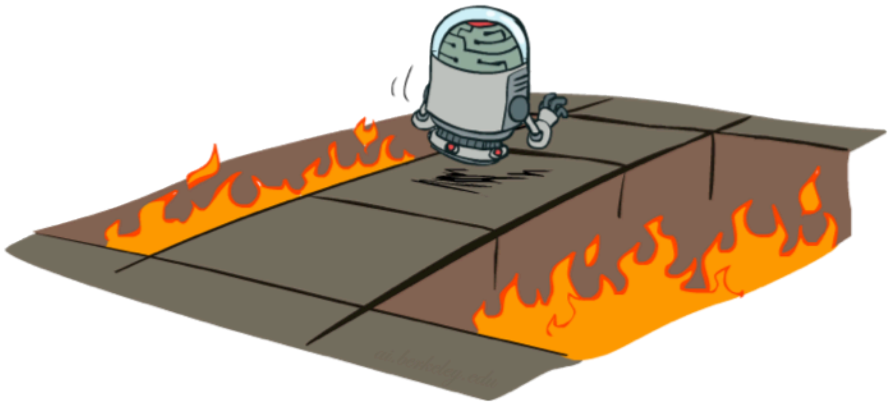
Active reinforcement learning



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… May mean diving into a pit!

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:

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

But Q-values are more useful and are just averages! 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:

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

Q-Learning

Q-Learning: sample-based Q-value iteration

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

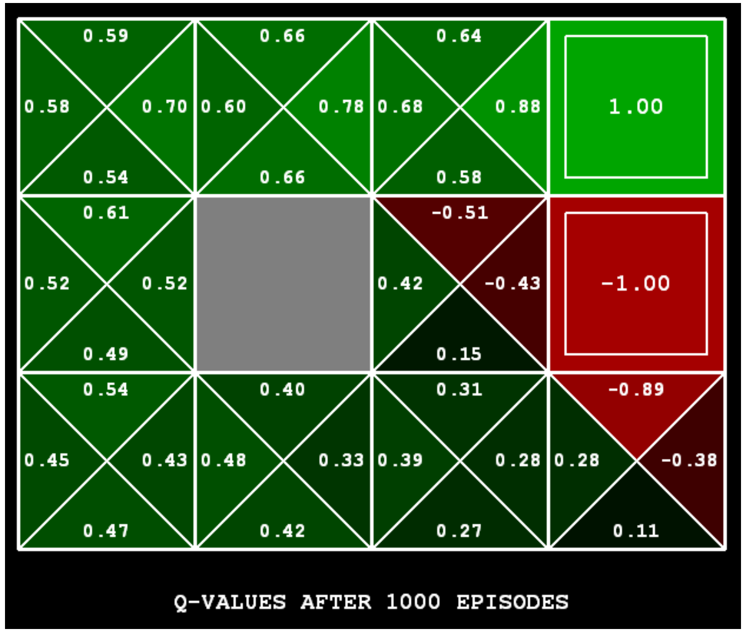
Learn Q(s,a) values as you go

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

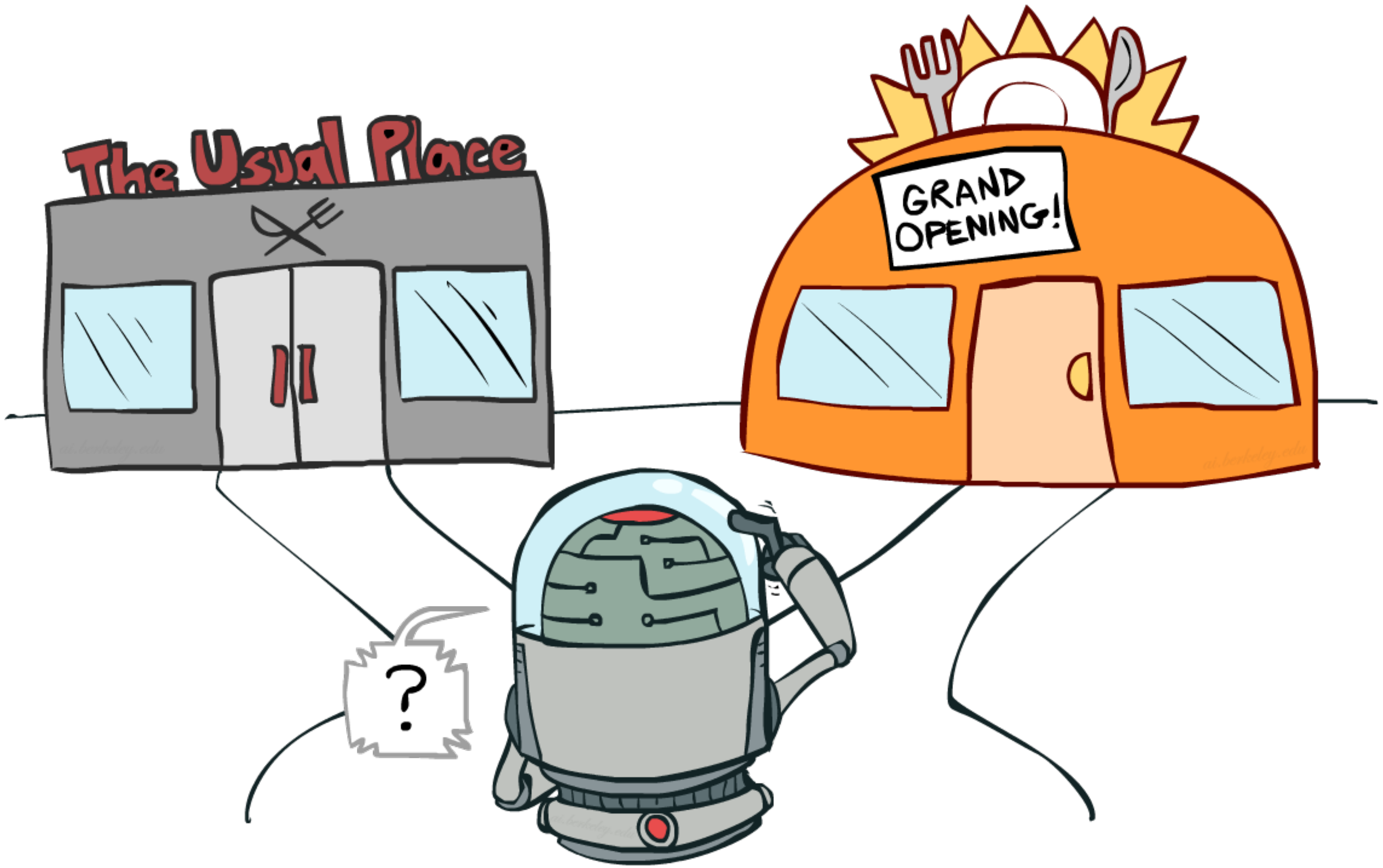
$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

- Incorporate the new estimate into a running average:

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



Exploration vs. exploitation



Exploration (or, the trouble with greed)

- Suppose we estimate model parameters at each step and then always acts optimally according to current estimates

- This may backfire! Why?

How to explore?

Several schemes for forcing exploration

- Simplest: random actions (ε-greedy)
  - Every time step, flip a coin
  - With (small) probability ε, act randomly
  - With (large) probability 1-ε, act on current policy

- Problems with random actions?
  - You do eventually explore the space, but keep thrashing around once learning is done
  - One solution: lower ε over time
  - Another solution: exploration functions

Exploration functions

When to explore?

- Random actions: explore a fixed amount
- Better idea: explore areas whose badness is not (yet) established, eventually stop exploring

Exploration function

- Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g.

$$f(u,n) = u + k/n$$

Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

Q-Learning Properties

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

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 (!)

Next time

- More reinforcement learning!
- Homeworks (programming + written bit) due by **Sunday midnight!**