

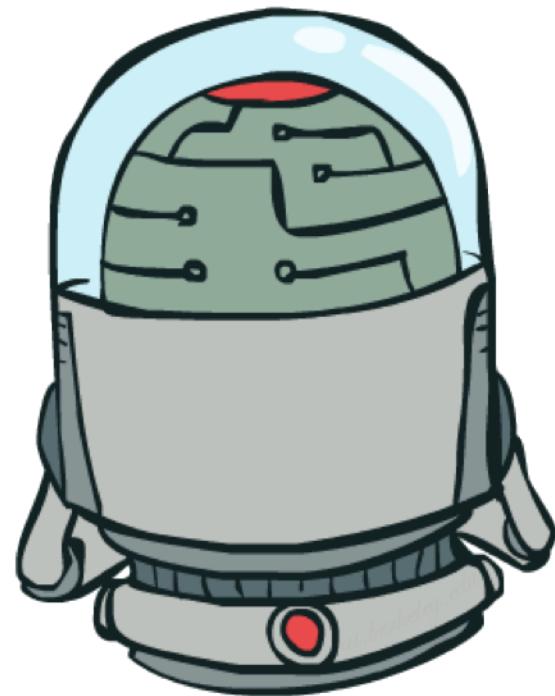
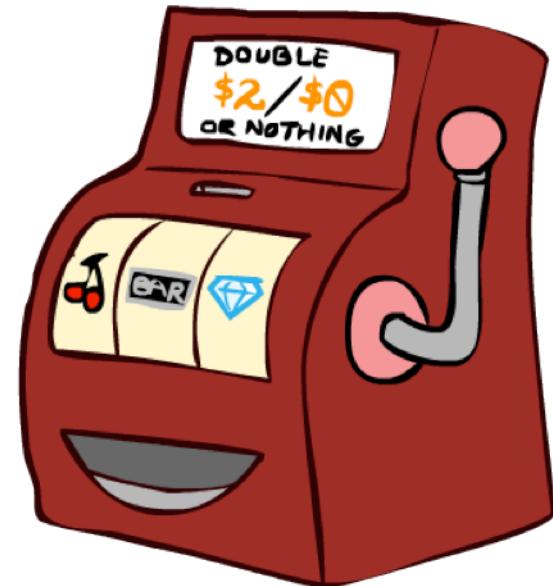
复旦大学大数据学院  
School of Data Science, Fudan University

魏忠钰

# Reinforcement Learning

May 2<sup>nd</sup>, 2018

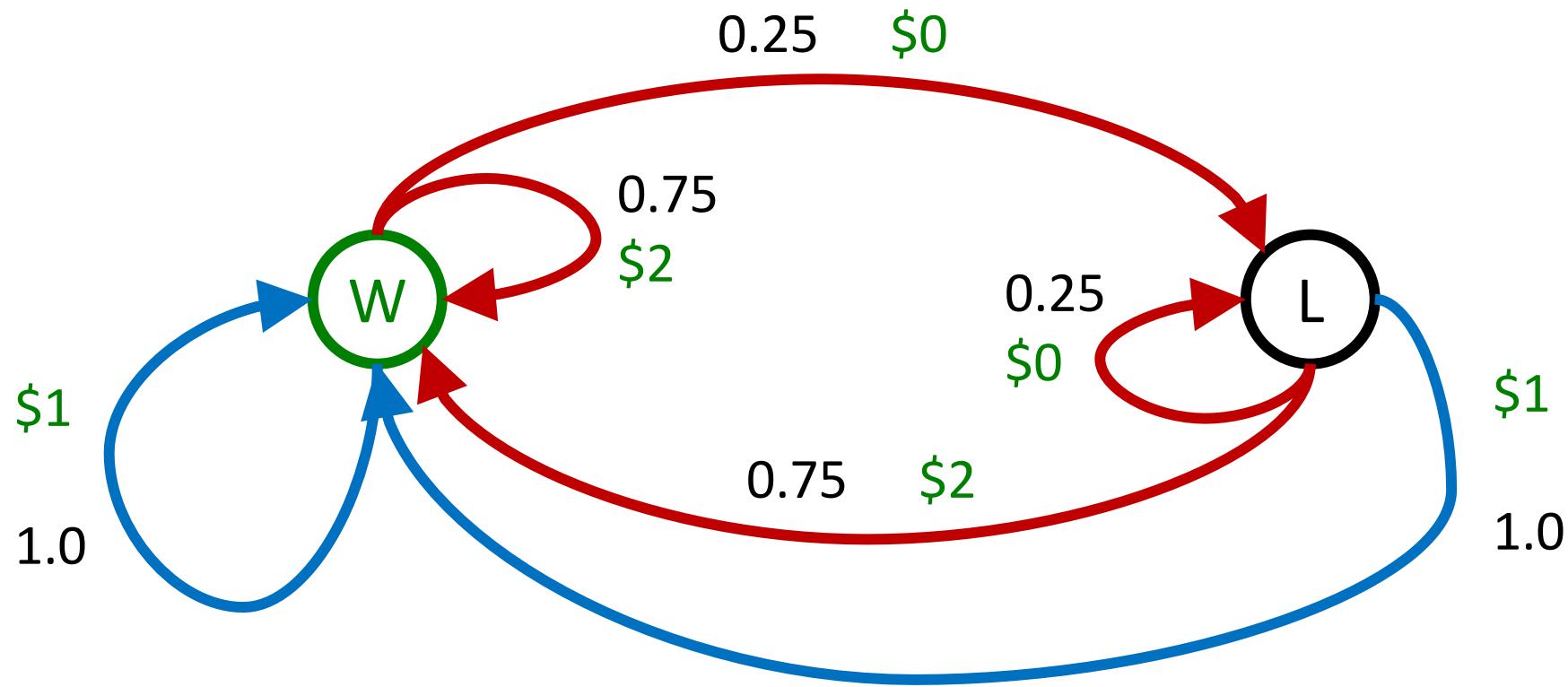
# Double Bandits



# Double-Bandit MDP

- Actions: *Blue, Red*
- States: *Win, Lose*

*No discount  
100 time steps  
Both states have  
the same value*

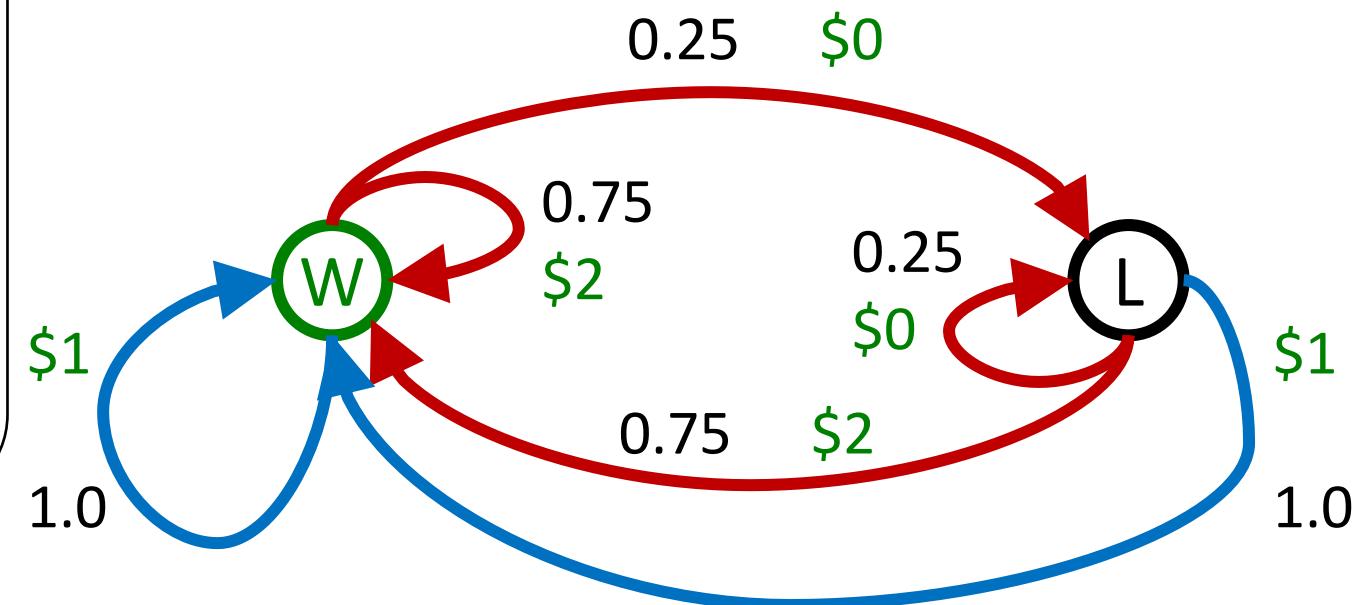


# Offline Planning

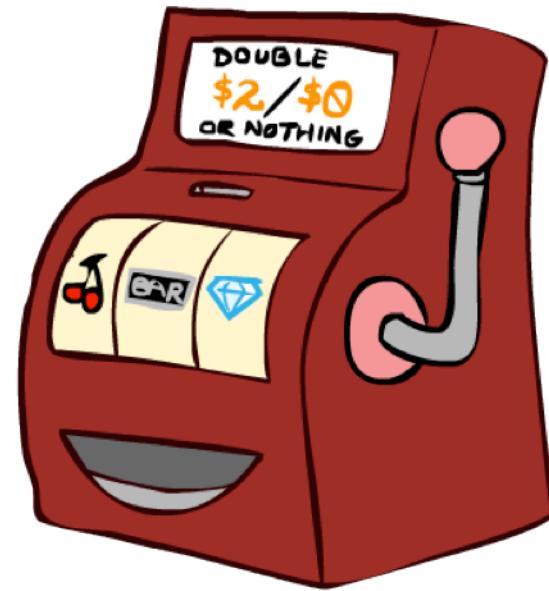
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

*No discount  
100 time steps  
Both states have  
the same value*

Value	
Play Red	150
Play Blue	100



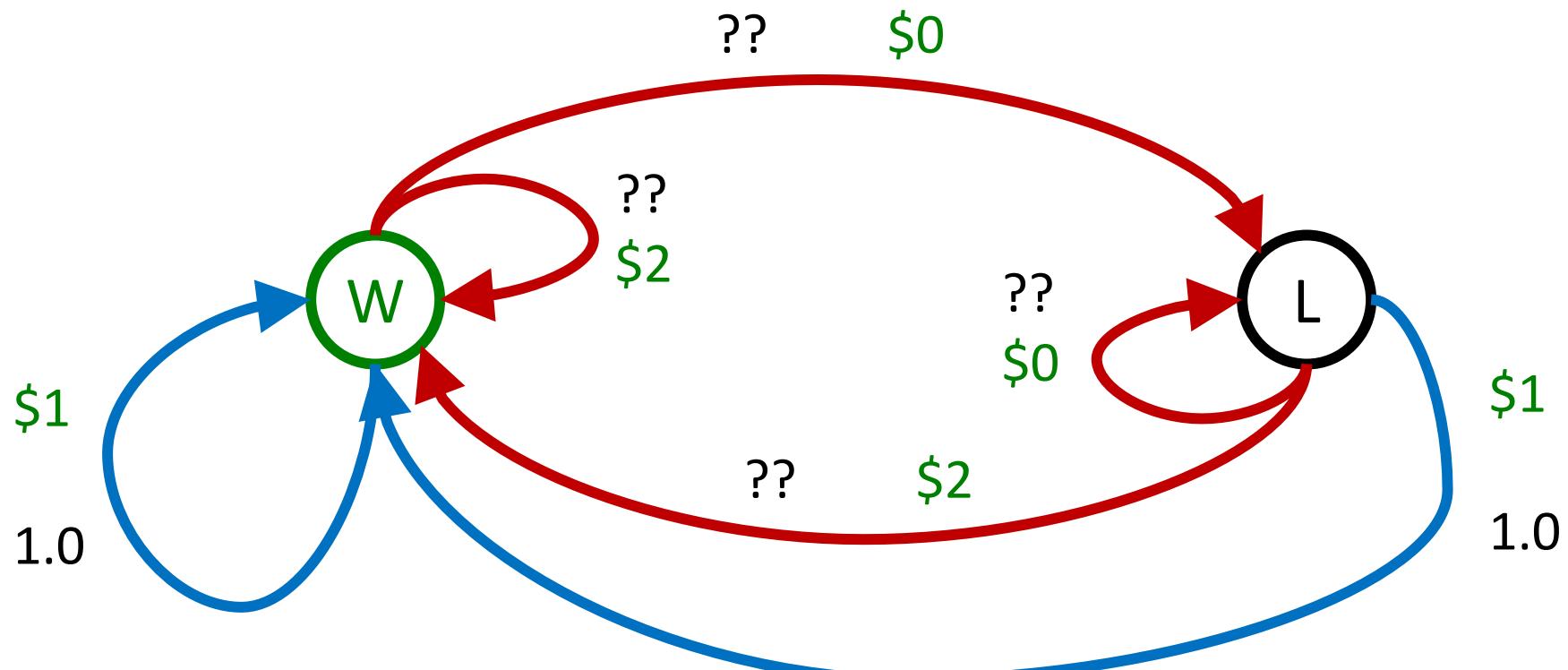
# Let's Play!



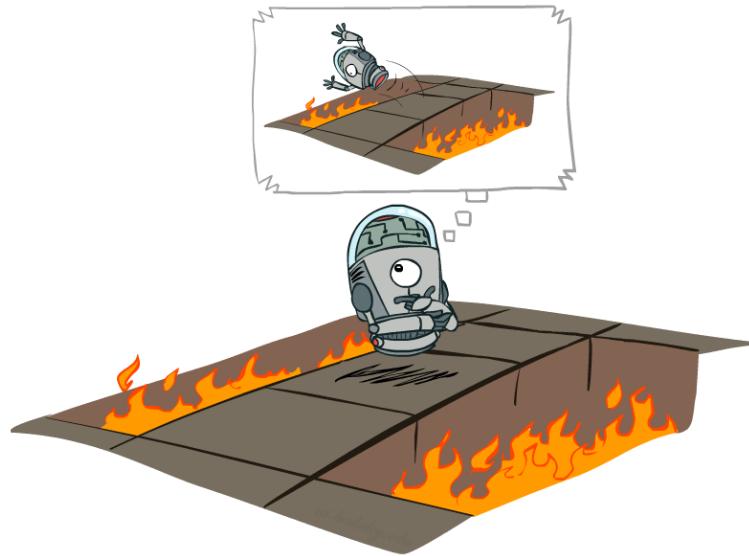
\$2	\$2	\$0	\$2	\$2
\$2	\$2	\$0	\$0	\$0

# Online Planning

- Rules changed! Red's win chance is different.



# Offline (MDPs) vs. Online (RL)



Offline Solution



Online Learning

# Example: Learning to Walk



Initial



A Learning Trial



After Learning  
[1K Trials]

## Example: Learning to Walk



Initial

## Example: Learning to Walk



## Example: Learning to Walk

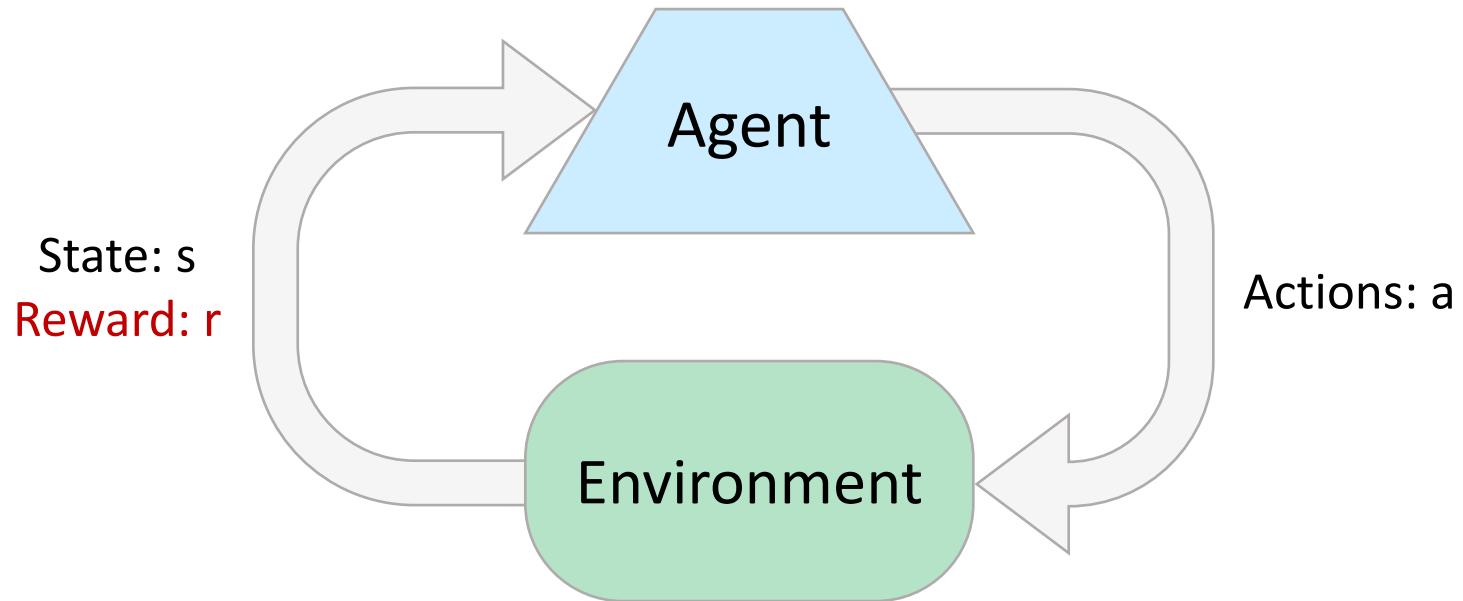


Finished

## Example: Toddler Robot



# Reinforcement Learning



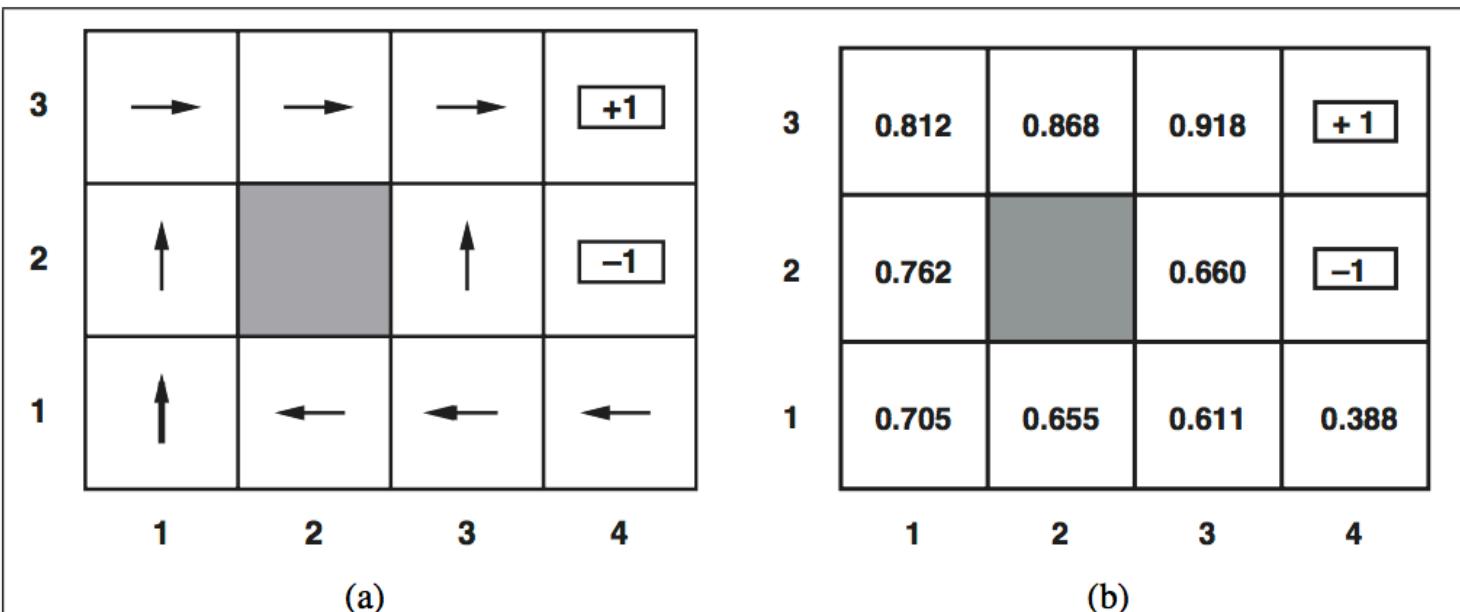
- Basic idea:
  - Receive feedback in the form of **rewards**
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to **maximize expected rewards**
  - All learning is based on observed samples of outcomes!

# 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  $\pi(s)$
- New twist: don't know  $T$  or  $R$ 
  - I.e. we don't know which states are good or what the actions do
  - Must actually try actions and states out to learn



# Trials for Reinforcement Learning



**Figure 21.1** (a) A policy  $\pi$  for the  $4 \times 3$  world; this policy happens to be optimal with rewards of  $R(s) = -0.04$  in the nonterminal states and no discounting. (b) The utilities of the states in the  $4 \times 3$  world, given policy  $\pi$ .

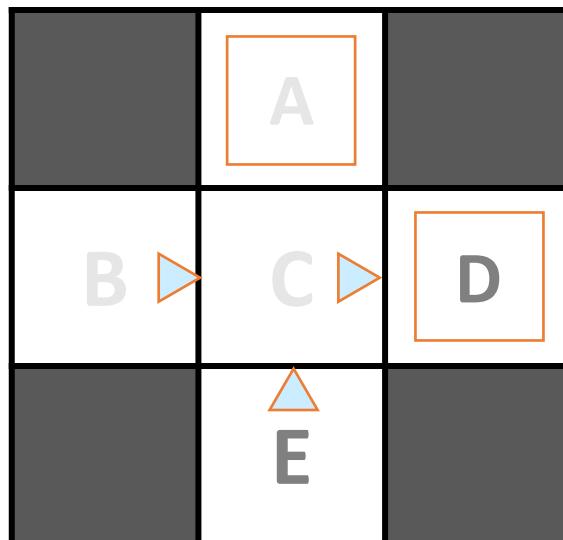
$(1, 1) \xrightarrow{-.04} (1, 2) \xrightarrow{-.04} (1, 3) \xrightarrow{-.04} (1, 2) \xrightarrow{-.04} (1, 3) \xrightarrow{-.04} (2, 3) \xrightarrow{-.04} (3, 3) \xrightarrow{-.04} (4, 3) +1$   
 $(1, 1) \xrightarrow{-.04} (1, 2) \xrightarrow{-.04} (1, 3) \xrightarrow{-.04} (2, 3) \xrightarrow{-.04} (3, 3) \xrightarrow{-.04} (3, 2) \xrightarrow{-.04} (3, 3) \xrightarrow{-.04} (4, 3) +1$   
 $(1, 1) \xrightarrow{-.04} (2, 1) \xrightarrow{-.04} (3, 1) \xrightarrow{-.04} (3, 2) \xrightarrow{-.04} (4, 2) -1 .$

# 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

# Example: Model-Based Learning

Input Policy  $\pi$



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

Learned Model

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

$$\begin{aligned} T(B, \text{east}, C) &= 1.00 \\ T(C, \text{east}, D) &= 0.75 \\ T(C, \text{east}, A) &= 0.25 \\ &\dots \end{aligned}$$

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

$$\begin{aligned} R(B, \text{east}, C) &= -1 \\ R(C, \text{east}, D) &= -1 \\ R(D, \text{exit}, x) &= +10 \\ &\dots \end{aligned}$$

What if we have millions of states?

# Example: Expected Age

**Goal: Compute expected age of DATA130008 students**

Known P(A)

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

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

Unknown P(A): “Model Based”

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

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

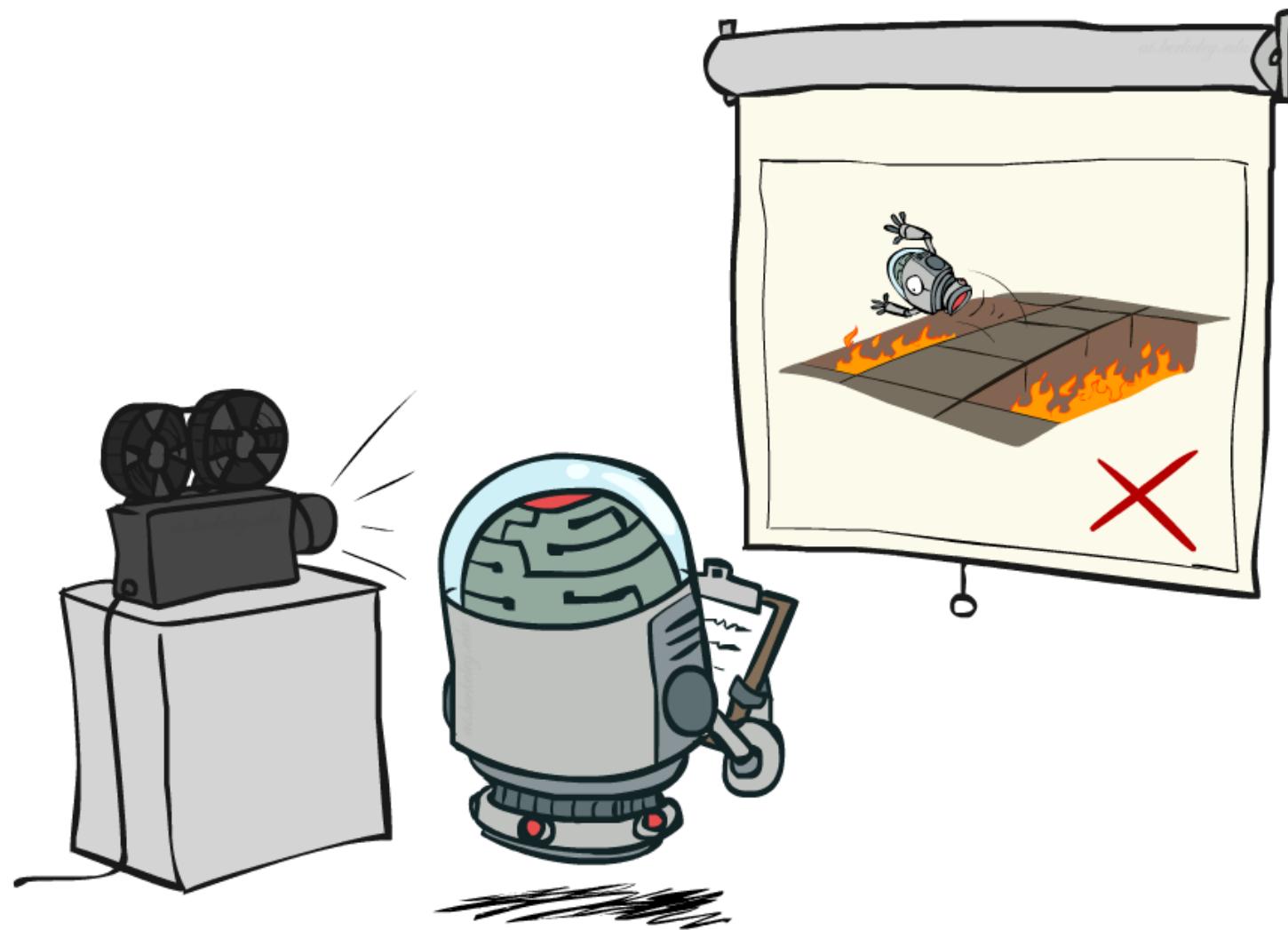
Unknown P(A): “Model Free”

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

# Model-Free Learning



# 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:
  - 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 (Model-free)

- Goal: Compute values for each state under  $\pi$

$$U^\pi(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t R(S_t) \right]$$

- Idea: Average together observed sample values
  - Act according to  $\pi$
  - Every time you visit a state, write down what **the sum of discounted rewards turned out to be**
  - Average those samples

$$U^\pi(s) = \frac{1}{N} \left( \sum_{i=1}^n u_i^\pi(s) \right)$$

(1, 1).-04~~(1, 2).-04~~(1, 3).-04~~(1, 2).-04~~(1, 3).-04~~(2, 3).-04~~(3, 3).-04~~(4, 3)+1  
(1, 1).-04~~(1, 2).-04~~(1, 3).-04~~(2, 3).-04~~(3, 3).-04~~(3, 2).-04~~(3, 3).-04~~(4, 3)+1  
(1, 1).-04~~(2, 1).-04~~(3, 1).-04~~(3, 2).-04~~(4, 2).-1 .

# Example: Direct Evaluation

Input Policy  $\pi$

Observed Episodes (Training)

Output Values

Episode 1

Episode 2

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

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

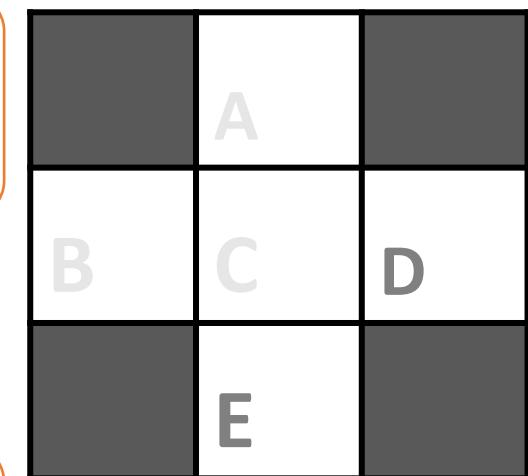
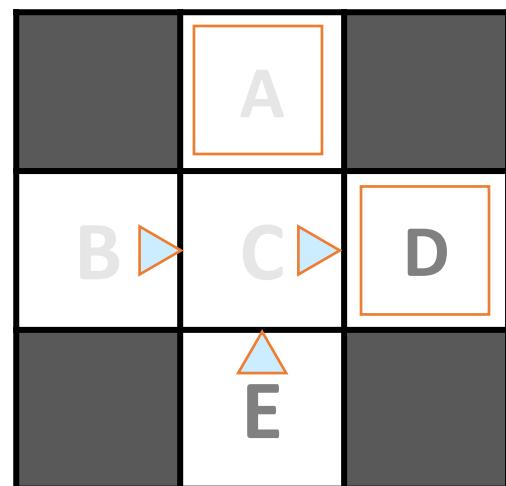
Episode 3

Episode 4

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

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

Assume:  $\gamma = 1$



# Value Updating

- Goal: Compute values for each state under  $\pi$

$$U^\pi(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t R(S_t) \right]$$

- Idea: Update the policy value based on observed value

$$U^\pi(s) = (1 - \eta)U^\pi(s) + \eta u^\pi(s)$$

# 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

Trial 1: B, C, D

Trial 2: B, C, D

Trial 3: E, C, D

Trial 4: E, C, A

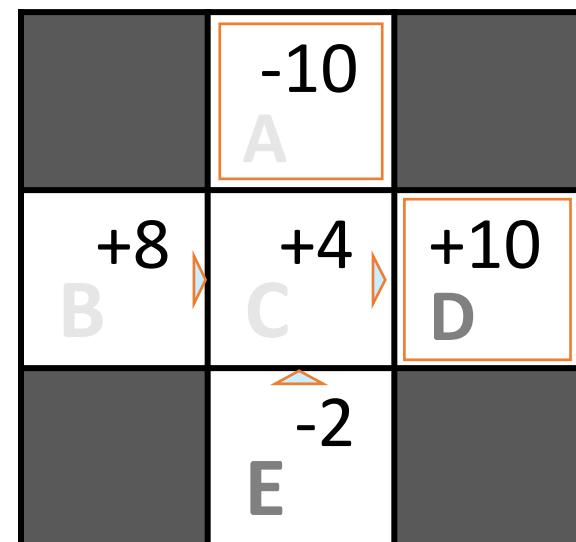
C: 4 times

E: 2 times

B: 2 times

Why not use the value of C to help estimate the value of E and B?

Output Values



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

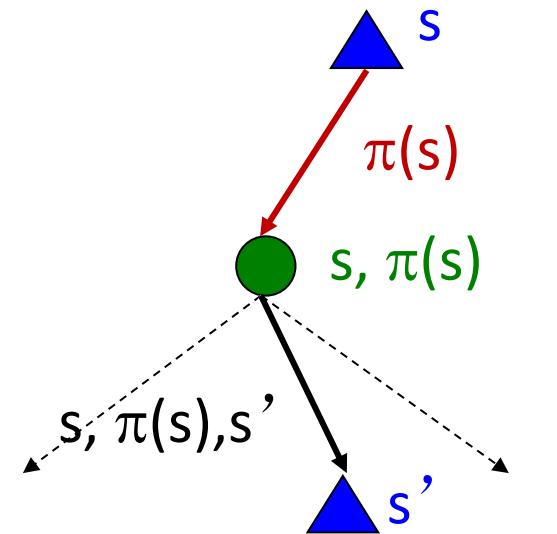
# Why Not Use Policy Evaluation?

- 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



# Model-based: Pseudocode for ADP algorithm

```
function PASSIVE-ADP-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $\pi$ , a fixed policy
     $mdp$ , an MDP with model  $P$ , rewards  $R$ , discount  $\gamma$ 
     $U$ , 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  $U[s'] \leftarrow r'; R[s'] \leftarrow r'$ 
  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
       $P(t | s, a) \leftarrow N_{s'|sa}[t, s, a] / N_{sa}[s, a]$ 
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$ 
  if  $s'.\text{TERMINAL?}$  then  $s, a \leftarrow \text{null}$  else  $s, a \leftarrow s', \pi[s']$ 
  return  $a$ 
```

**Figure 21.2** A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 657.

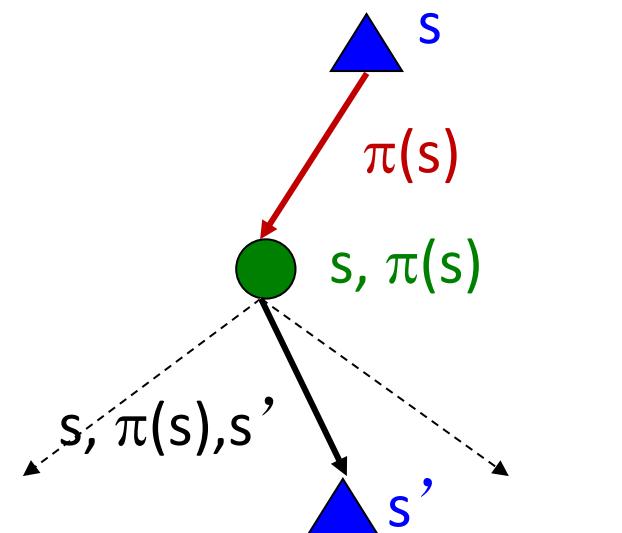
# Why Not Use Policy Evaluation?

- 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
- How about model-free approach?
- Key question: how can we do this update to  $V$  without knowing  $T$  and  $R$ ?
  - In other words, how do 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

$$\text{sample}_1 = R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1)$$

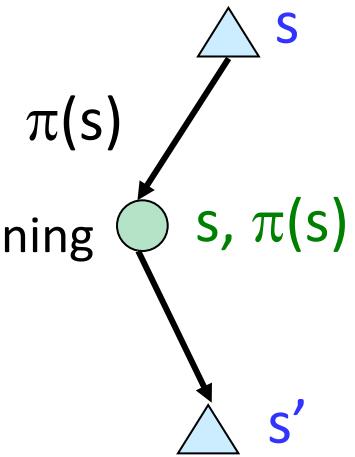
$$\text{sample}_2 = R(s, \pi(s), s'_2) + \gamma V_k^{\pi}(s'_2)$$

$$\text{sample}_n = R(s, \pi(s), s'_n) + \gamma V_k^{\pi}(s'_n)$$

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

# Temporal Difference Learning

- 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 for passive RL
  - 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  $V(s)$ :  $V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)sample$

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

# Exponential Moving Average

- 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

- Alpha = 0: averaging all the samples

# Algorithm for Temporal Difference Learning

```
function PASSIVE-TD-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $\pi$ , a fixed policy
     $U$ , a table of utilities, initially empty
     $N_s$ , a table of frequencies for states, initially zero
     $s, a, r$ , the previous state, action, and reward, initially null

  if  $s'$  is new then  $U[s'] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_s[s]$ 
     $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$ 
  if  $s'.\text{TERMINAL?}$  then  $s, a, r \leftarrow \text{null}$  else  $s, a, r \leftarrow s', \pi[s'], r'$ 
  return  $a$ 
```

**Figure 21.4** A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function  $\alpha(n)$  is chosen to ensure convergence, as described in the text.

# Example: Temporal Difference Learning

States

Observed Transitions

B, east, C, -2

C, east, D, -2

	A								
B	C	D							
E									

	0								
0	0	8		-1	0	8	-1	3	8

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

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

# Example: Temporal Difference Learning

States

Observed Transitions

B, east, C, -2

C, east, D, -2

	A	
B	C	D
E		

	0	
-1	3	8
	0	

	0	
0		8
	0	

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

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

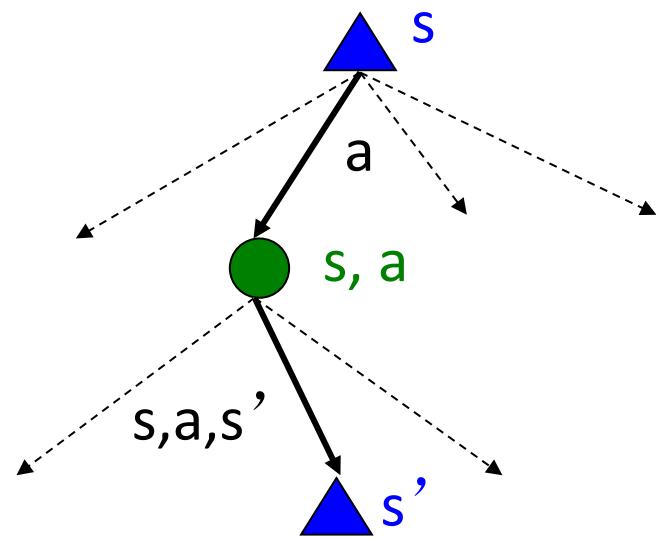
# What passive learning can not do?

- 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:

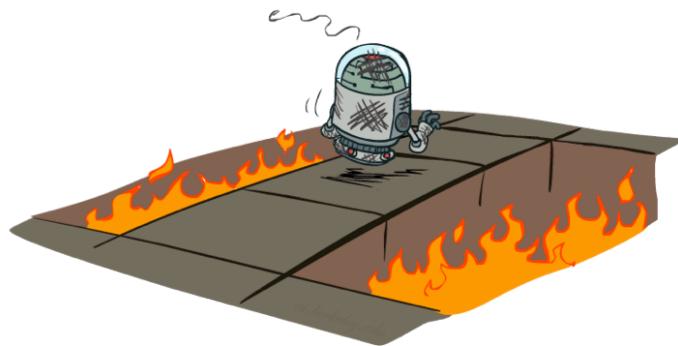
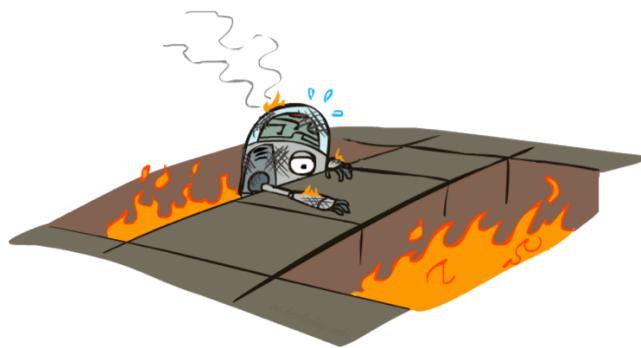
$$\pi(s) = \arg \max_a Q(s, a)$$

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

- Learn all Q 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:
  - Fundamental tradeoff: **exploration** vs. **exploitation**
  - This is NOT offline planning! You actually take actions in the world and find out what happens...

# Q-Value Iteration

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

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

- 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:

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

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

# Q-Learning

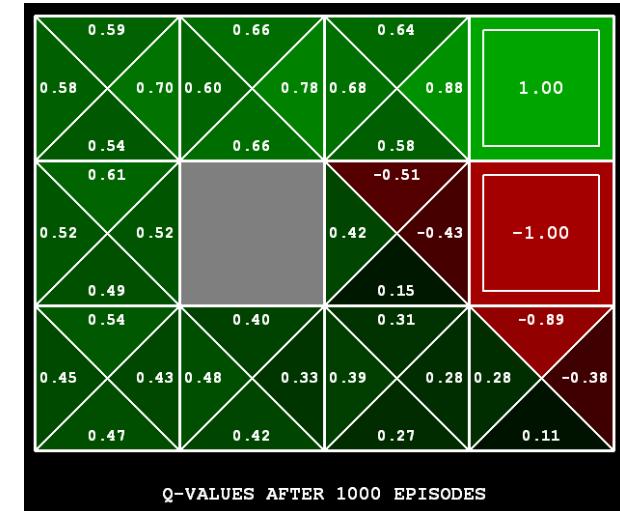
- Q-Learning: sample-based Q-value iteration

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

- Learn  $Q(s, a)$  values

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

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



- Incorporate the new estimate into a running average:

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

# Video of Demo Q-Learning -- Gridworld

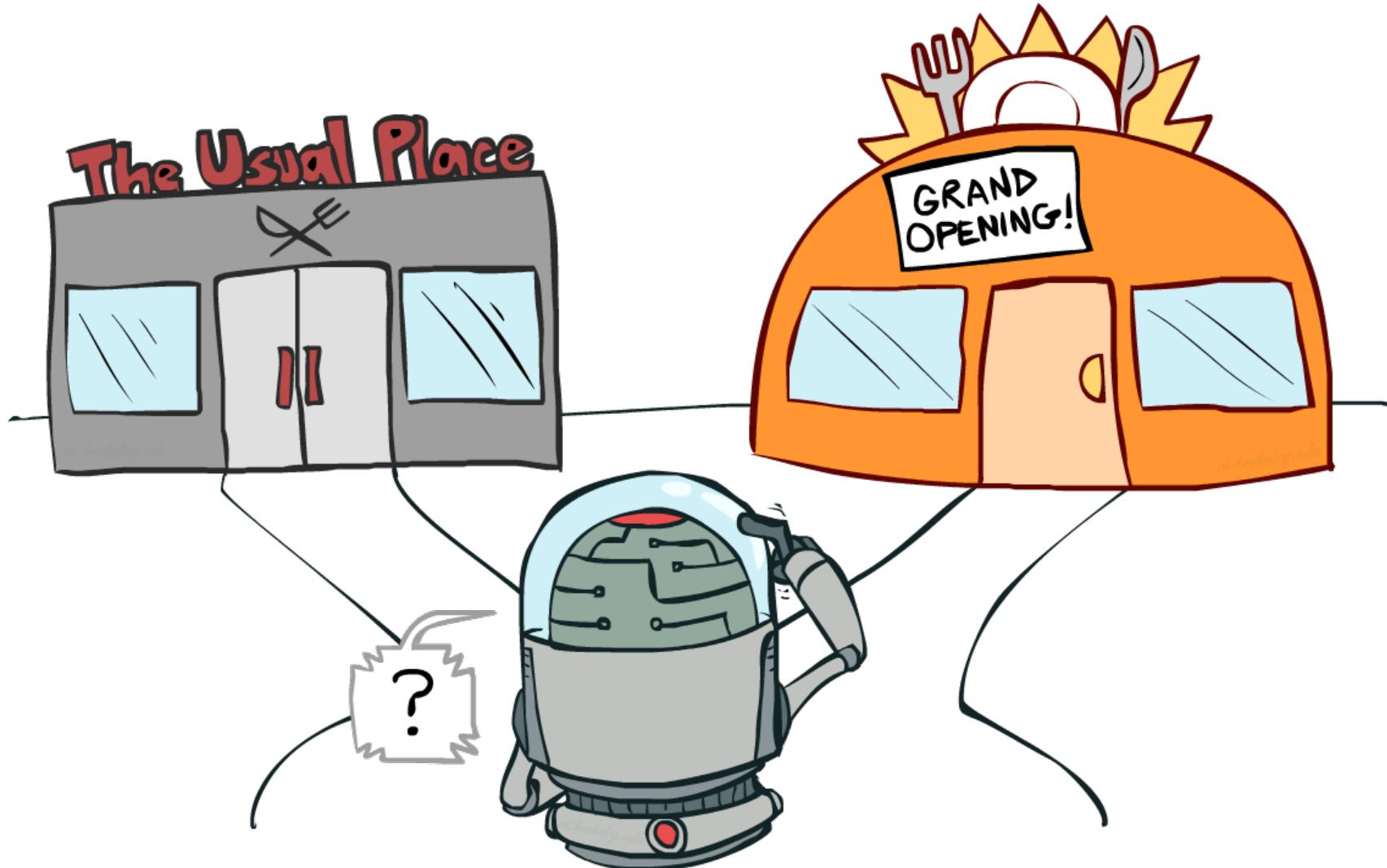


# Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting sub-optimally!
- This is called **off-policy learning**
- In Comparison with the another approach named SARSA (on-policy, refer to the text book).
- 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 in sequence (!)

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

# Exploration vs. Exploitation



# How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions ( $\varepsilon$ -greedy)
    - Every time step, flip a coin
    - With (small) probability  $\varepsilon$ , act randomly
    - With (large) probability  $1-\varepsilon$ , 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  $\varepsilon$  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:

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

- Note: this propagates the “bonus” back to states that lead to unknown states as well!

# Regret

---

- Even if you learn the optimal policy, you still make mistakes along the way!
- Regret is a measure of your total mistake cost: the difference between your (expected) rewards and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal – it requires optimally **learning to be optimal**
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret

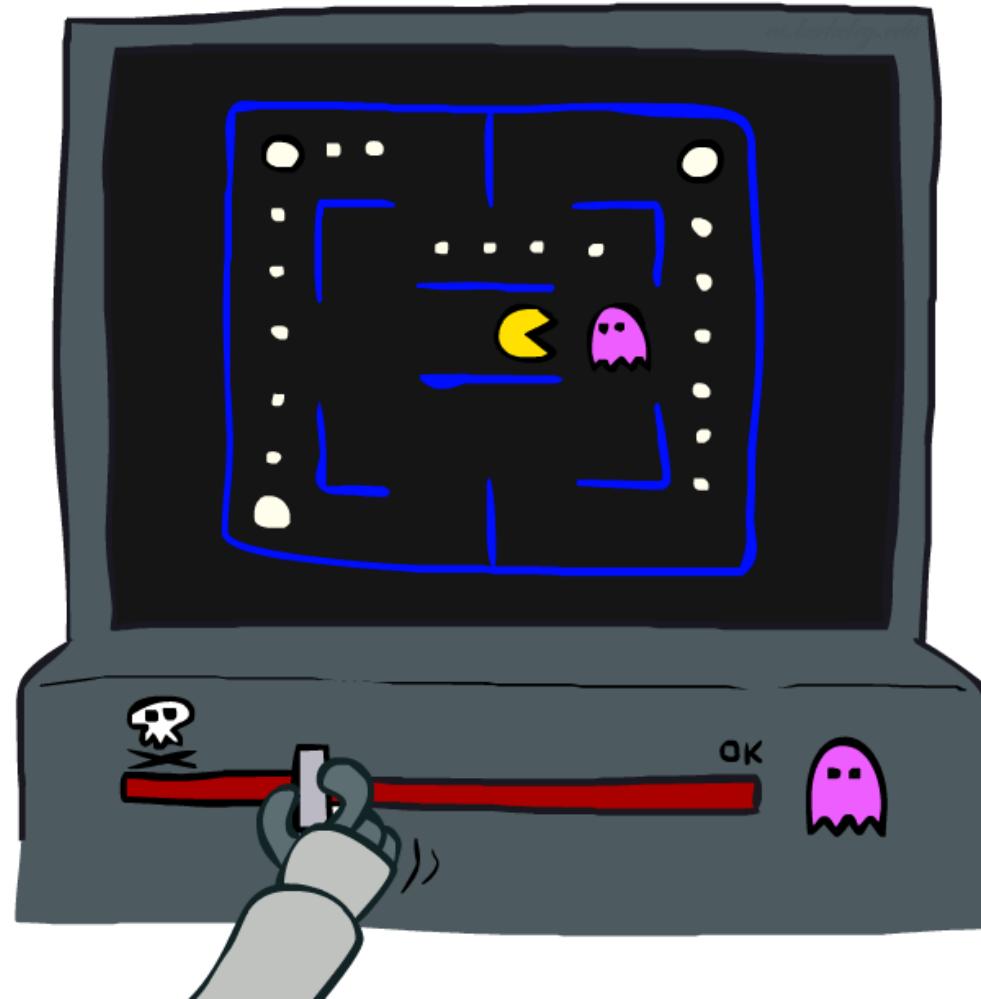
# Q-Learning Algorithm

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $Q$ , a table of action values indexed by state and action, initially zero
     $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
     $s, a, r$ , the previous state, action, and reward, initially null

  if TERMINAL?( $s$ ) then  $Q[s, \text{None}] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$ 
     $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$ 
     $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$ 
  return  $a$ 
```

**Figure 21.8** An exploratory Q-learning agent. It is an active learner that learns the value  $Q(s, a)$  of each action in each situation. It uses the same exploration function  $f$  as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

# Approximate Q-Learning

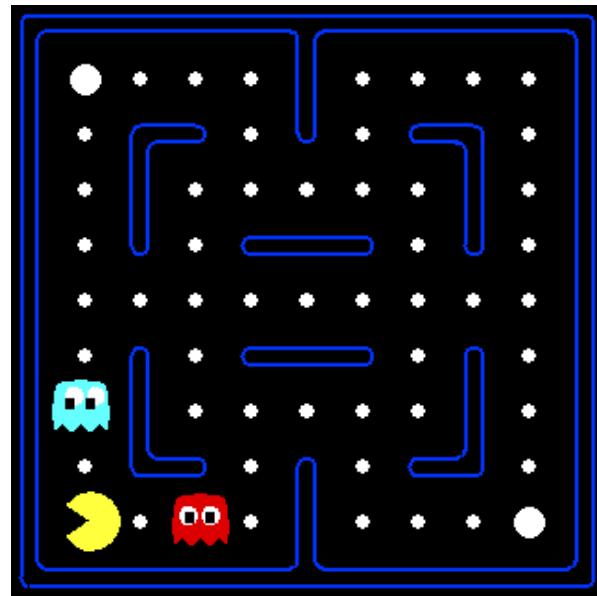


# Generalizing Across States

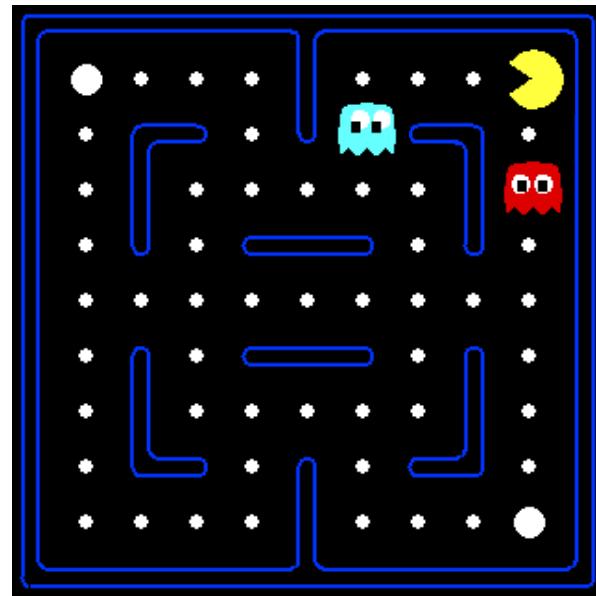
- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations

# Example: Pacman

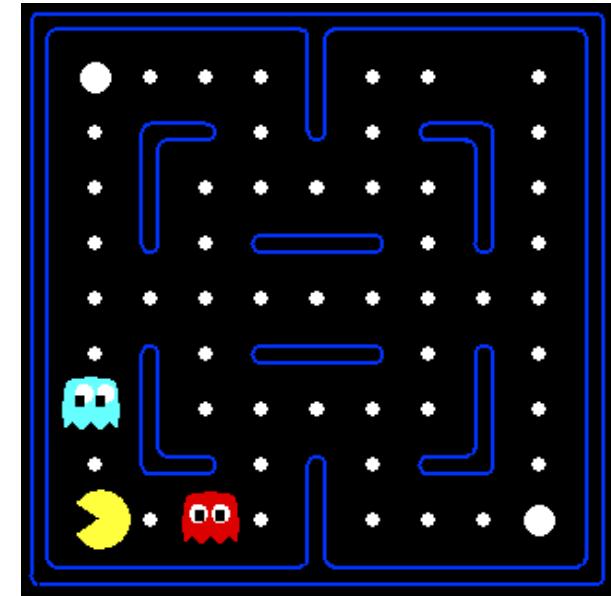
Let's say we discover through experience that this state is bad:



In naïve q-learning, we know nothing about this state:

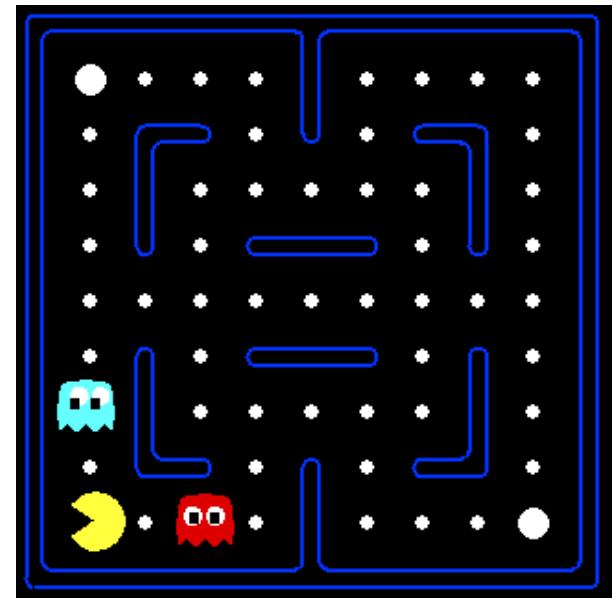


Or even this one!



# Feature-Based Representations

- Solution: describe a state using a vector of features (properties)
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
- Example features:
  - Distance to closest ghost
  - Distance to closest dot
  - Number of ghosts
  - $1 / (\text{dist to dot})^2$
  - Is Pacman in a tunnel? (0/1)
  - ..... etc.
  - Is it the exact state on this slide?
- Can also describe a q-state  $(s, a)$  with features (e.g. action moves closer to food)



# Linear Value Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$$

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but actually be very different in value!

# Approximate Q-Learning

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$$

- Q-learning with linear Q-functions:

transition =  $(s, a, r, s')$

$$\text{difference} = \left[ r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a)$$

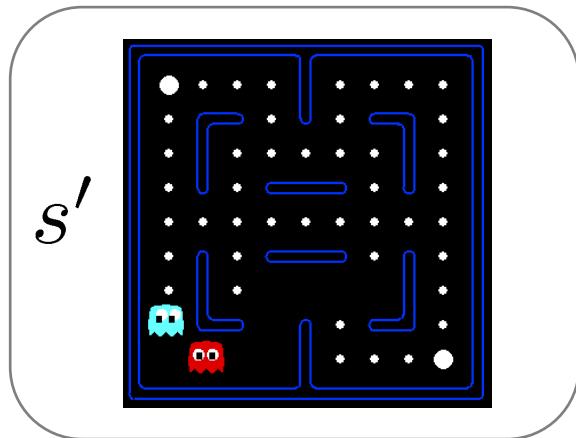
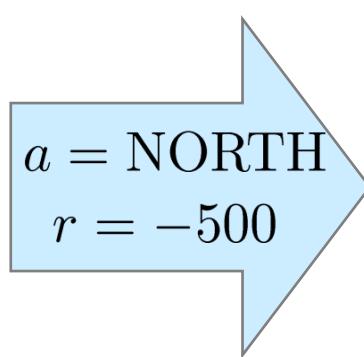
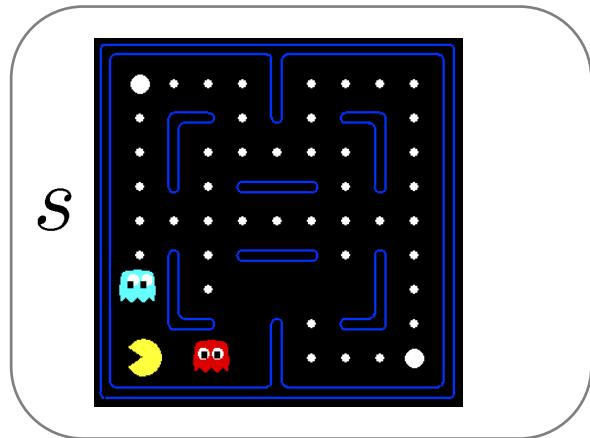
$$Q(s, a) \leftarrow Q(s, a) + \alpha \text{ [difference]}$$

$$w_i \leftarrow w_i + \alpha \text{ [difference]} f_i(s, a)$$

- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: dis-prefer all states with that state's features
- Formal justification: online least squares

## Example: Q-Pacman

$$Q(s, a) = 4.0f_{DOT}(s, a) - 1.0f_{GST}(s, a)$$



$$f_{DOT}(s, \text{NORTH}) = 0.5$$

$$f_{GST}(s, \text{NORTH}) = 1.0$$

$$Q(s, \text{NORTH}) = +1$$

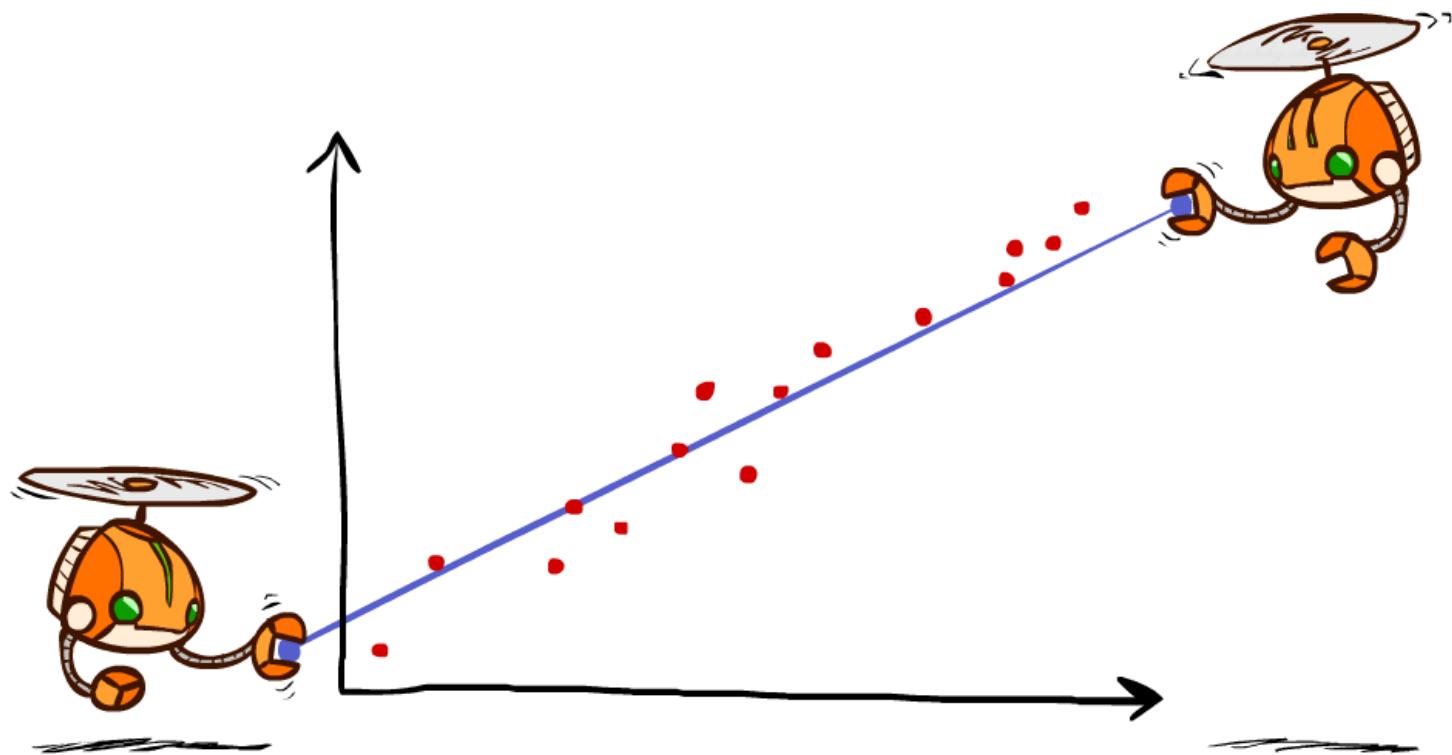
$$Q(s', \cdot) = 0$$

$$r + \gamma \max_{a'} Q(s', a') = -500 + 0$$

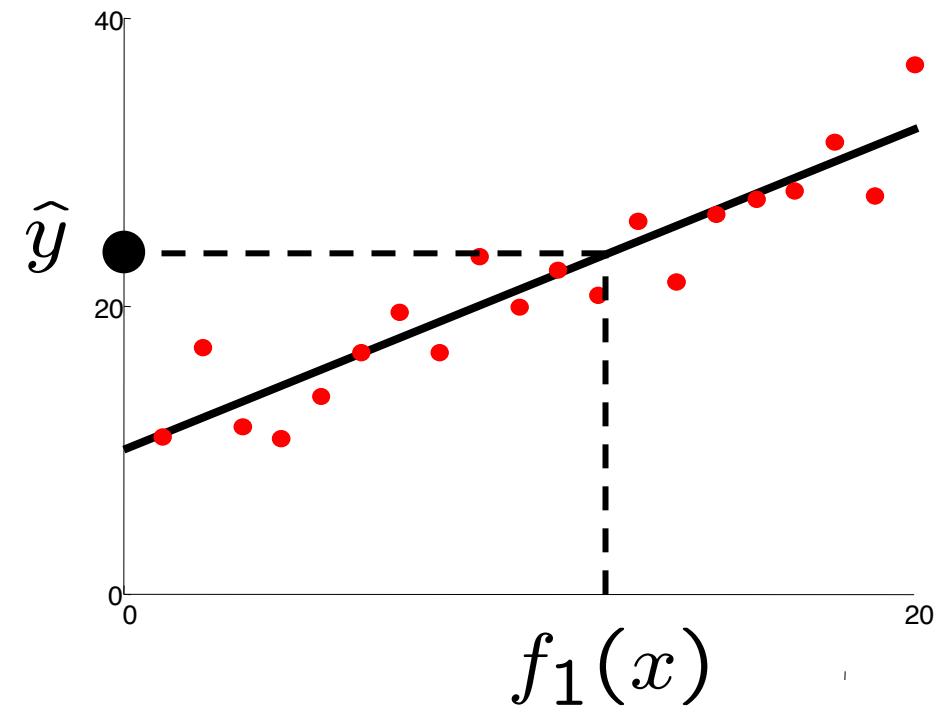
difference =  $-501$   $w_{DOT} \leftarrow 4.0 + \alpha [-501] 0.5$   
 $w_{GST} \leftarrow -1.0 + \alpha [-501] 1.0$

$$Q(s, a) = 3.0f_{DOT}(s, a) - 3.0f_{GST}(s, a)$$

# Q-Learning and Least Squares



# Linear Approximation: Regression

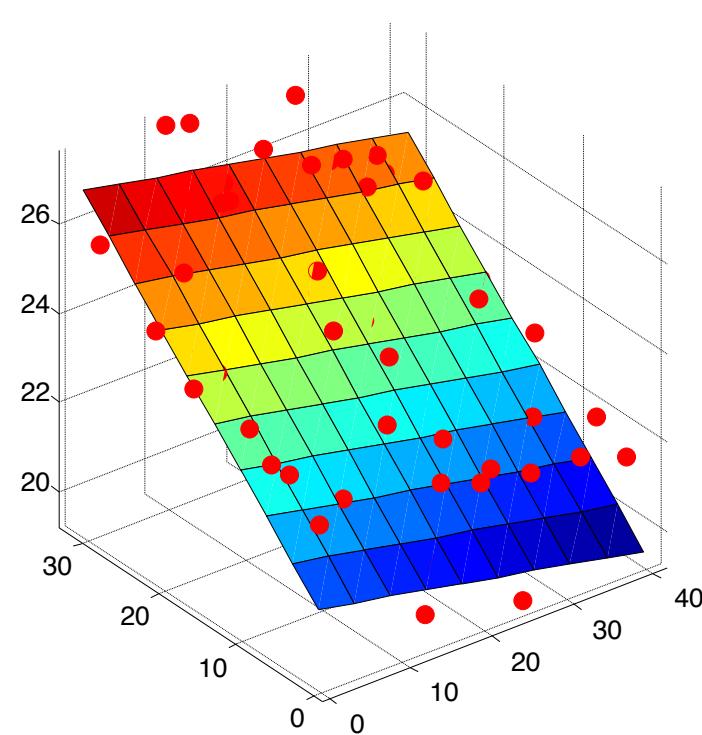


Prediction:

$$\hat{y} = w_0 + w_1 f_1(x)$$

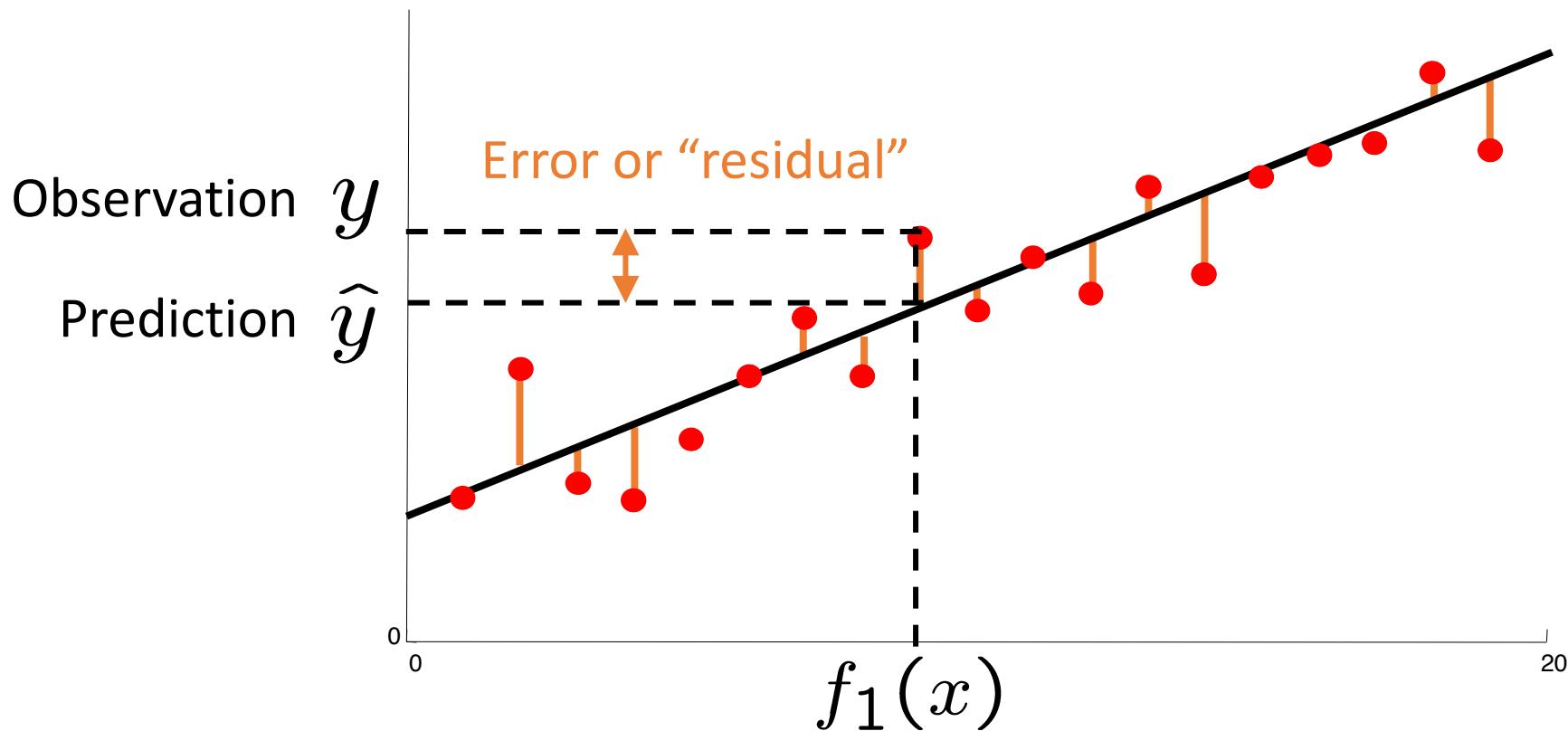
Prediction:

$$\hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x)$$



# Optimization: Least Squares

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left( y_i - \sum_k w_k f_k(x_i) \right)^2$$



# Minimizing Error

Imagine we had only one point  $x$ , with features  $f(x)$ , target value  $y$ , and weights  $w$ :

$$\text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2$$

$$\frac{\partial \text{error}(w)}{\partial w_m} = - \left( y - \sum_k w_k f_k(x) \right) f_m(x)$$

$$w_m \leftarrow w_m + \alpha \left( y - \sum_k w_k f_k(x) \right) f_m(x)$$

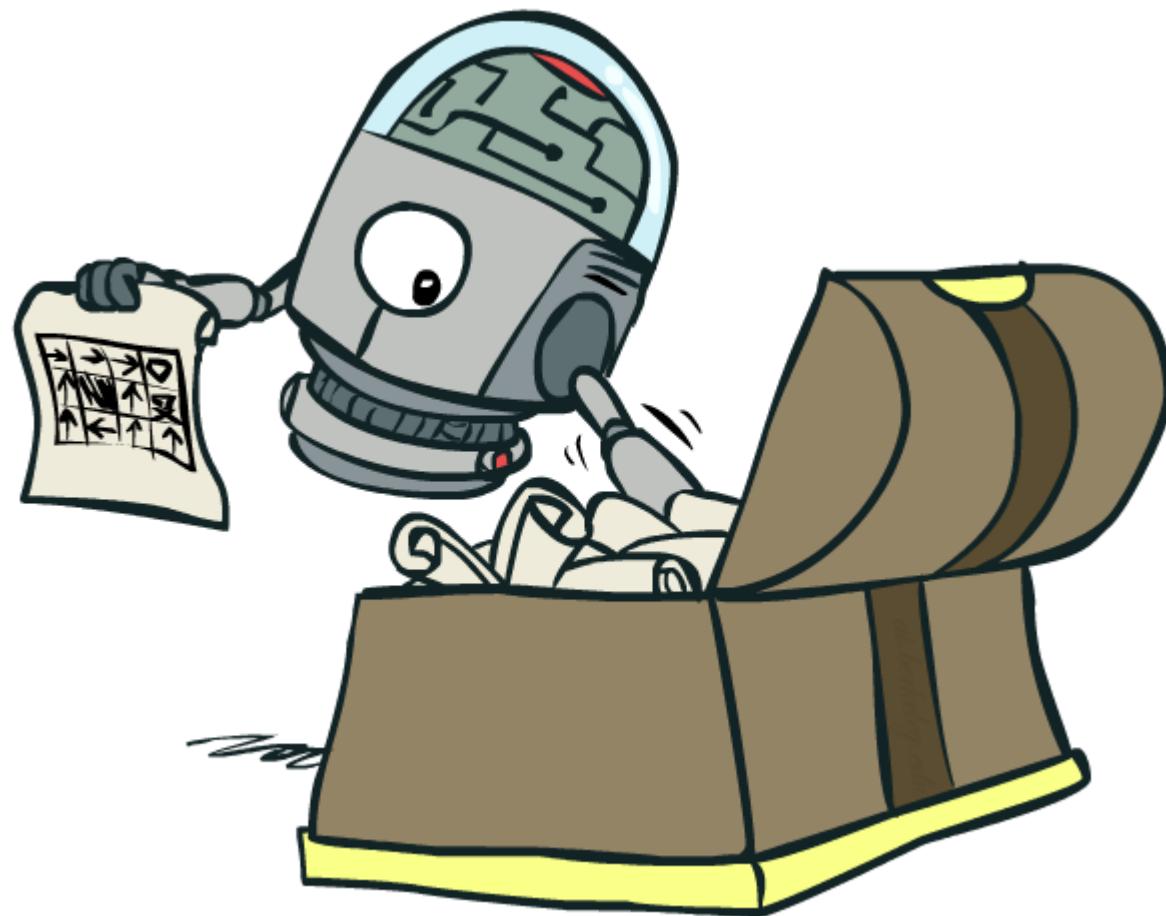
Approximate q update explained:

$$w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a') - Q(s, a) \right] f_m(s, a)$$

“target”

“prediction”

# Policy Search



# Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate  $V$  /  $Q$  best
  - Q-learning's priority: get  $Q$ -values close (modeling)
  - Action selection priority: get ordering of  $Q$ -values right (prediction)
- Solution: learn policies that maximize rewards, not the values that predict them
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

# Policy Search

- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - Tune each feature weight up and down and see if your policy is better than before
- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical
- Better methods exploit look-ahead structure, sample wisely, change multiple parameters...

# Conclusion

---

- We've seen how AI methods can solve problems in:
  - Search
  - Constraint Satisfaction Problems
  - Games
  - Markov Decision Problems
  - Reinforcement Learning