

CS 246:

Artificial Intelligence



010101
010101
010101



Instructor: Br. Tamal Mj

Image credit
<https://futureoflife.org/>

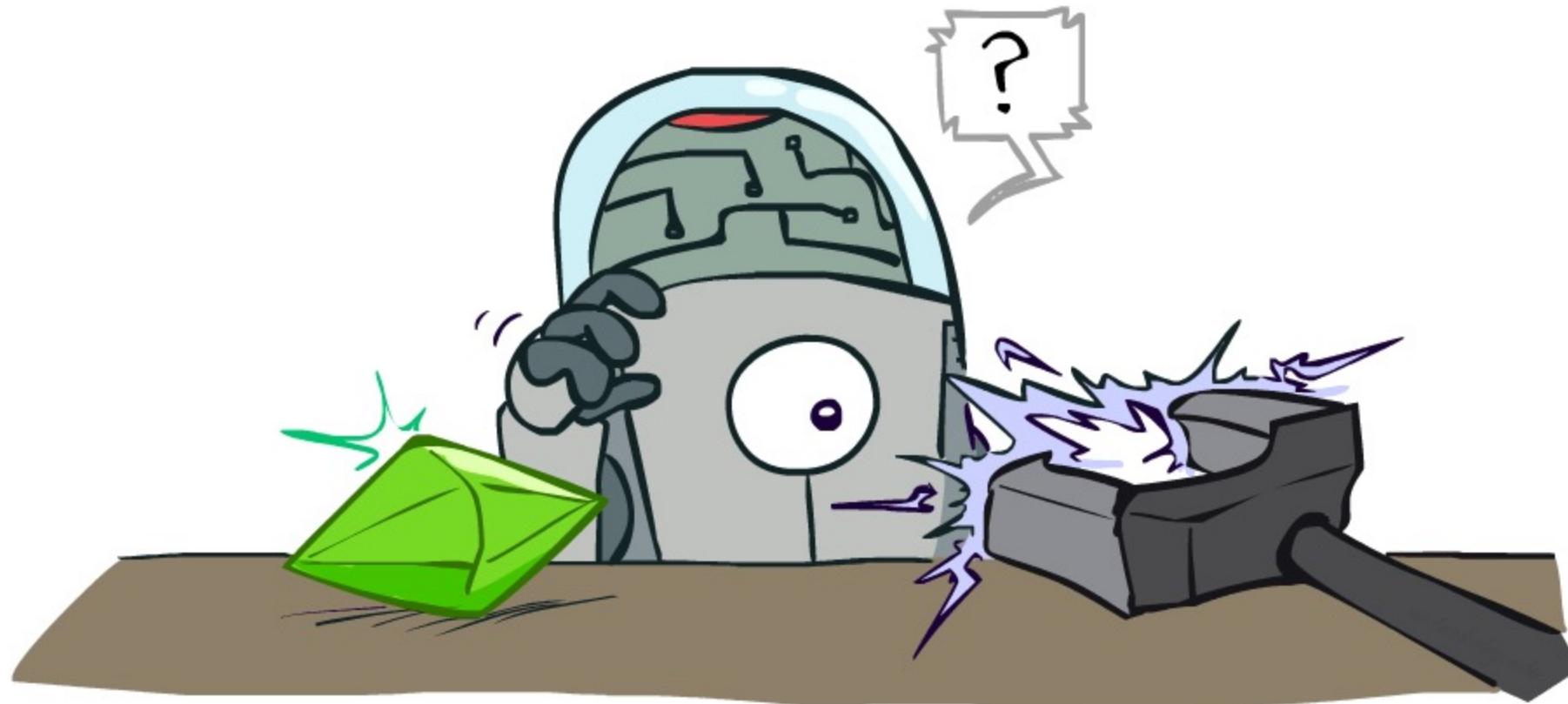
[slides adapted from Dan Klein, Pieter Abbeel, Sergey Levine & Stuart Russel (University of California, Berkeley)]



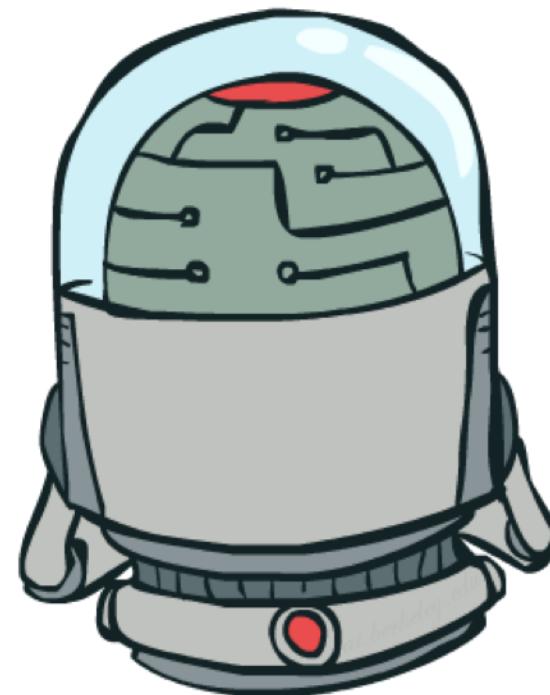
Om Saha Naav[au]-Avatu
Saha Nau Bhunaktu
Saha Viiryam Karavaavahai
Tejasvi Naav[au]-Adhiitam-
Astu Maa Vidvissaavahai
Om Shaantih Shaantih
Shaantih

Om, May we all be protected
May we all be nourished
May we work together with great energy
May our intellect be sharpened (may our study be effective)
Let there be no Animosity amongst us
Om, peace (in me), peace (in nature), peace (in divine forces)

Reinforcement Learning

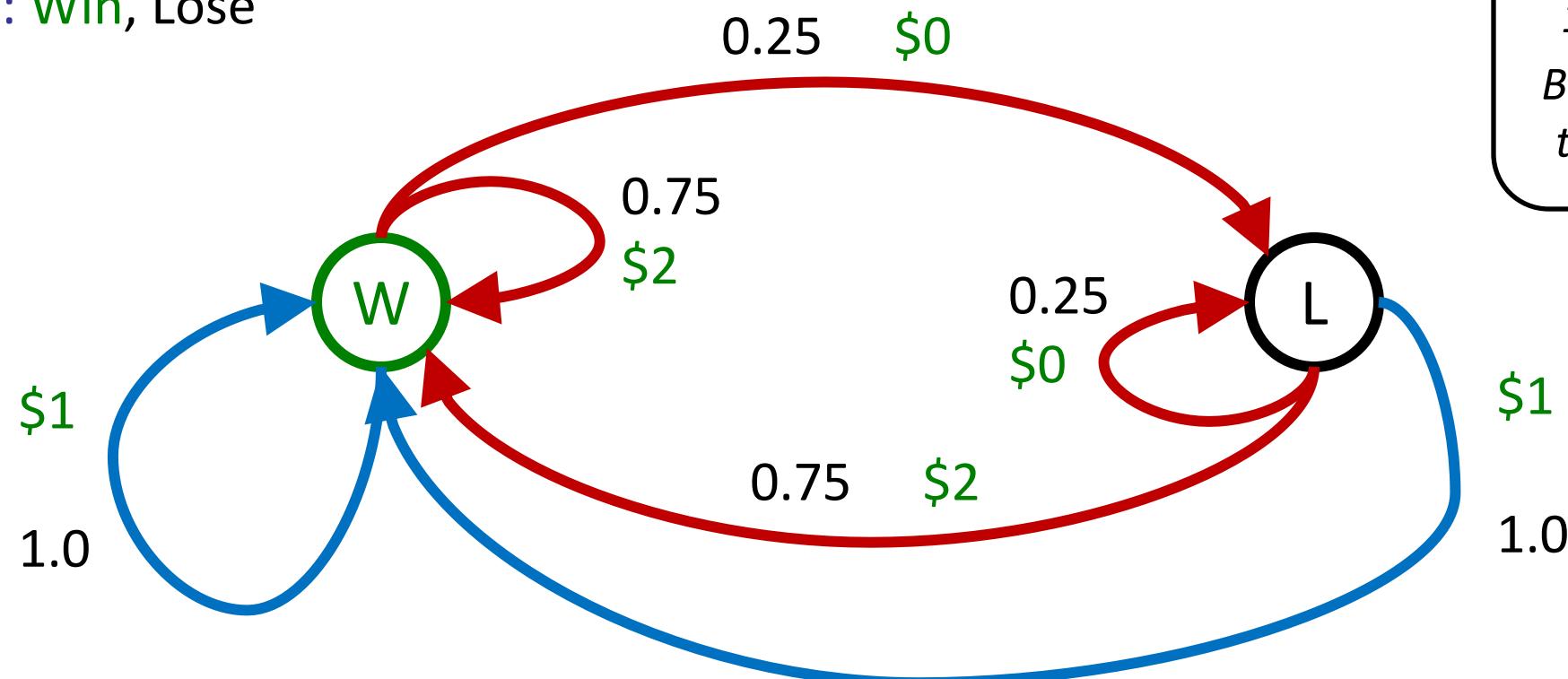


Double Bandits



Double-Bandit MDP

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



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

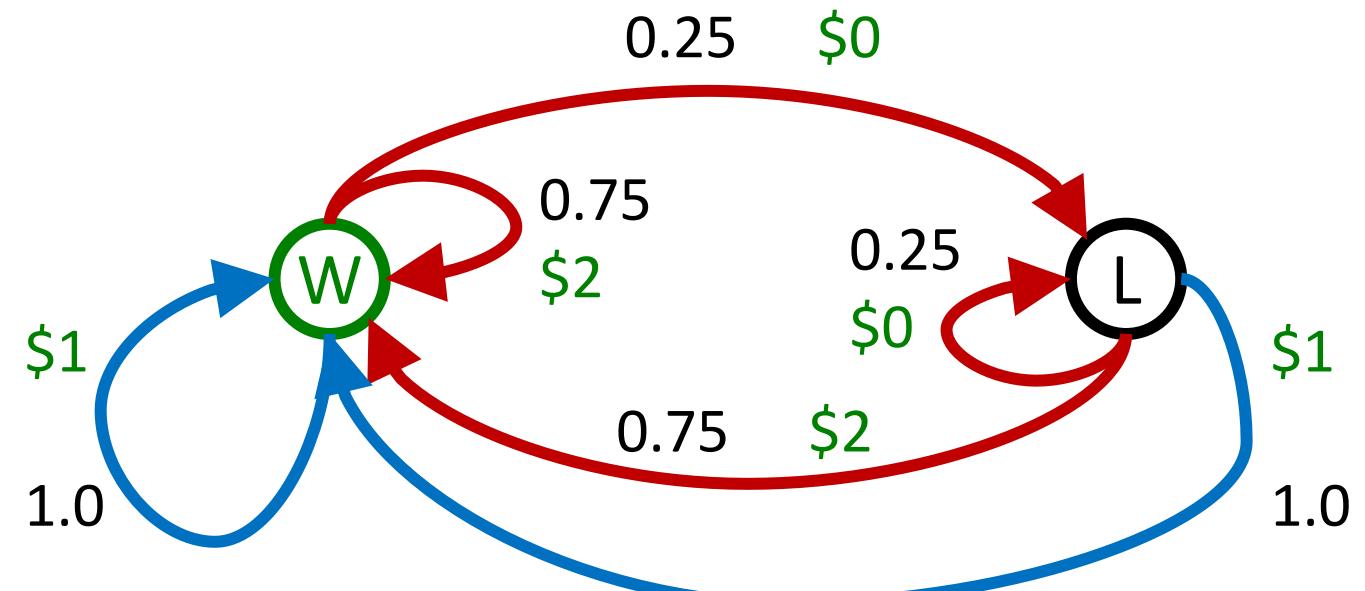
Offline Planning

- Solving MDPs is offline planning

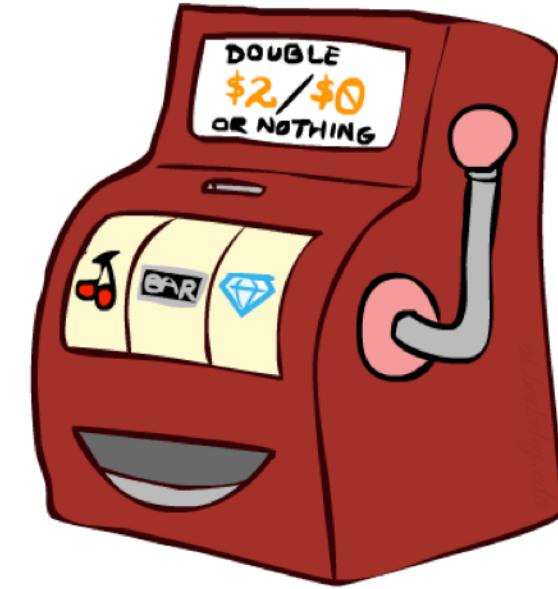
- You determine all quantities through computation
- You need to know the details of the MDP
- You do not actually play the game!

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

	Value
Play Red	150
Play Blue	100



Let's Play!

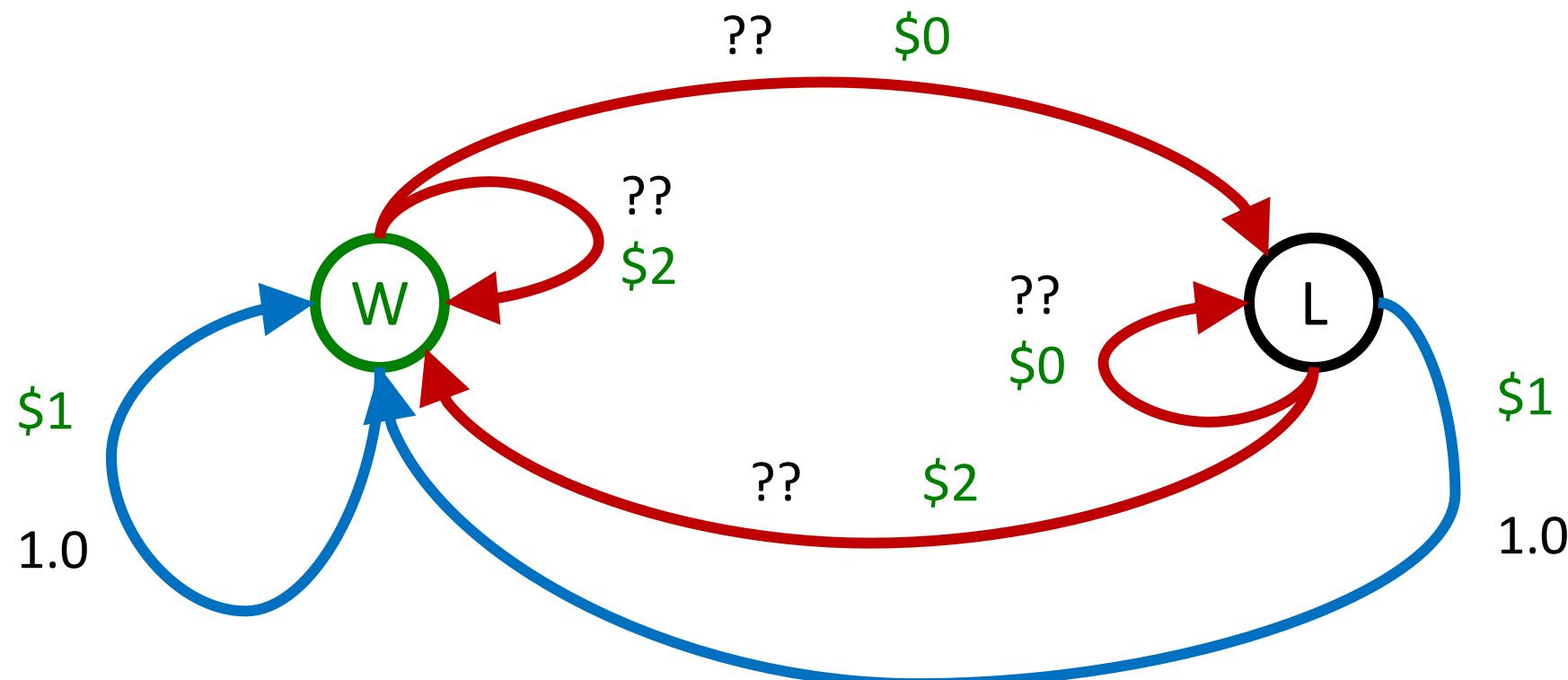


\$2 \$2 \$0 \$2 \$2

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

Online Planning

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



Let's Play!



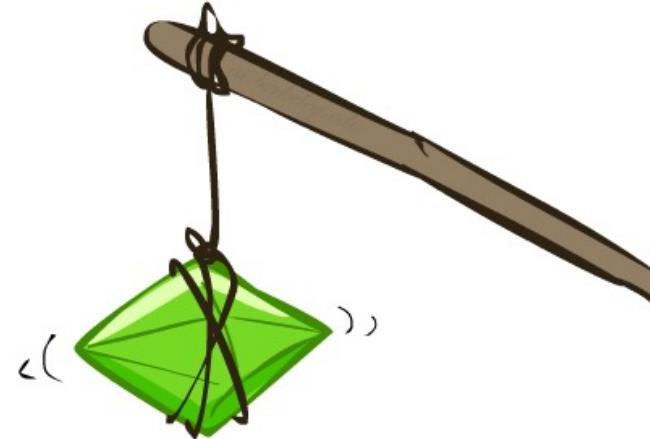
\$0	\$0	\$0	\$2	\$0
\$2	\$0	\$0	\$0	\$0

What Just Happened?

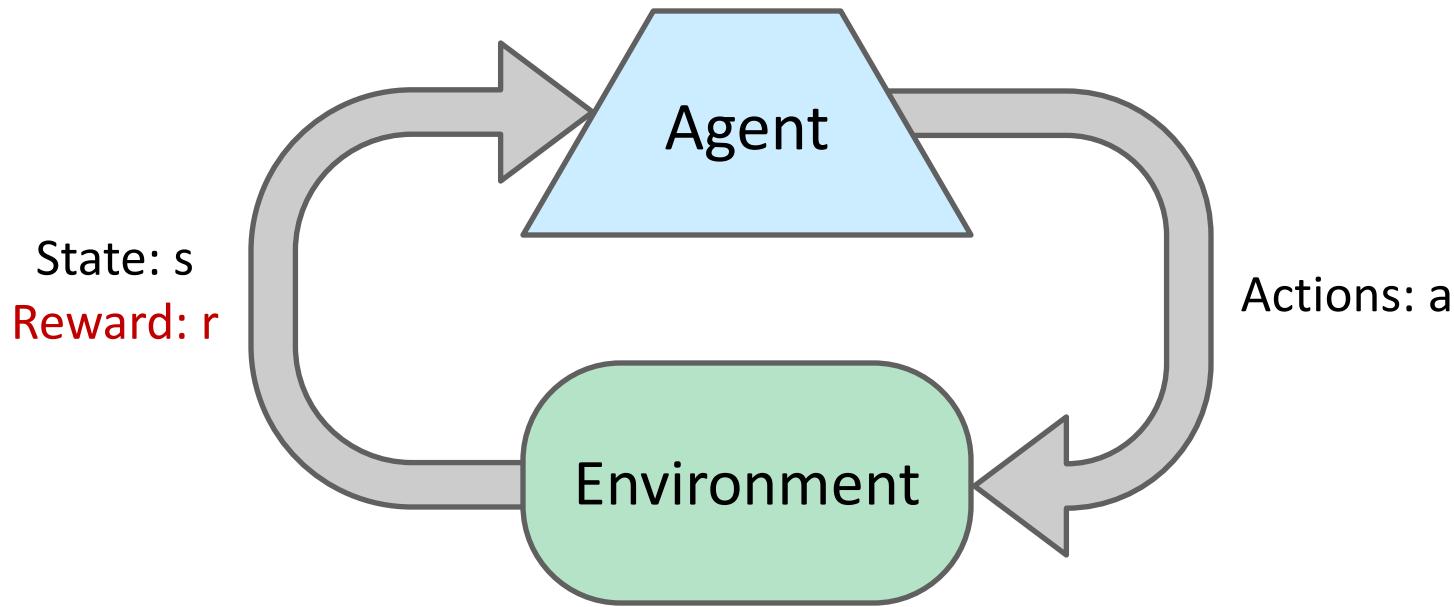
- That wasn't planning, it was learning!
 - Specifically, reinforcement learning
 - There was an MDP, but you couldn't solve it with just computation
 - You needed to actually act to figure it out
- Important ideas in reinforcement learning that came up
 - Exploration: you have to try unknown actions to get information
 - Exploitation: eventually, you have to use what you know
 - Regret: even if you learn intelligently, you make mistakes
 - Sampling: because of chance, you have to try things repeatedly
 - Difficulty: learning can be much harder than solving a known MDP



Reinforcement Learning



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!

Example: Learning to Walk



Initial



A Learning Trial



After Learning [1K Trials]

Example: Learning to Walk



Initial

Example: Learning to Walk



Training

[Kohl and Stone, ICRA 2004]

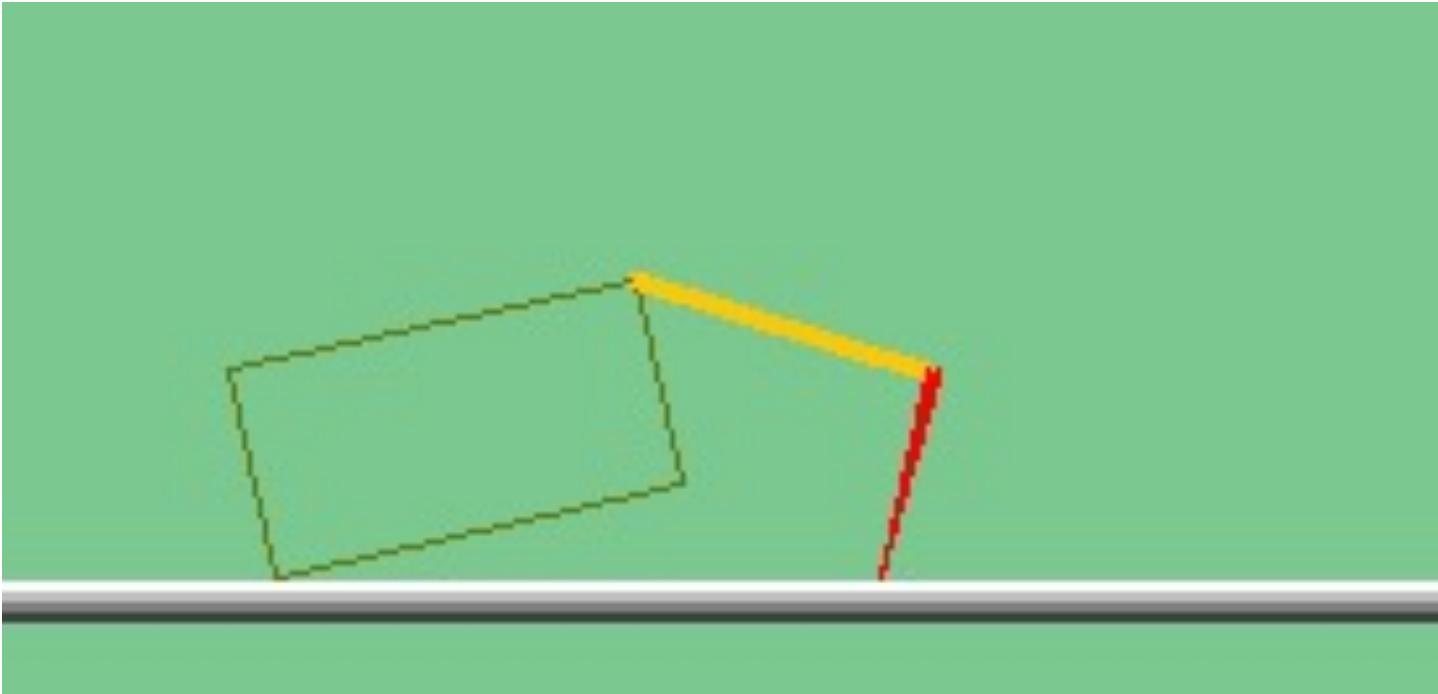
[Video: AIBO WALK – training]

Example: Learning to Walk



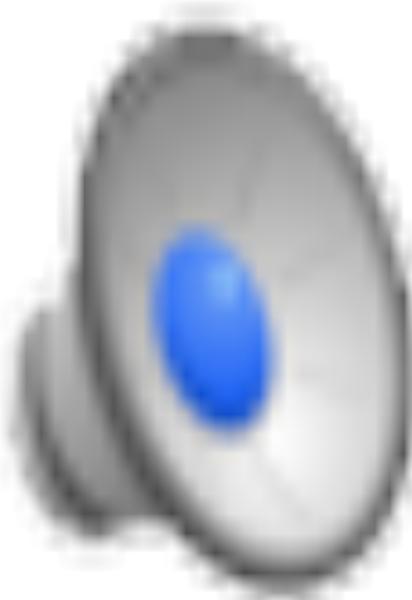
Finished

The Crawler!



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

Video of Demo Crawler Bot

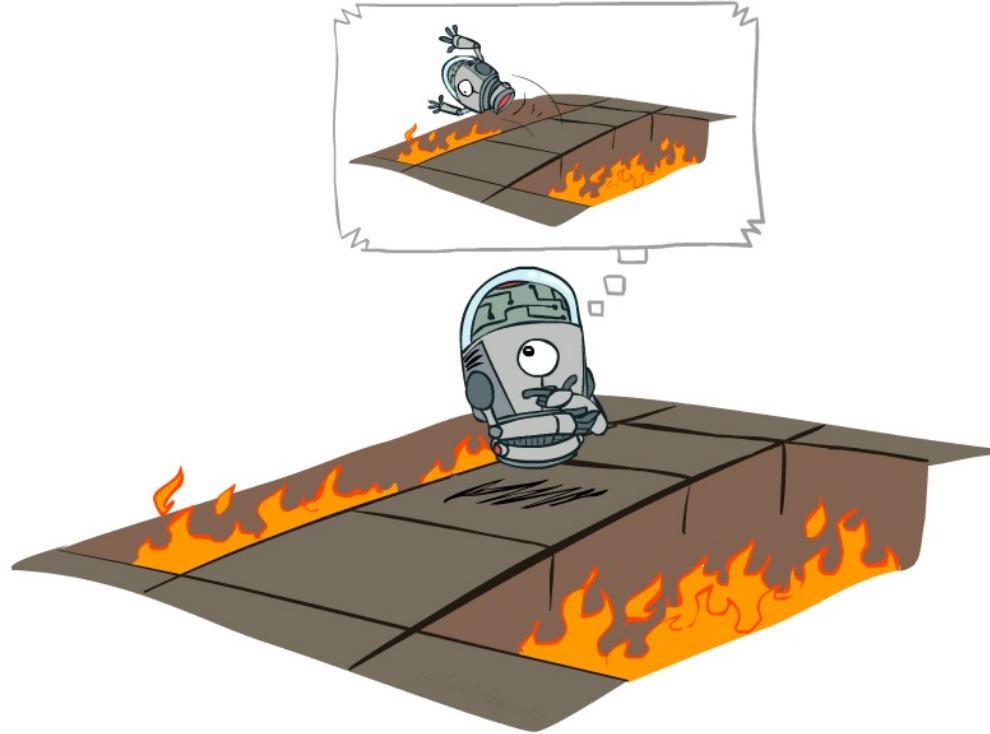


Reinforcement Learning

- 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 $\pi(s)$
- 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



Offline (MDPs) vs. Online (RL)

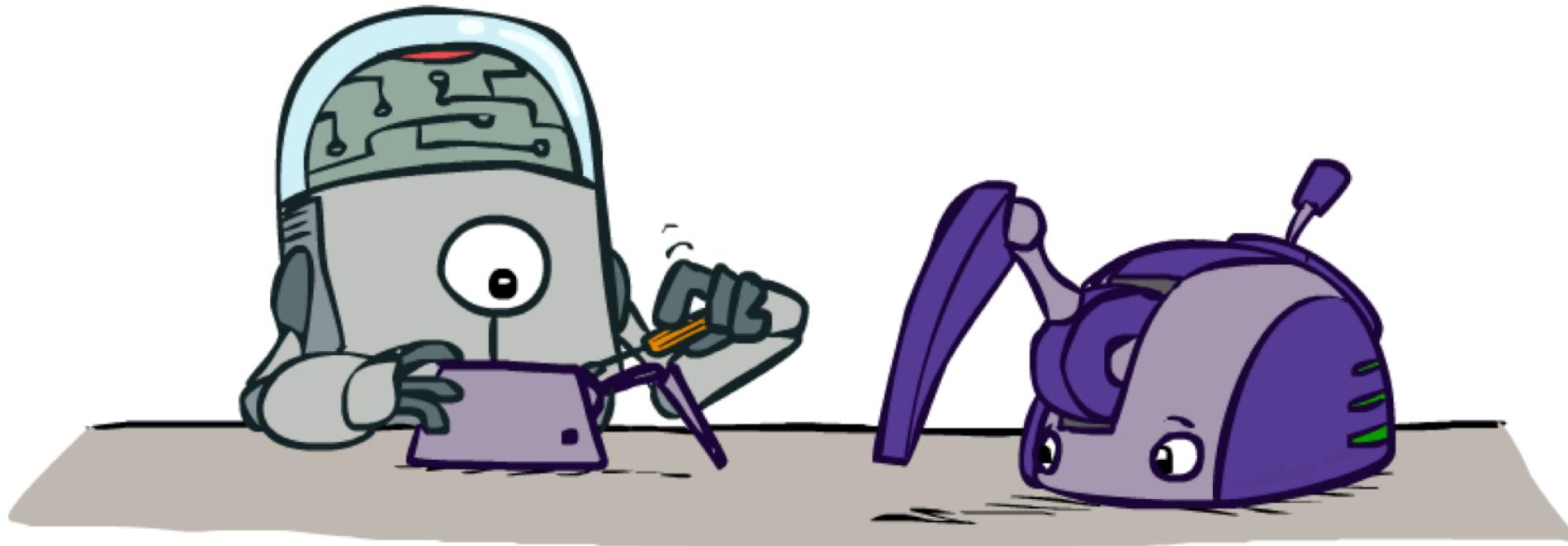


Offline Solution



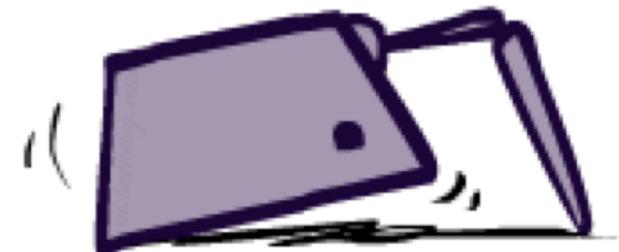
Online Learning

Model-Based Learning



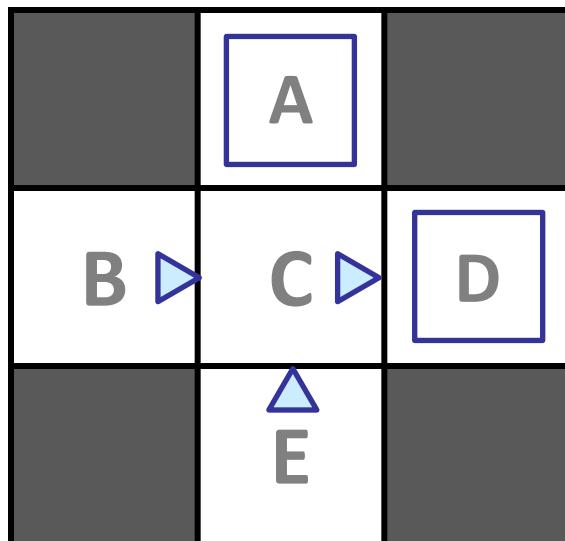
Model-Based Learning

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

Learned Model

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

$T(B, \text{east}, C) = 1.00$
 $T(C, \text{east}, D) = 0.75$
 $T(C, \text{east}, A) = 0.25$
...

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

$R(B, \text{east}, C) = -1$
 $R(C, \text{east}, D) = -1$
 $R(D, \text{exit}, x) = +10$
...

Example: Expected Age

Goal: Compute expected age of 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”

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

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

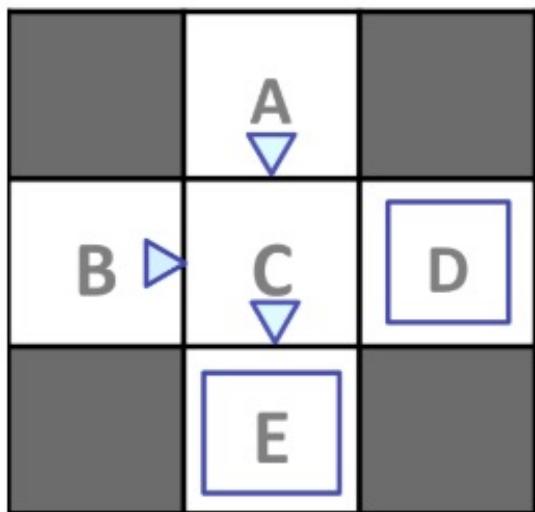
$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

Unknown $P(A)$: “Model Free”

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

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

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

A, south, C, -1
C, south, E, -1
E, exit, x, +10

Episode 2

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

Episode 3

B, east, C, -1
C, south, E, -1
E, exit, x, +10

Episode 4

A, south, C, -1
C, south, E, -1
E, exit, x, +10

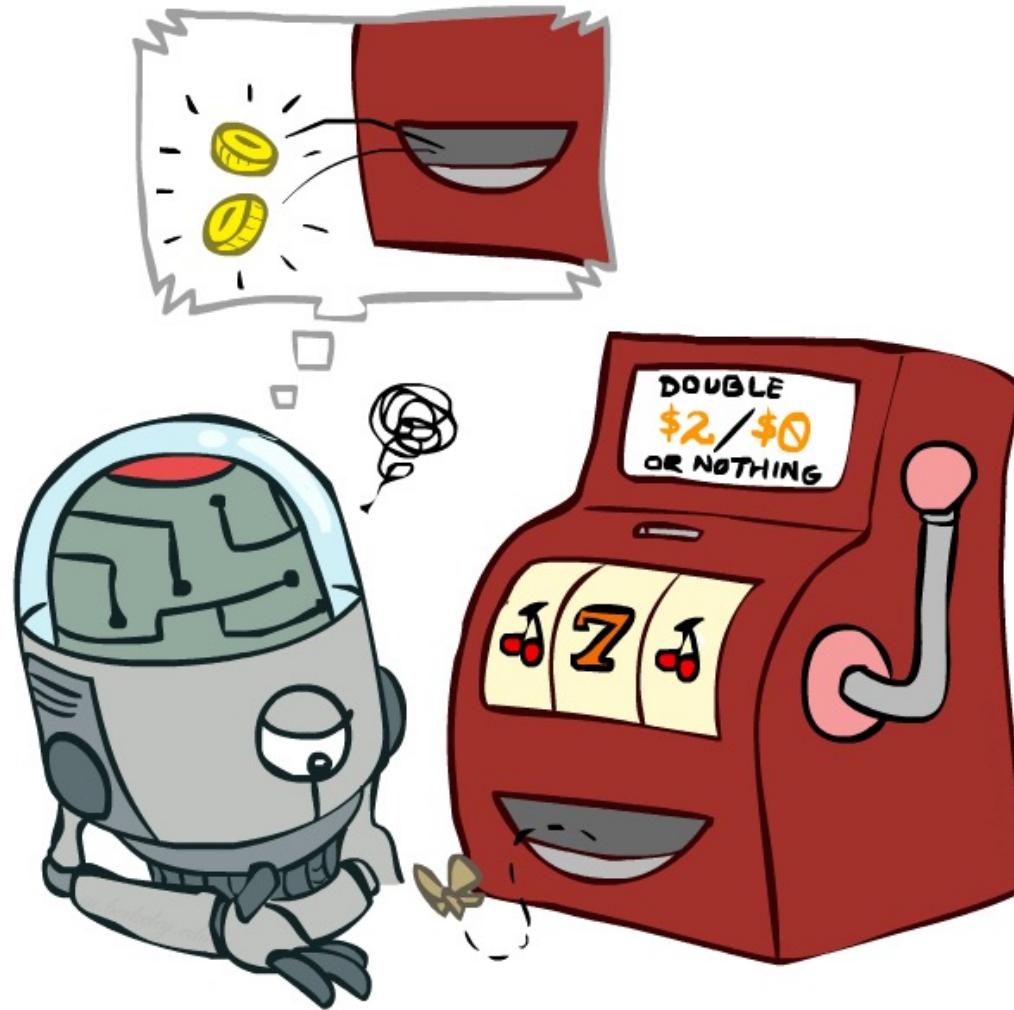
$T(A, \text{south}, C) =$

$T(B, \text{east}, C) =$

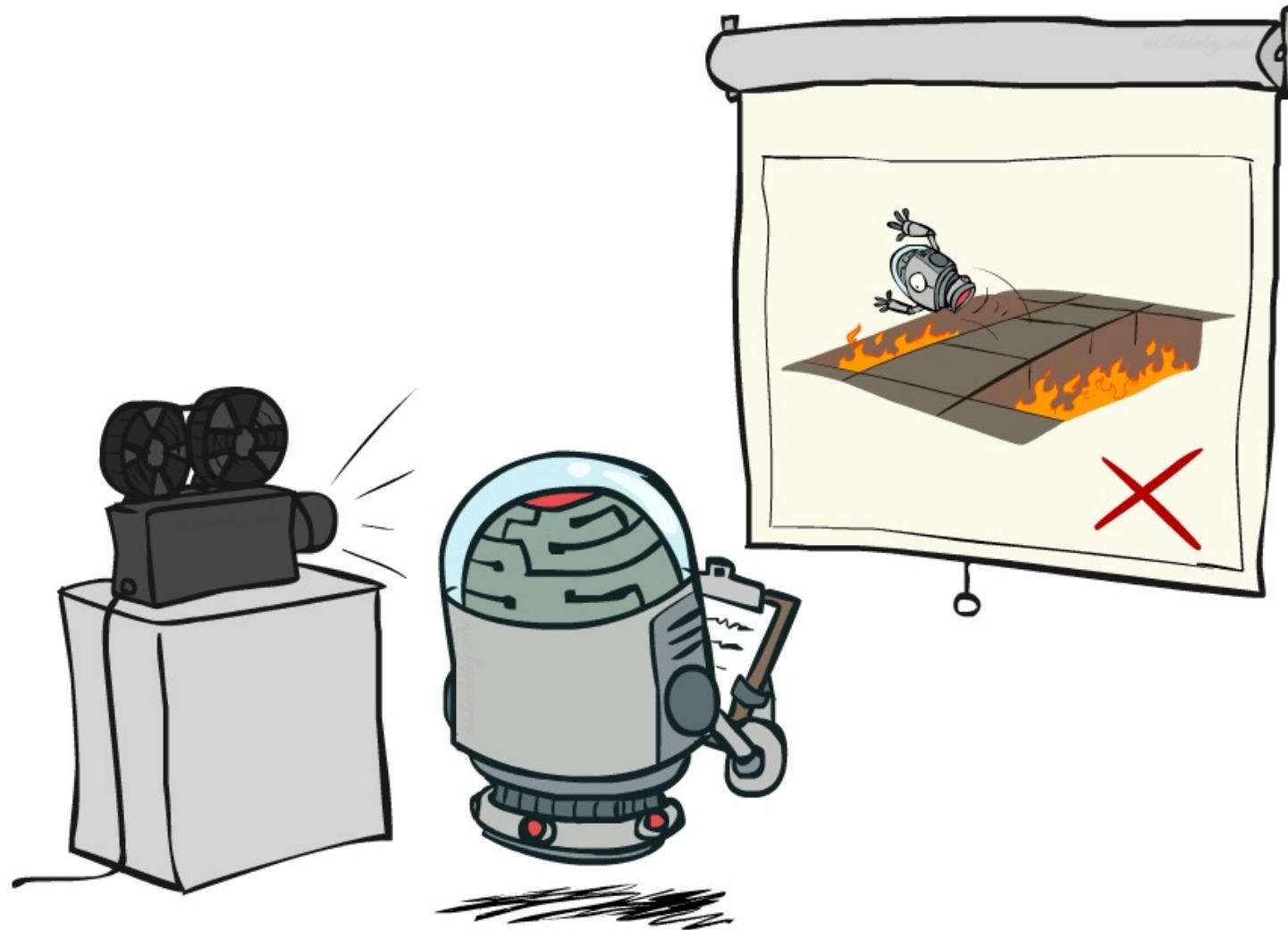
$T(C, \text{south}, E) =$

$T(C, \text{south}, D) =$

Model-Free Learning



Passive Reinforcement Learning



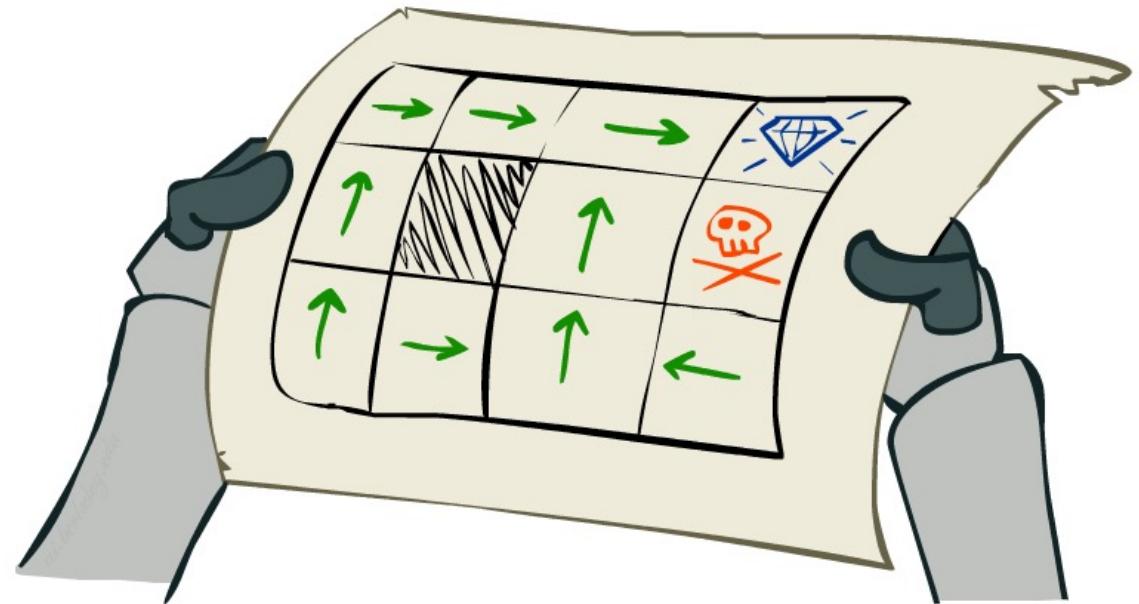
Passive Reinforcement Learning

- 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

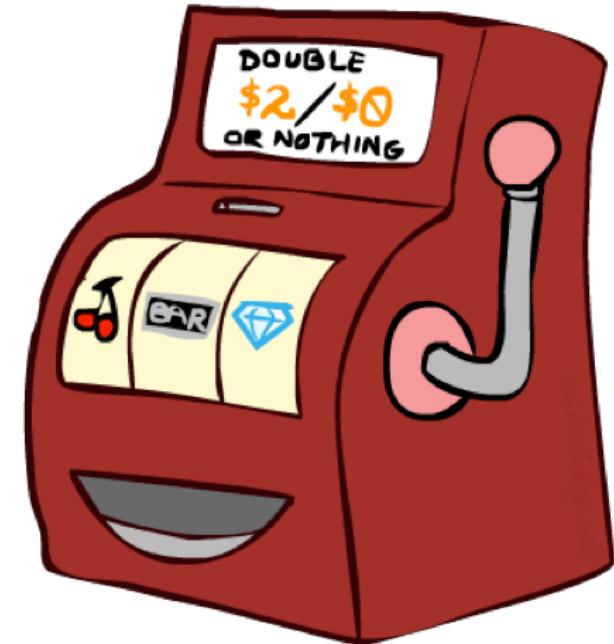
- In this case:

- 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.



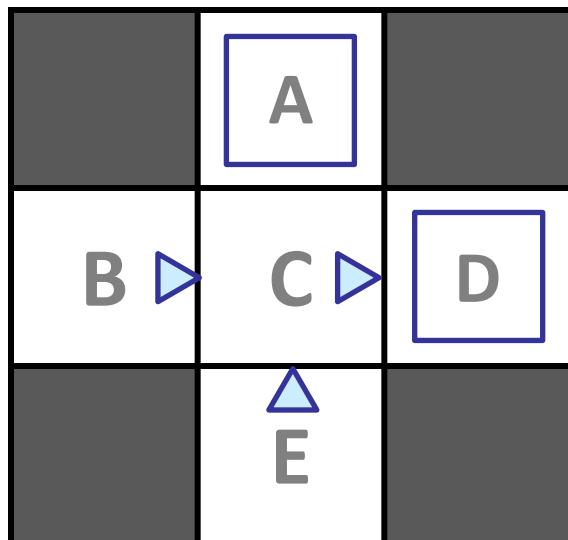
Direct Evaluation

- 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 IN THE END
 - 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

	-10	
A	+4	+10
B	+8	
C		D
	-2	
E		

$$\hat{V}^{\pi}(A) =$$

8

$$\hat{V}^{\pi}(B) =$$

-2

$$\hat{V}^{\pi}(C) =$$

4

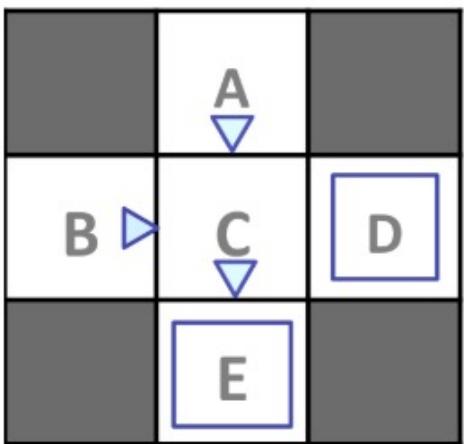
$$\hat{V}^{\pi}(D) =$$

-10

$$\hat{V}^{\pi}(E) =$$

10

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

A, south, C, -1
C, south, E, -1
E, exit, x, +10

Episode 2

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

Episode 3

B, east, C, -1
C, south, E, -1
E, exit, x, +10

Episode 4

A, south, C, -1
C, south, E, -1
E, exit, x, +10

Problems with Direct Evaluation

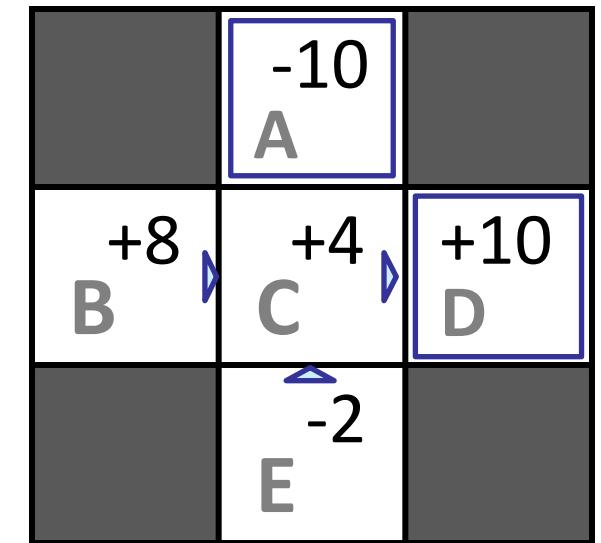
- What's good about direct evaluation?

- 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

Output Values



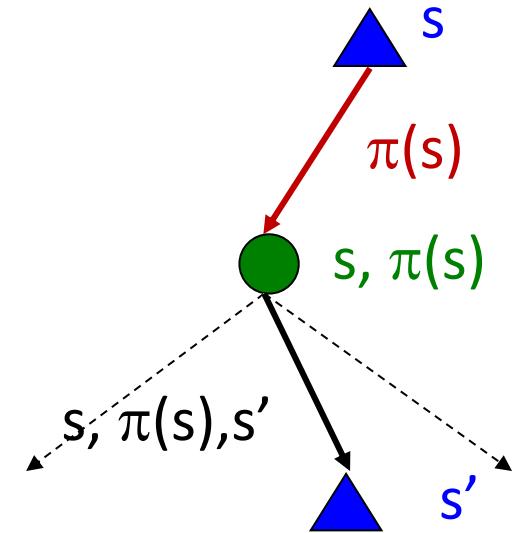
If B and E both go to C under this policy, how can their values be different?

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

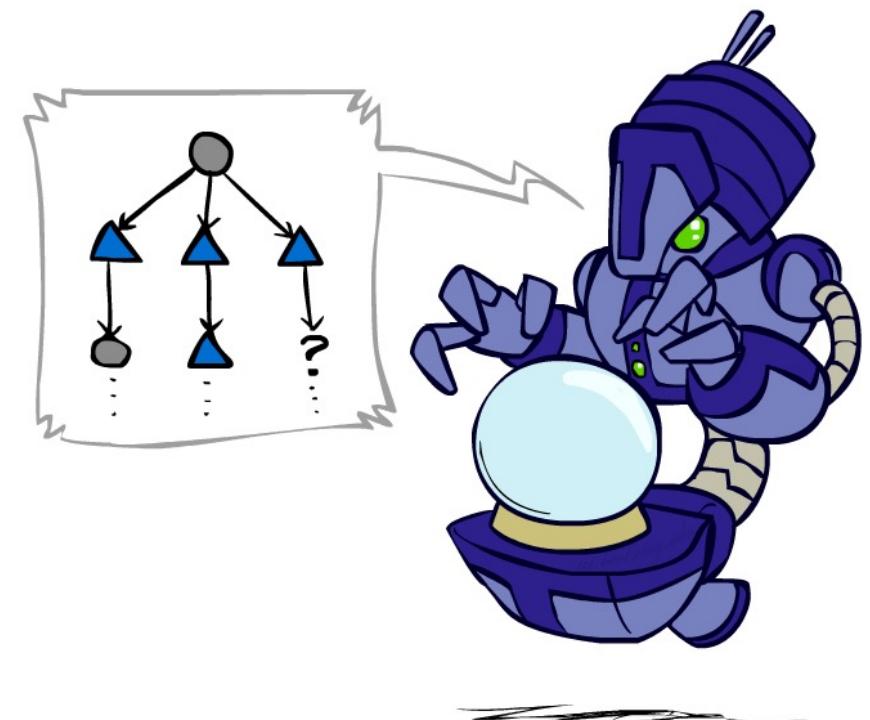
$$sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1)$$

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

...

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

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



So far: Reinforcement Learning

- We still assume an 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 , so must try out actions
- Big idea: Compute all averages over T using sample outcomes



The Story So Far: MDPs and RL

Known MDP: Offline Solution

Goal

Compute V^* , Q^* , π^*

Evaluate a fixed policy π

Technique

Value / policy iteration

Policy evaluation

Unknown MDP: Model-Based

Goal

Compute V^* , Q^* , π^*

Evaluate a fixed policy π

Technique

VI/PI on approx. MDP

PE on approx. MDP

Unknown MDP: Model-Free

Goal

Compute V^* , Q^* , π^*

Evaluate a fixed policy π

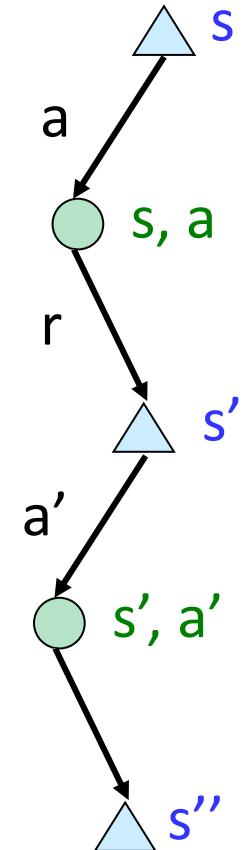
Technique

Q-learning

Temporal Difference (TD)
Value Learning

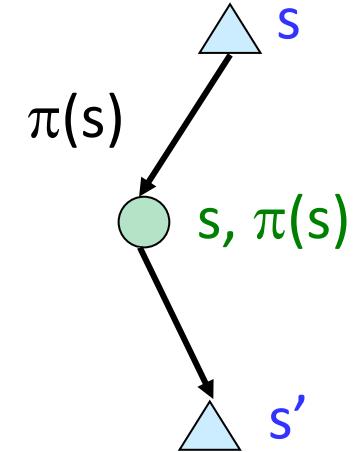
Model-Free Learning

- Model-free (temporal difference) learning
 - Experience world through episodes
$$(s, a, r, s', a', r', s'', a'', r'', s''', \dots)$$
 - Update estimates each transition (s, a, r, s')
 - Over time, updates will mimic Bellman updates



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



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

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

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

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

Example: Temporal Difference Learning

States

	A	
B	C	D
	E	

Observed Transitions

B, east, C, -2

	0	
0	0	8
	0	

C, east, D, -2

	0	
-1	0	8
	0	

	0	
-1	3	8
	0	

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

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

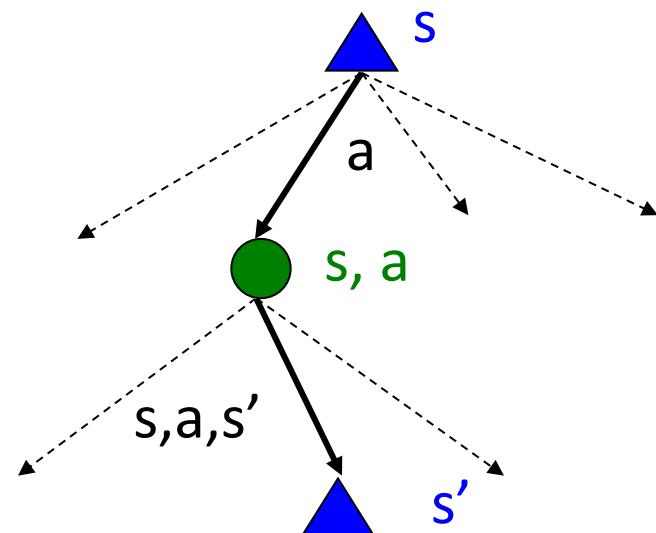
Problems with TD Value Learning

- TD value learning 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:

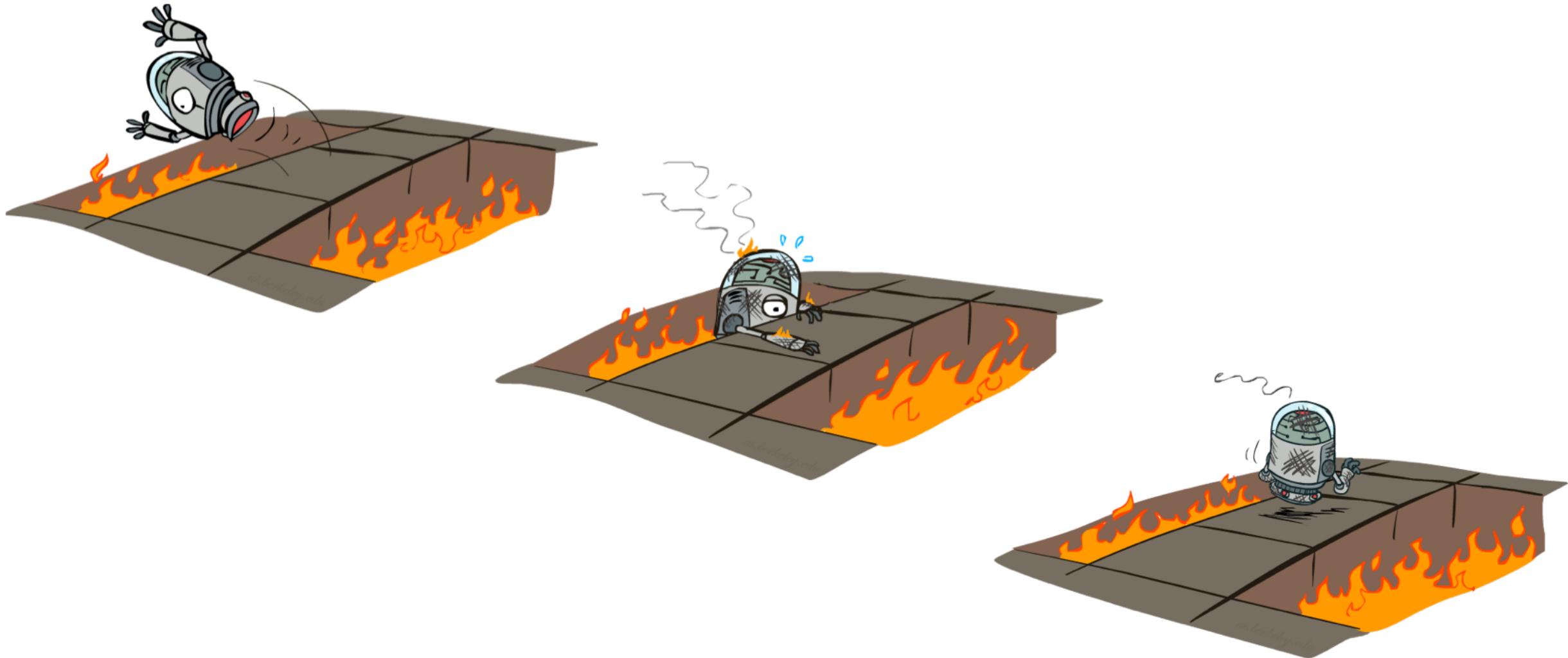
$$\pi(s) = \arg \max_a Q(s, a)$$

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

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

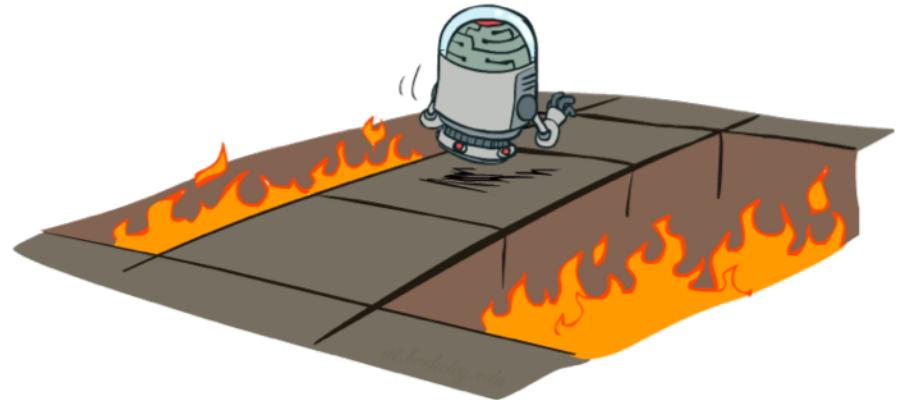


Active Reinforcement Learning



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...



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:

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

- 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:

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

Q-Learning

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

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

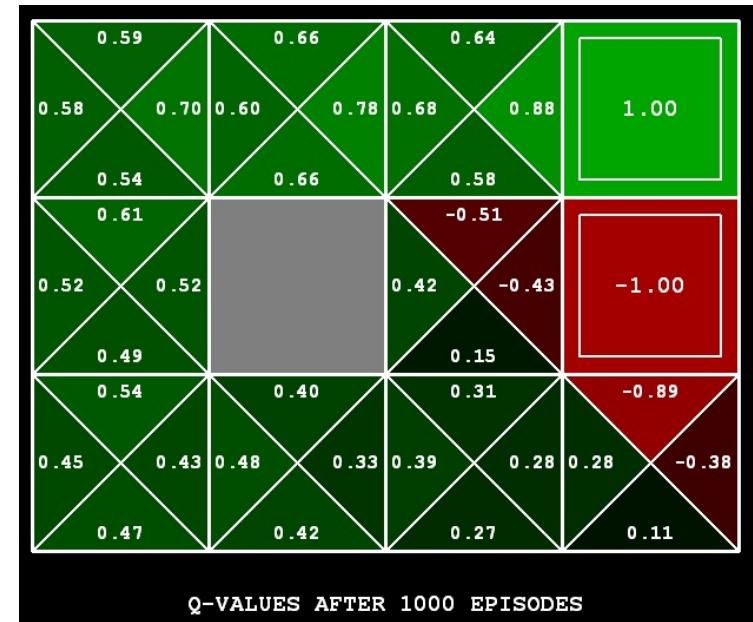
- Learn $Q(s, a)$ values as you go

- 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:

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



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

Video of Demo Q-Learning -- Gridworld



Q-Learning

- We'd like to do Q-value updates to each Q-state:

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

- But can't compute this update without knowing T , R
- Instead, compute average as we go
 - Receive a sample transition (s, a, r, s')
 - This sample suggests

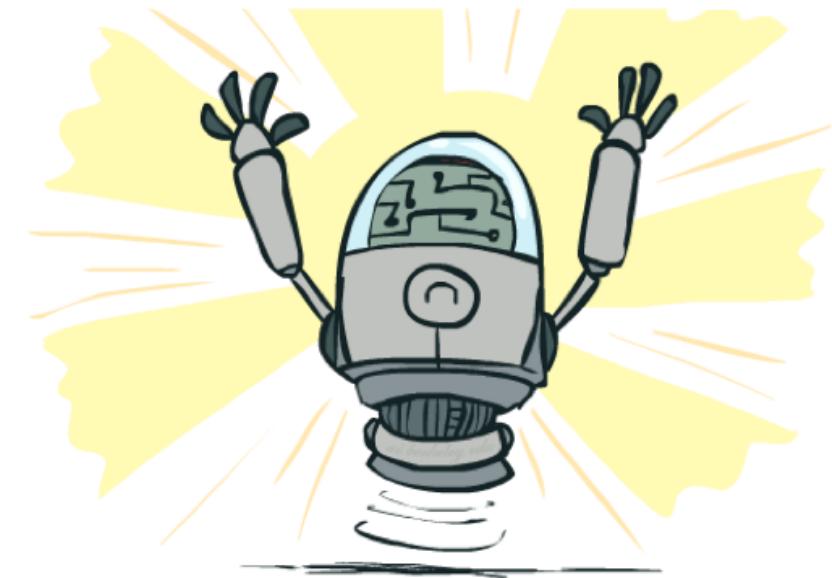
$$Q(s, a) \approx r + \gamma \max_{a'} Q(s', a')$$

- But we want to average over results from (s, a) (Why?)
- So keep a running average

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha) \left[r + \gamma \max_{a'} Q(s', a') \right]$$

Q-Learning Properties

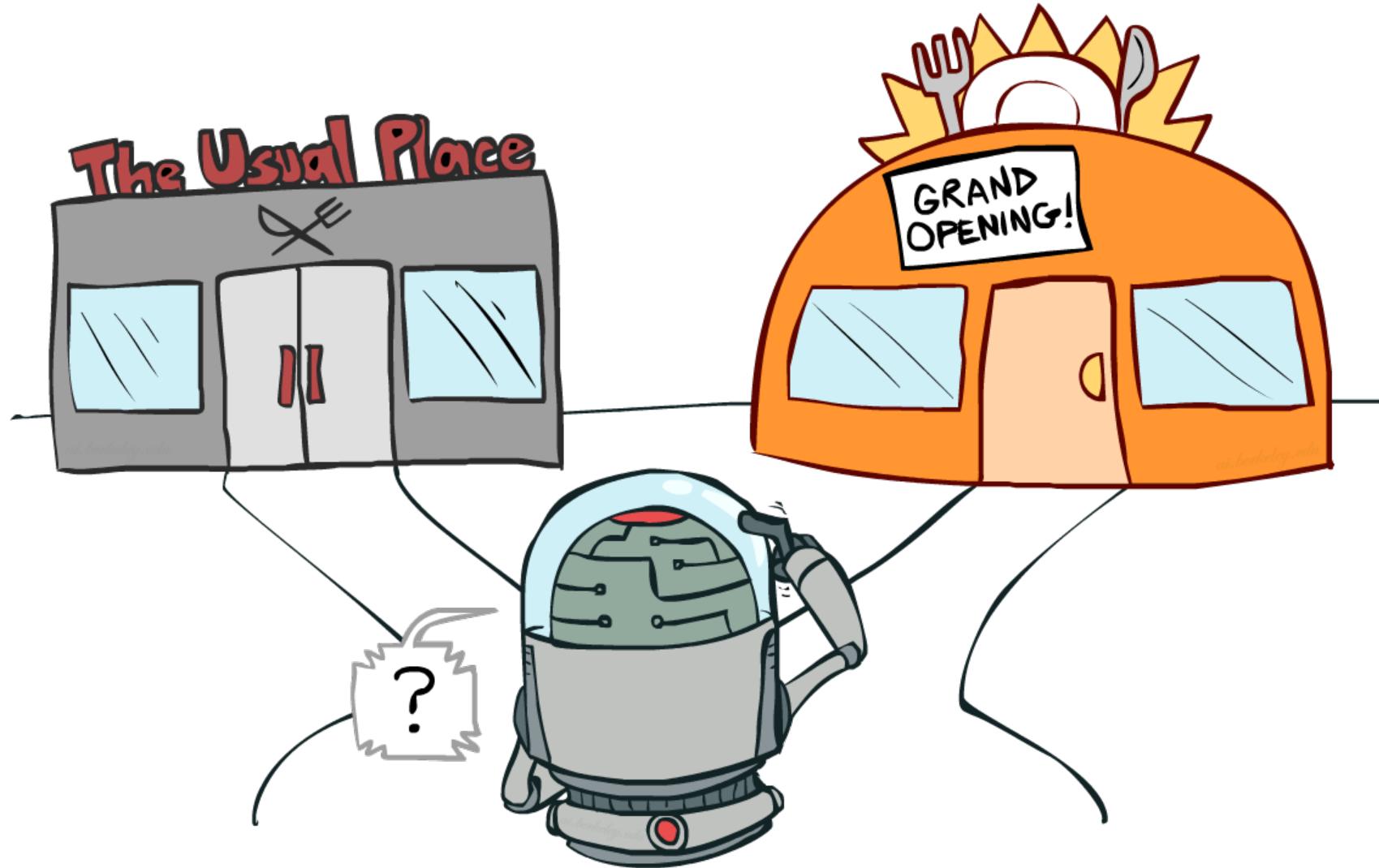
- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called **off-policy learning**
- Caveats:
 - You have to explore enough
 - You have to eventually make the learning rate small enough
 - ... but not decrease it too quickly
 - Basically, in the limit, it doesn't matter how you select actions (!)



Video of Demo Q-Learning Auto Cliff Grid



Exploration vs. Exploitation



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 $1-\varepsilon$, 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 ε over time
 - Another solution: exploration functions

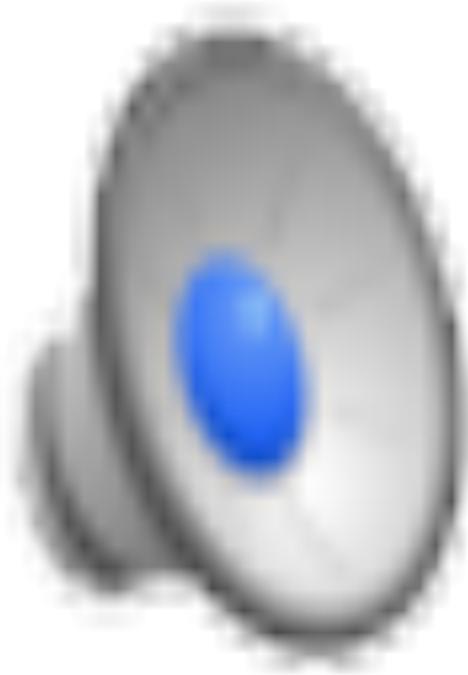


[Demo: Q-learning – manual exploration – bridge grid (L11D2)]
[Demo: Q-learning – epsilon-greedy -- crawler (L11D3)]

Video of Demo Q-learning – Manual Exploration – Bridge Grid



Video of Demo Q-learning – Epsilon-Greedy – Crawler



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

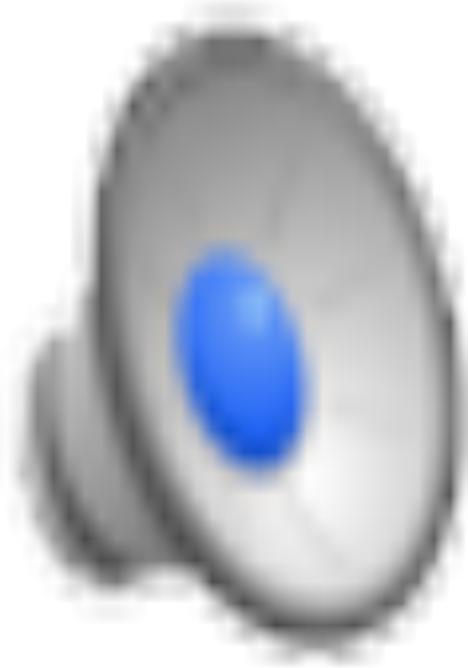
Regular Q-Update:
$$Q(s, a) \leftarrow_{\alpha} R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

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



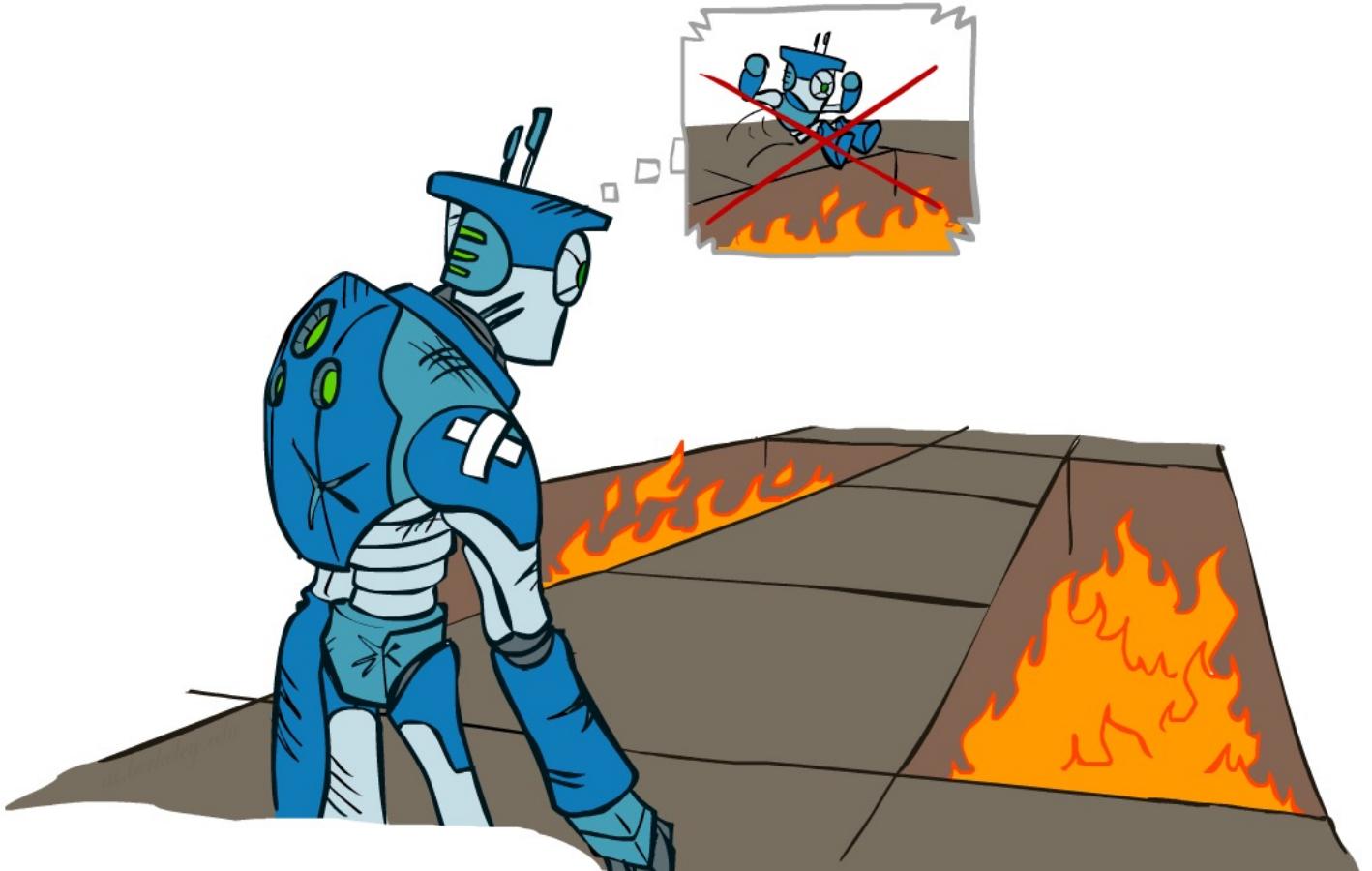
- Note: this propagates the “bonus” back to states that lead to unknown states as well!

Video of Demo Q-learning – Exploration Function – Crawler

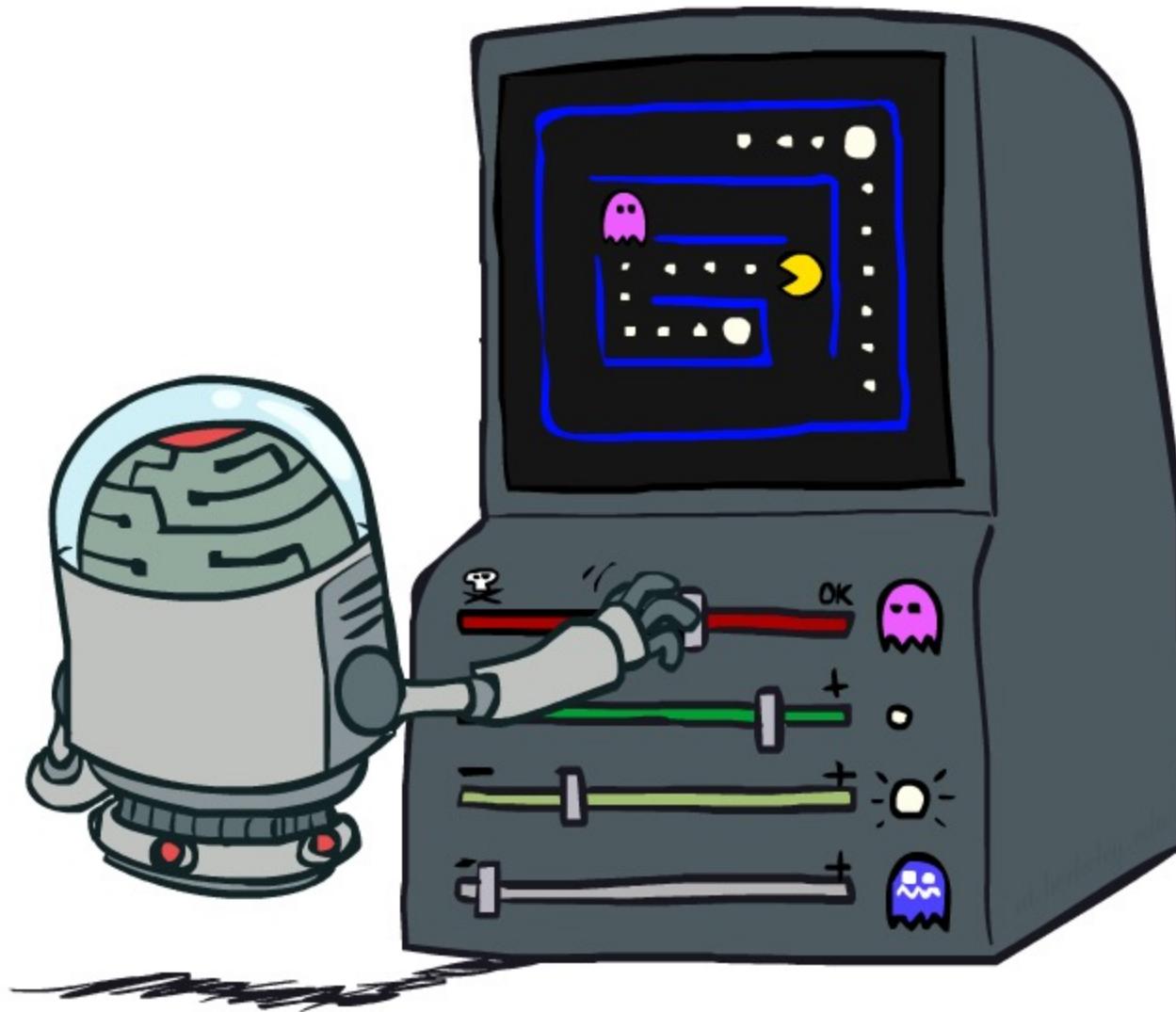


Regret

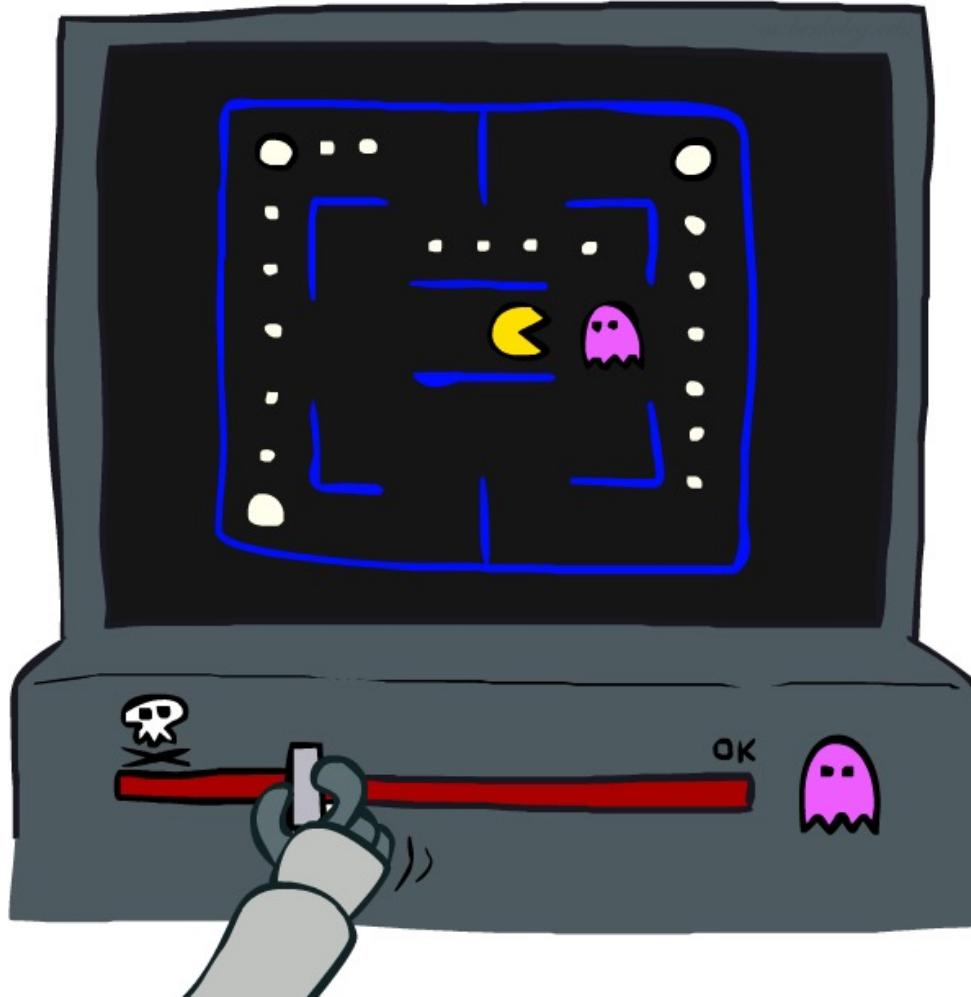
- Even if you learn the optimal policy, you still make mistakes along the way
- Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal – it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



Approximate Q-Learning

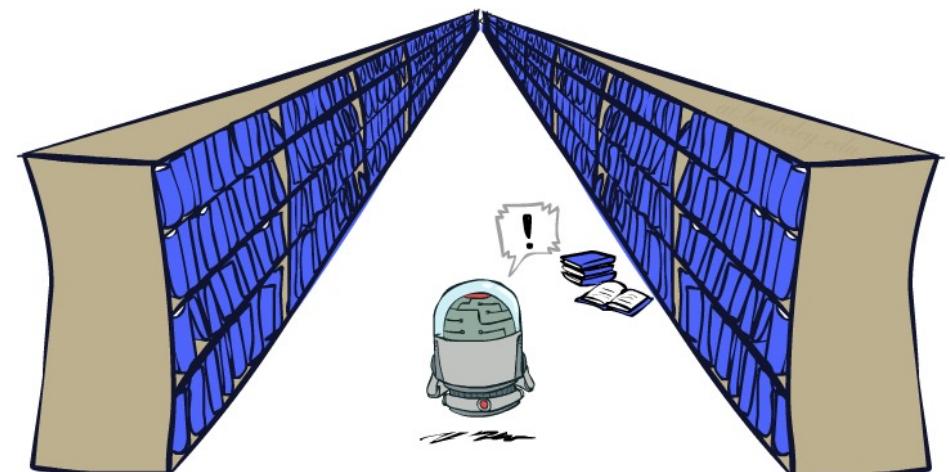
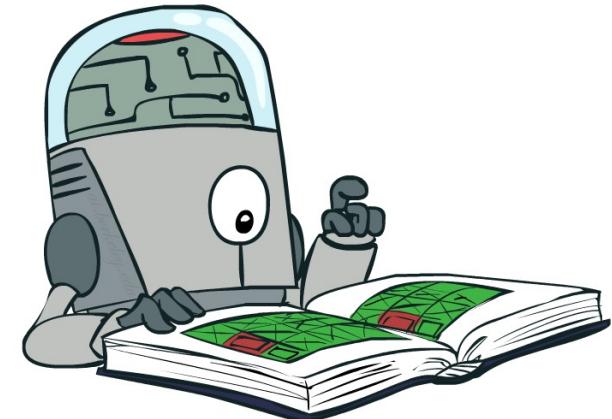


Approximate Q-Learning



Generalizing Across States

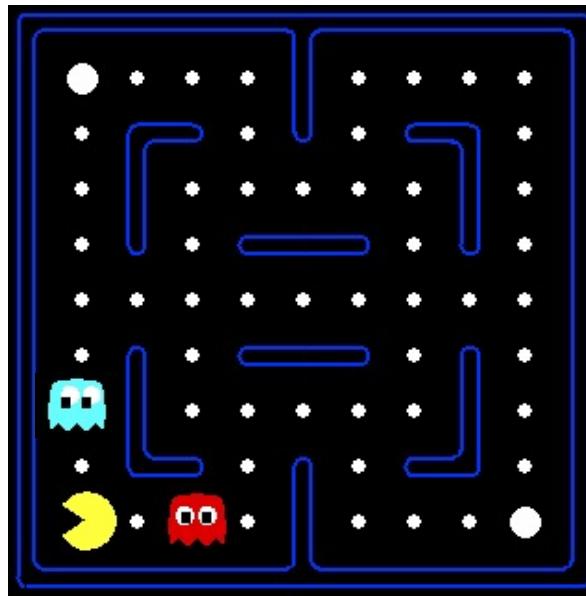
- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
 - Too many states to visit them all in training
 - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
 - Learn about some small number of training states from experience
 - Generalize that experience to new, similar situations
 - This is a fundamental idea in machine learning, and we'll see it over and over again



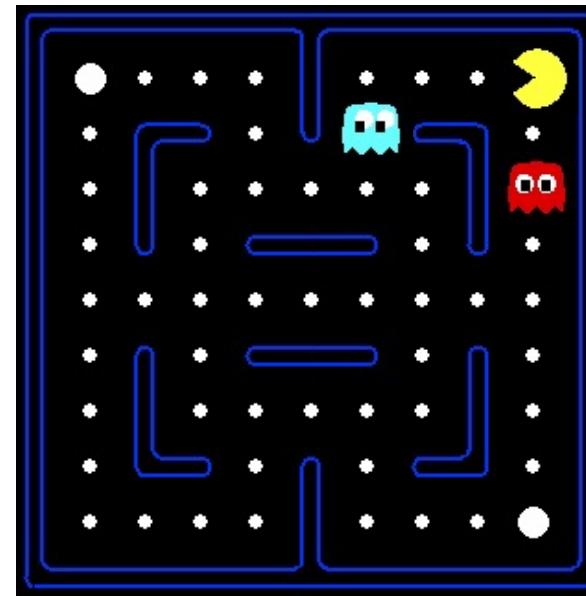
[demo – RL pacman]

Example: Pacman

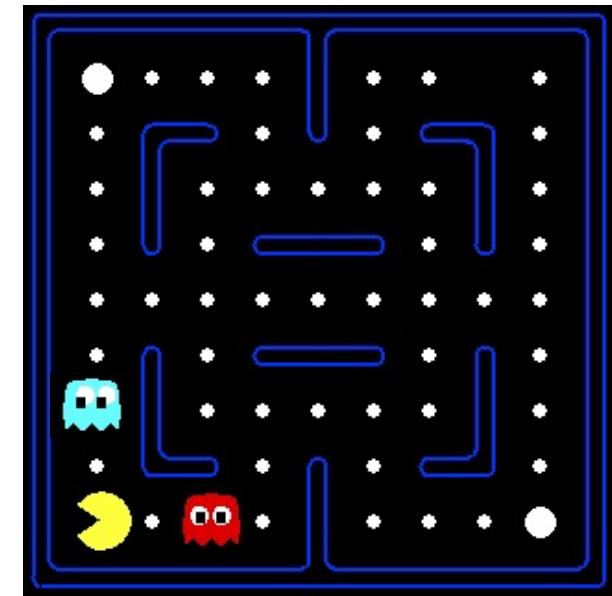
Let's say we discover through experience that this state is bad:



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



Or even this one!



[Demo: Q-learning – pacman – tiny – watch all (L11D5)]
[Demo: Q-learning – pacman – tiny – silent train (L11D6)]
[Demo: Q-learning – pacman – tricky – watch all (L11D7)]

Video of Demo Q-Learning Pacman – Tiny – Watch All



Video of Demo Q-Learning Pacman – Tiny – Silent Train



Video of Demo Q-Learning Pacman – Tricky – Watch All



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')

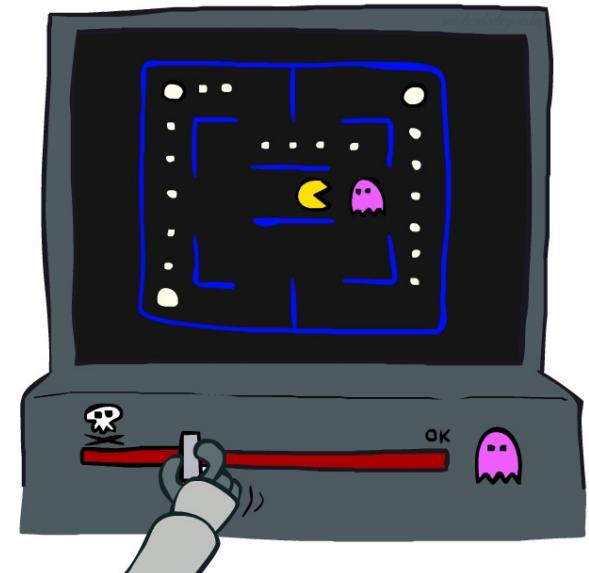
$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a)$$

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

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

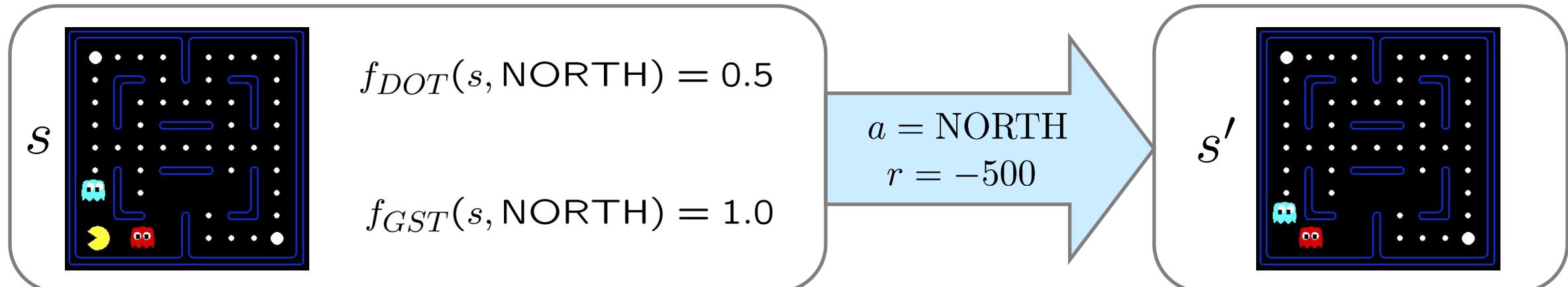
- Intuitive interpretation:

- Adjust weights of active features
- E.g., if something unexpectedly bad happens, blame the features that were on:
disprefer all states with that state's features



Example: Q-Pacman

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



difference = -501

$$\begin{aligned} w_{DOT} &\leftarrow 4.0 + \alpha [-501] 0.5 \\ w_{GST} &\leftarrow -1.0 + \alpha [-501] 1.0 \end{aligned}$$

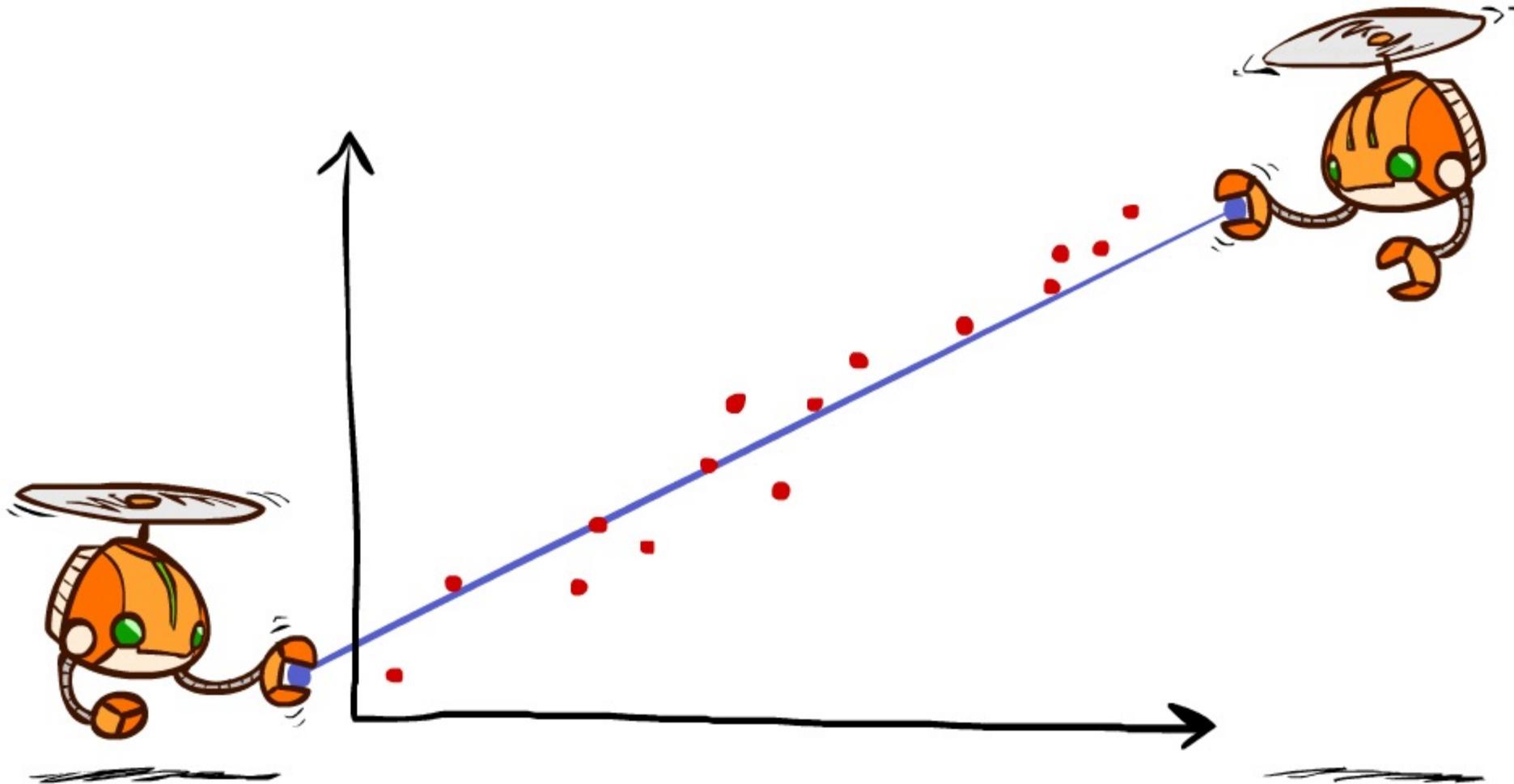
$$Q(s, a) = 3.0f_{DOT}(s, a) - 3.0f_{GST}(s, a)$$

[Demo: approximate Q-learning pacman (L11D10)]

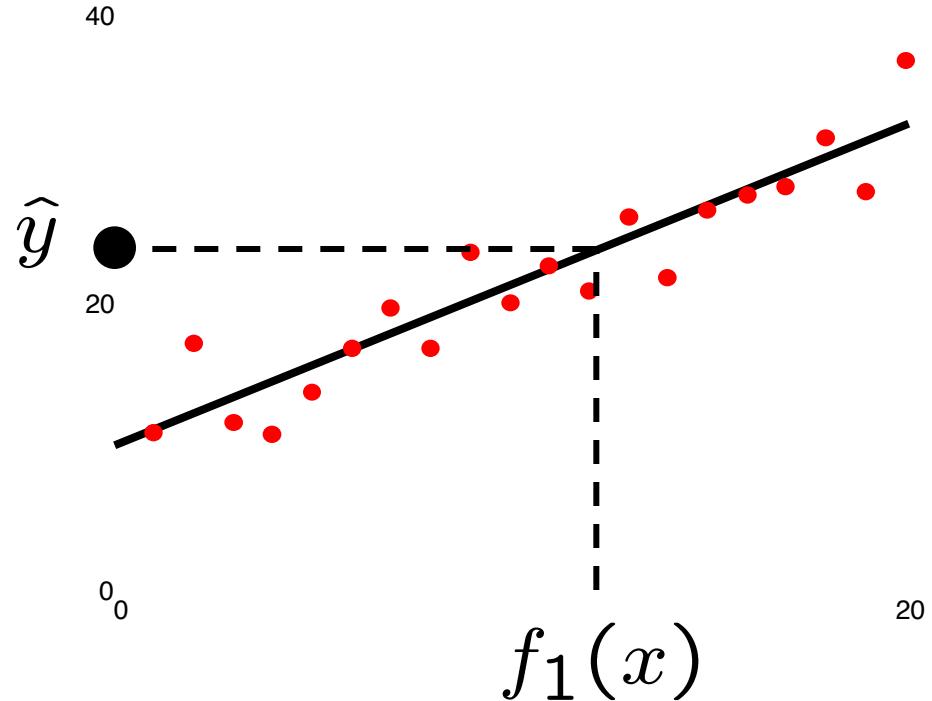
Video of Demo Approximate Q-Learning -- Pacman



Q-Learning and Least Squares

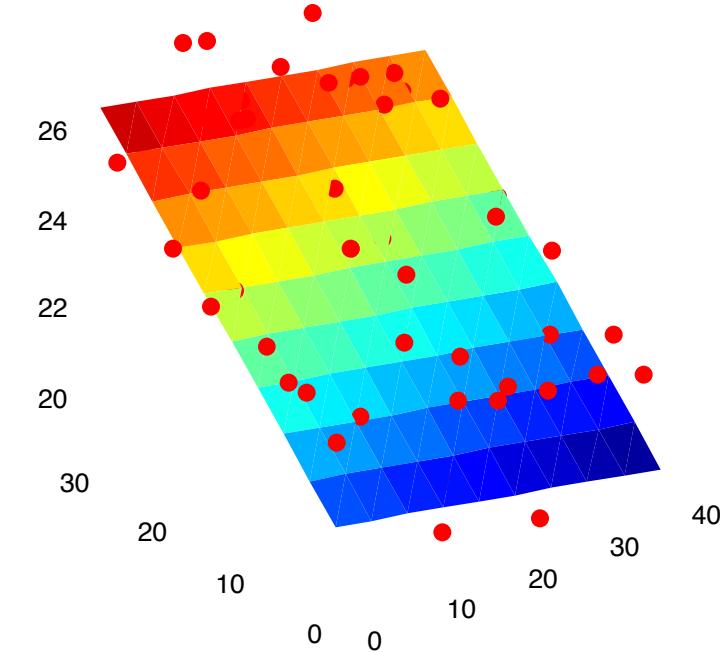


Linear Approximation: Regression*



Prediction:

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

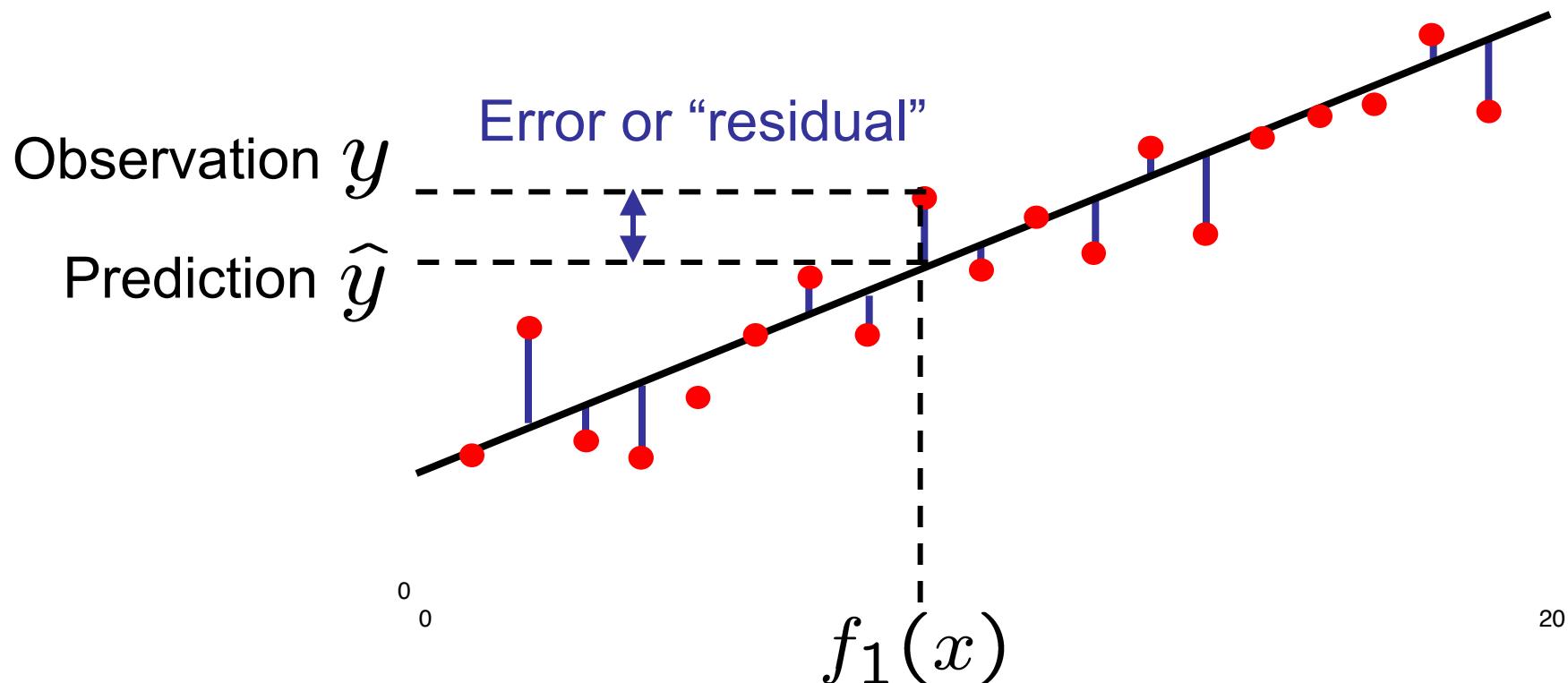


Prediction:

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

Optimization: Least Squares*

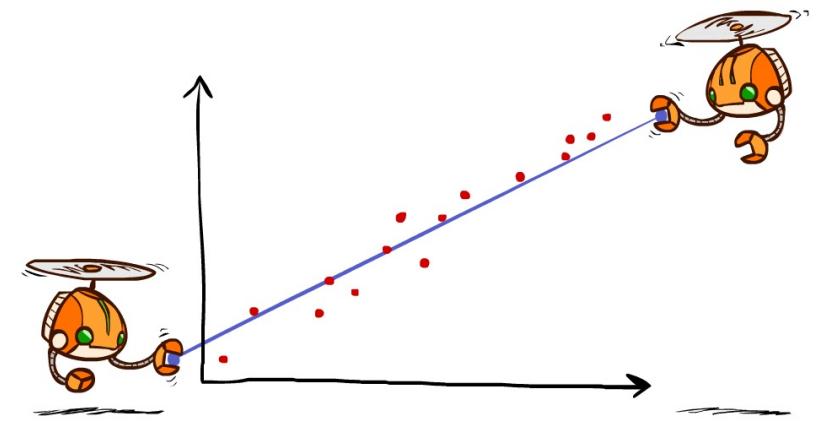
$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left(y_i - \sum_k w_k f_k(x_i) \right)^2$$



Minimizing Error*

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

$$\text{error}(w) = \frac{1}{2} \left(y - \sum_k w_k f_k(x) \right)^2$$
$$\frac{\partial \text{error}(w)}{\partial w_m} = - \left(y - \sum_k w_k f_k(x) \right) f_m(x)$$
$$w_m \leftarrow w_m + \alpha \left(y - \sum_k w_k f_k(x) \right) f_m(x)$$



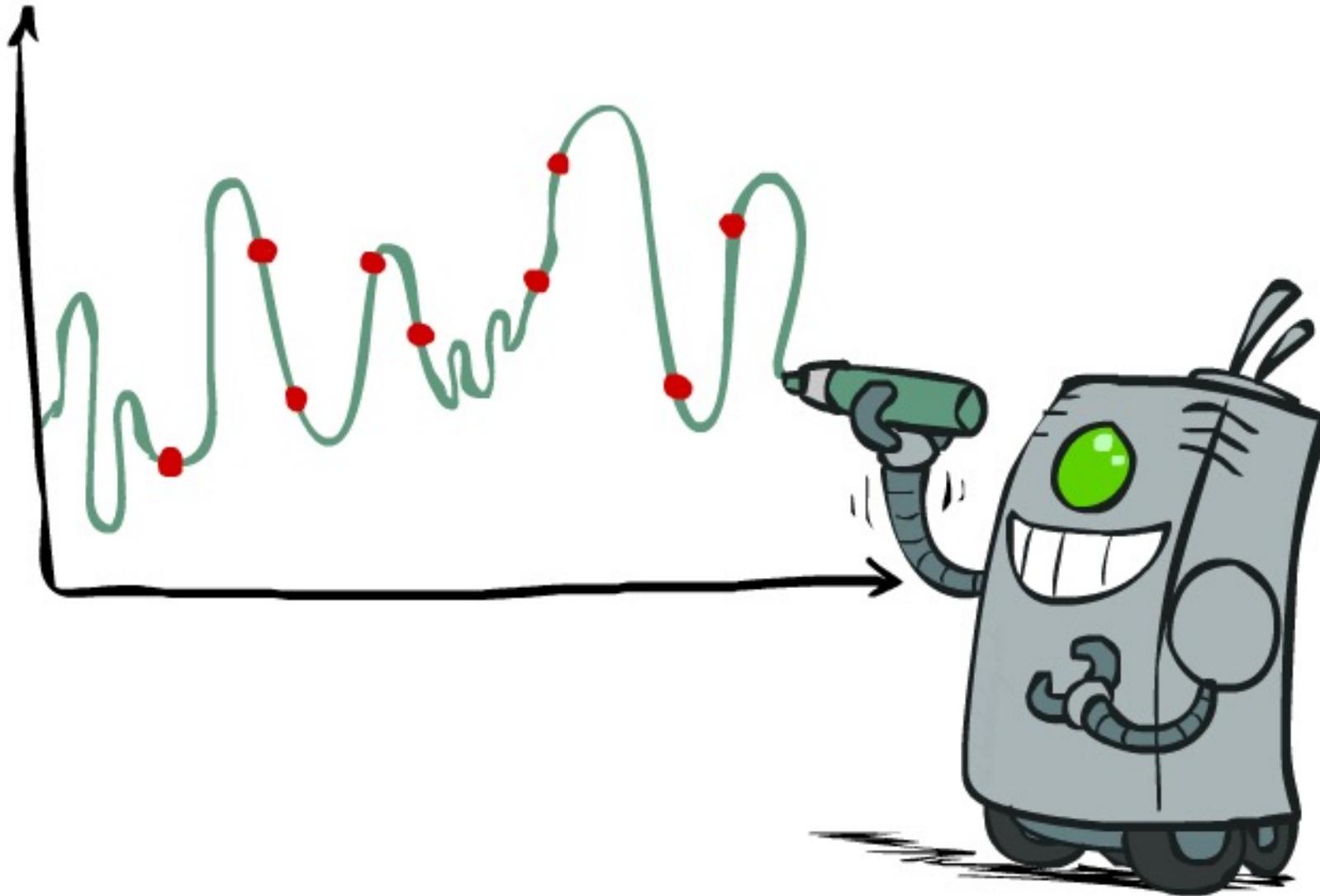
Approximate q update explained:

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

“target”

“prediction”

Overfitting: Why Limiting Capacity Can Help*



$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a)$$

Quiz

$f_1(s, a) = 1/\text{(distance to nearest dot after having executed action } a \text{ in state } s)$

$f_2(s, a) = \text{(distance to nearest ghost after having executed action } a \text{ in state } s)$

Assume $w_1 = 1$, $w_2 = 10$. For the state s shown below, find the following quantities. Assume that the red and blue ghosts are both sitting on top of a dot.



$$Q(s, West) =$$

$$Q(s, South) =$$

Based on this approximate Q-function, which action would be chosen:

West

South

$$Q(s', \text{West}) =$$

$$Q(s', \text{East}) =$$

The reward for this transition is $r = +10 - 1 = 9$

+10: for food pellet eating, -1 for time passed



What is the sample value (assuming $\gamma = 1$)?

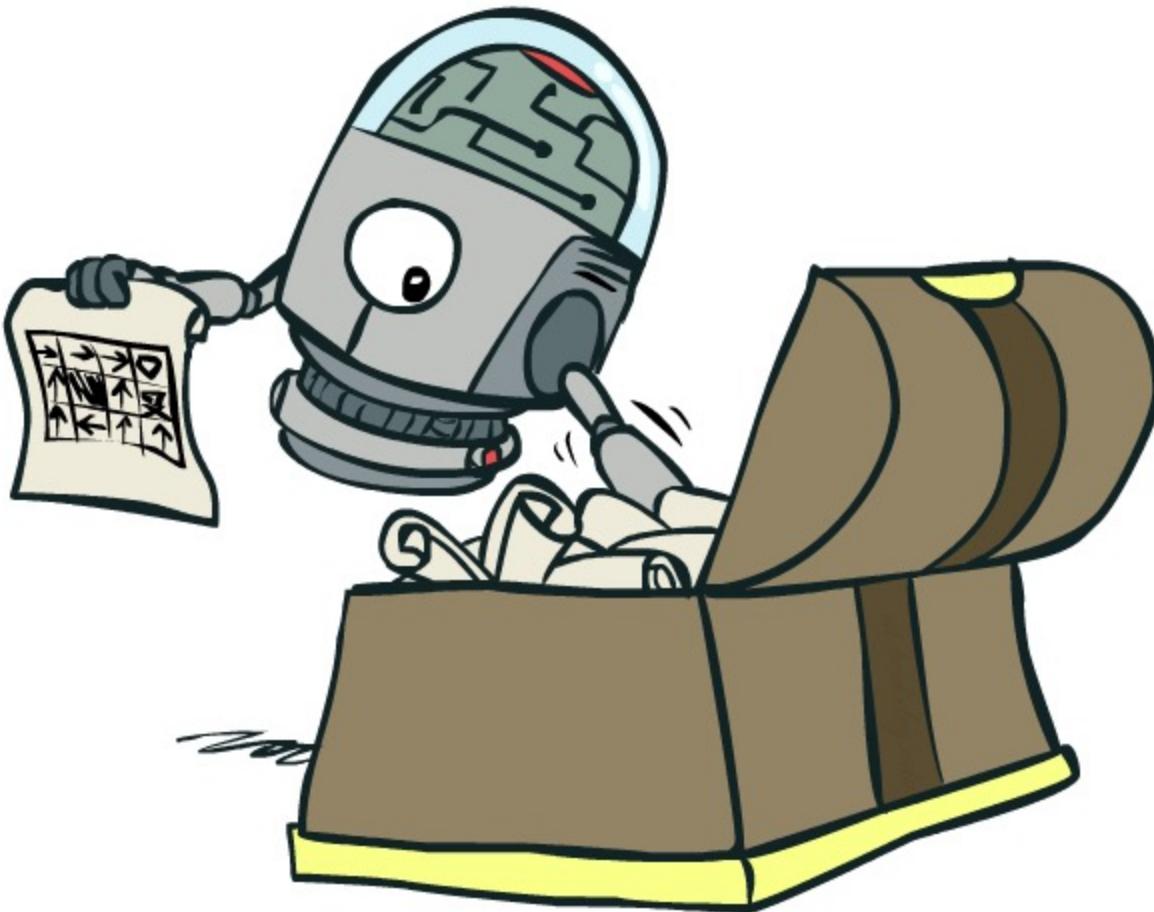
Now let's compute the update to the weights. Let $\alpha = 0.5$.

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

$$w_1 \leftarrow w_1 + \alpha(\text{difference}) f_1(s, a) =$$

$$w_2 \leftarrow w_2 + \alpha(\text{difference}) f_2(s, a) =$$

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)
 - We'll see this distinction between modeling and prediction again later in the course
- 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
 - Nudge 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 lookahead structure, sample wisely, change multiple parameters...

Policy Search





Conclusion

- We're done with Part I: Search and Planning!
- We've seen how AI methods can solve problems in:
 - Search
 - Constraint Satisfaction Problems
 - Games
 - Markov Decision Problems
 - Reinforcement Learning
- Next: Part II: Uncertainty and Learning!



Watch this!

- AlphaGo:The Movie
- <https://youtu.be/WXuK6gekU1Y>
- Go - Basic Rules
- <https://youtu.be/5PTXdR8hLIQ>



Read the Paper

- Mastering the game of Go with deep neural networks and tree search
 - by David Silver et al.
 - Link: <https://www.nature.com/articles/nature16961>
 - Also shared on Google Classroom ([1_Mastering the game of Go.pdf](#))

Neural Networks

- Neural networks by 3Blue1Brown
 - Watch as many videos as you can, especially, if you are unfamiliar with Neural Networks.
 - The videos have got amazing animation.