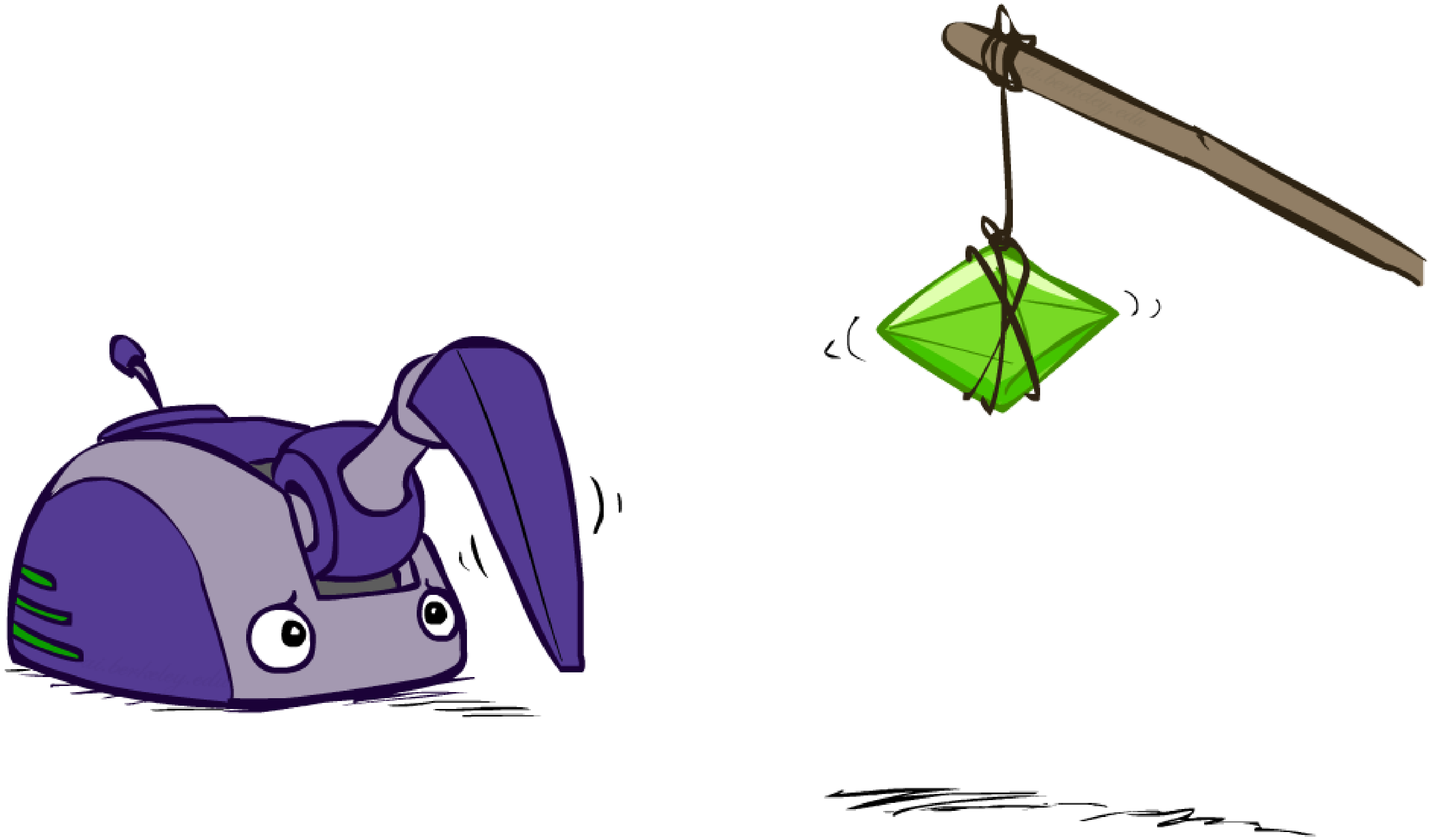
# CSE 573: Artificial Intelligence

Hanna Hajishirzi

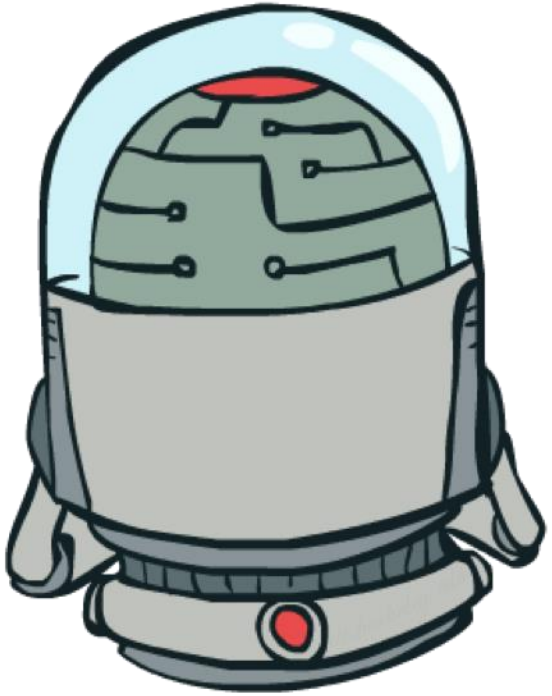
Reinforcement Learning


slides adapted from
Dan Klein, Pieter Abbeel ai.berkeley.edu
And Dan Weld, Luke Zettlemoyer

# Reinforcement Learning

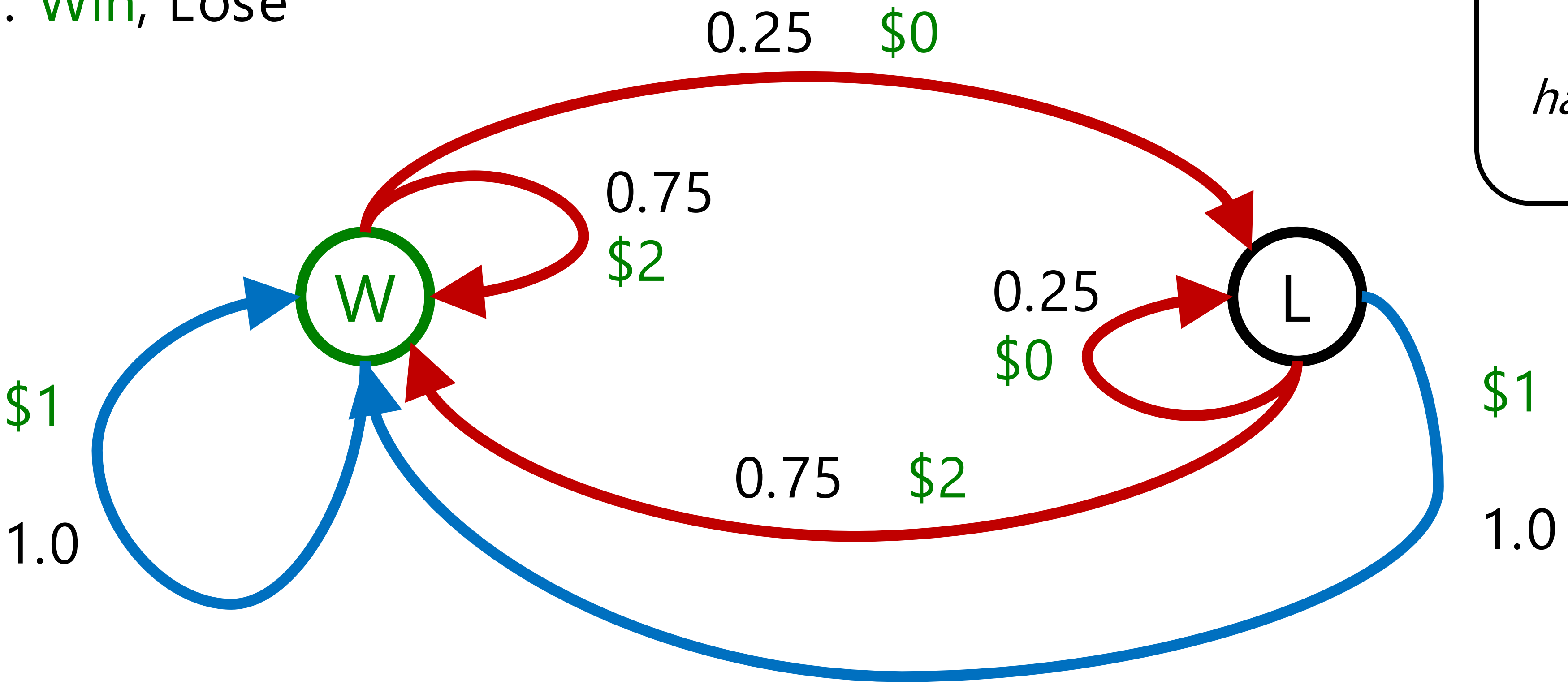


# Double Bandits

# Double-Bandit MDP

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



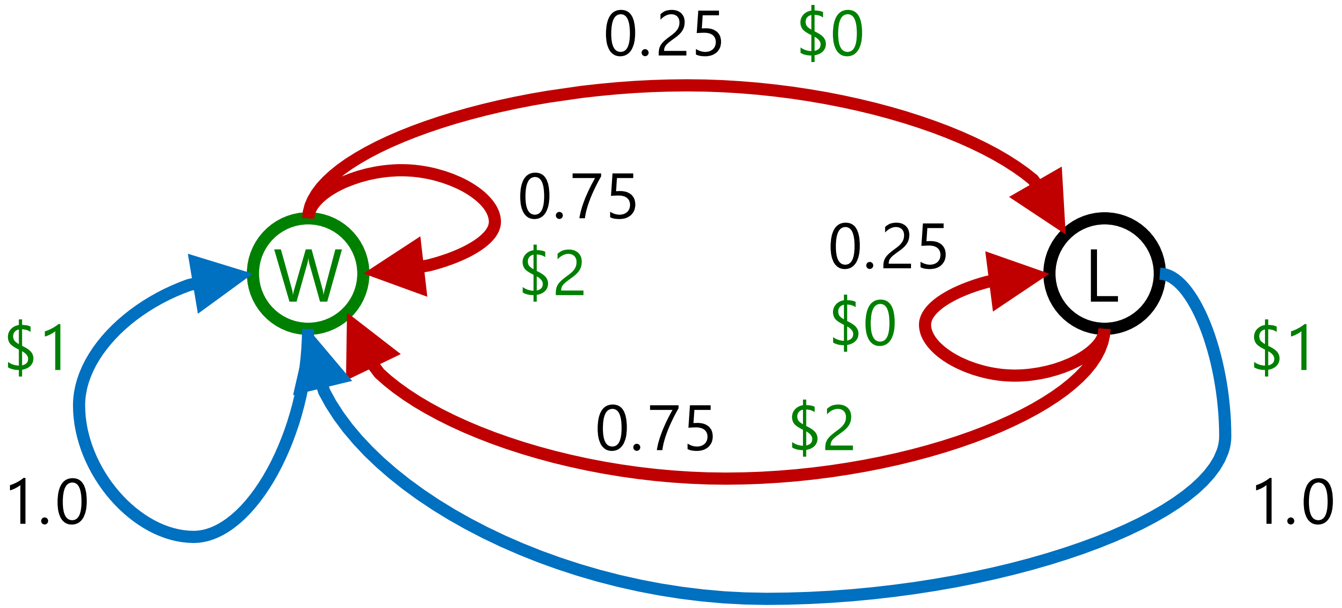

*No discount*
*10 time steps*
*Both states have the same value*

# 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*
*10 time steps*

| | Value |
|---|---|
| Play Red | 15 |
| Play Blue | 10 |



# Let’s Play!

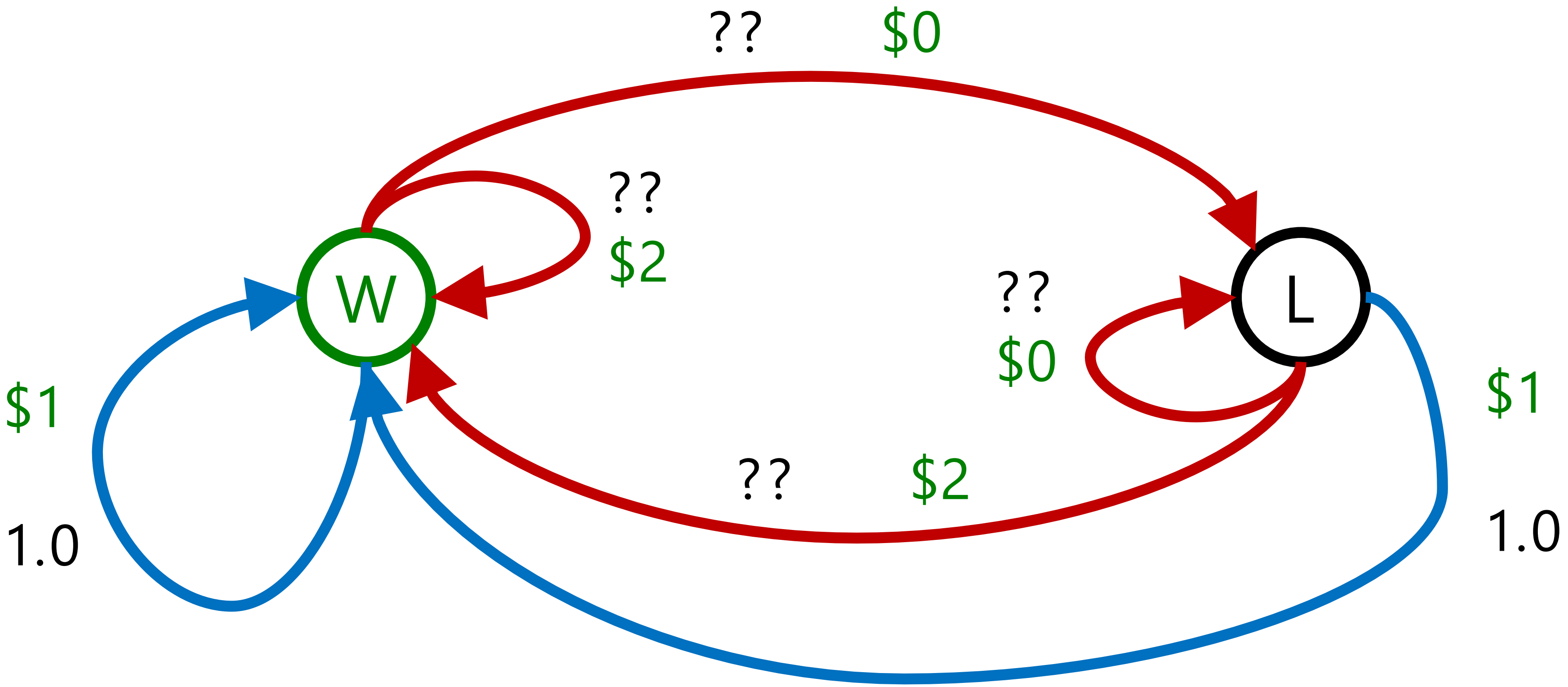


$2 $2 $0 $2 $2

$2 $2 $0 $0 $0

# Online Planning

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



# Let's Play!



$0 $0 $2 $0

$0 $2 $2 $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

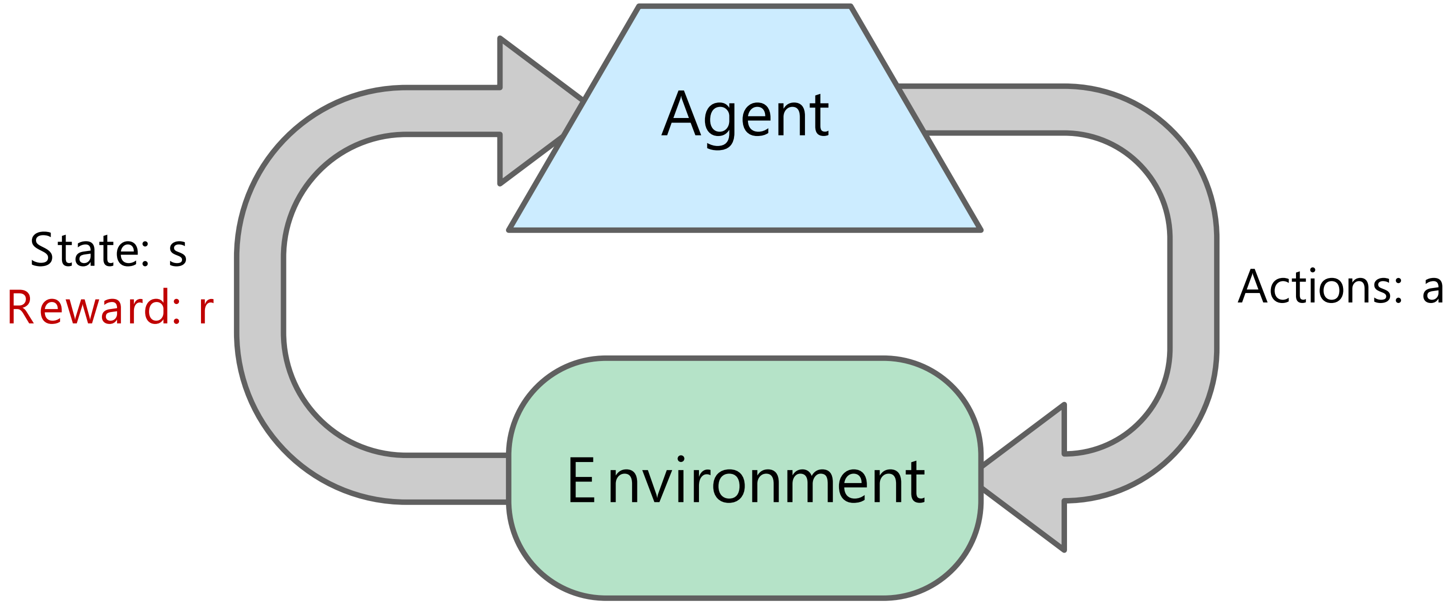

- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

# Reinforcement Learning

- 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

# 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]

[Kohl and Stone, ICRA 2004]

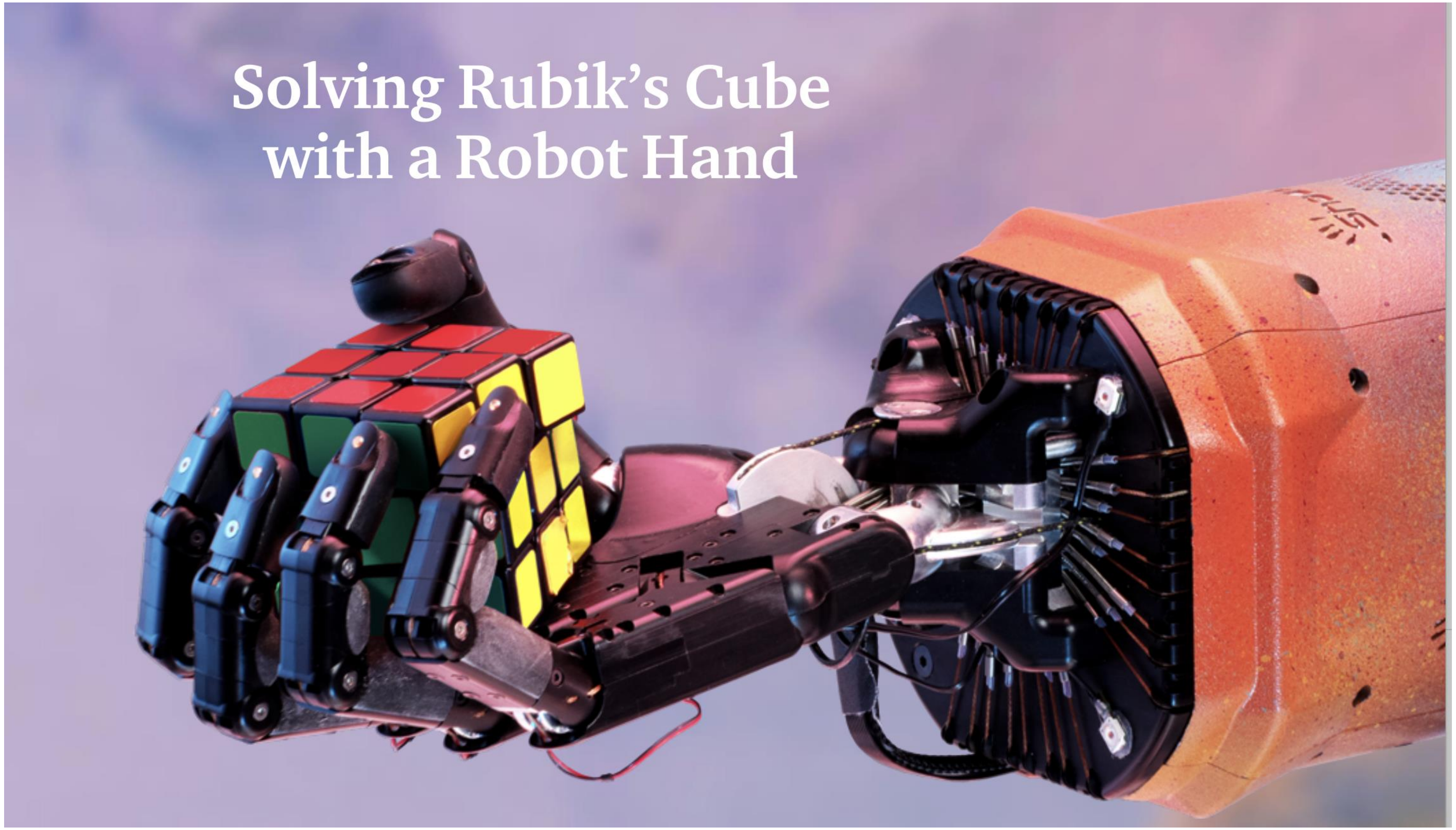
# Example: Toddler Robot

[Tedrake, Zhang and Seung, 2005]

[Video: TODDLER – 40s]

# Robotics Rubik Cube

- https://www.youtube.com/watch?v=x4O8pojMF0w



# CSE 573: Artificial Intelligence

Hanna Hajishirzi

Reinforcement Learning


slides adapted from
Dan Klein, Pieter Abbeel ai.berkeley.edu
And Dan Weld, Luke Zettlemoyer

# Announcements

- PS2: April 29
- Project proposals: May 6th
- Paper review: May 13

# Project Proposal

- Project proposals: May 6th
  - Pick projects close to you interest, or select from here: list of potential projects. Your final project can also be a re-implementation of one of the recent papers from AI/ML/NLP/Computer vision conferences.
- The project proposal is a 1-page summary of the project topic, motivation, definition, dataset, and resources. It should also include the milestones, detailed experiment plan, and the timeline to complete each milestone.

# Paper Review

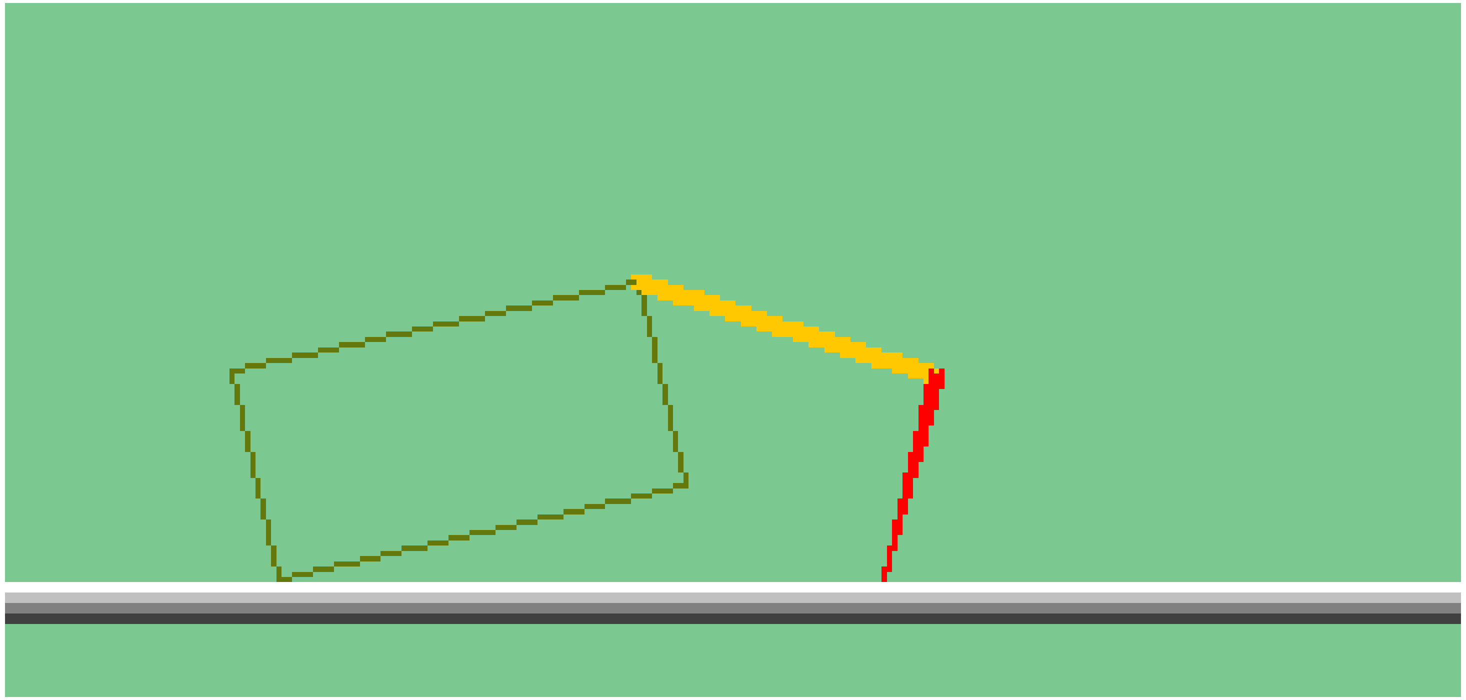
- Paper review:
  - 1. Describe what problem or question this paper addresses, and the main contributions that it makes towards a solution or answer. a. Problem/Question:
  - b. Solution/approach:
  - c. Contributions (list at least two):
- 2. Evaluate the paper in terms of novelty, significance, and empirical results. 3. Describe the main strengths you see in the paper. 4. Describe critiques and weaknesses you see in the paper.

# 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 $\pi(s)$

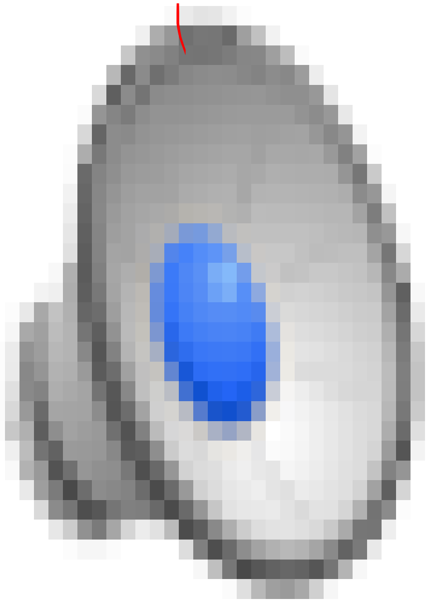

- New twist: don't know T or R
  - I.e. we don't know which states are good or what the actions do
  - Must actually try actions and states out to learn

# The Crawler!

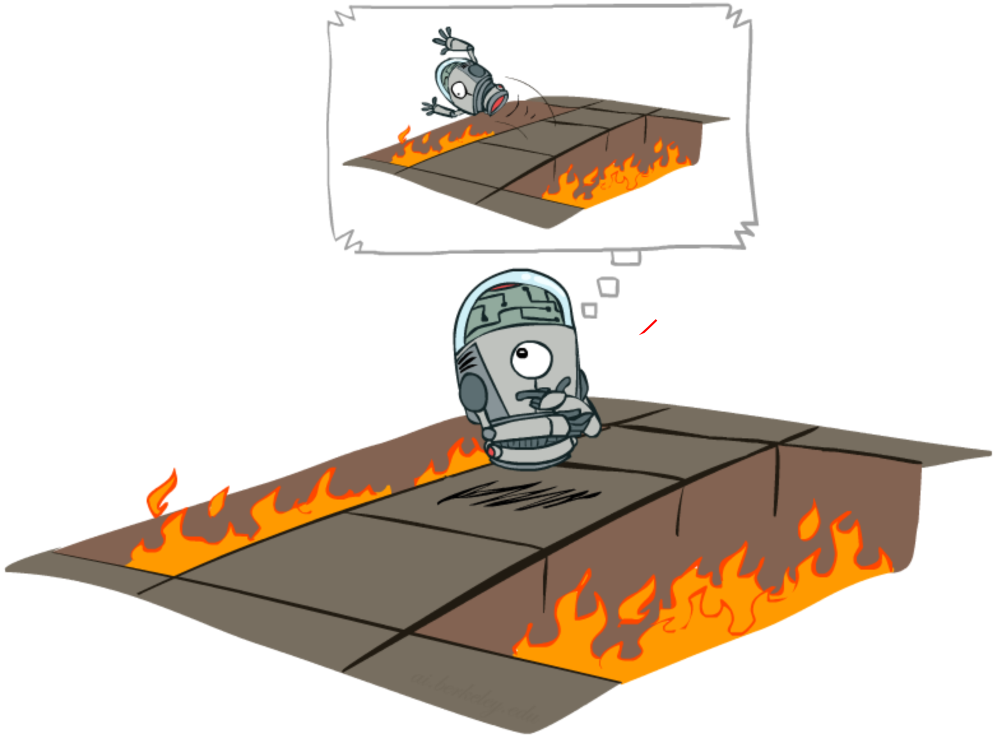


[Demo: Crawler Bot (L10D1)] [You, in Project 3]

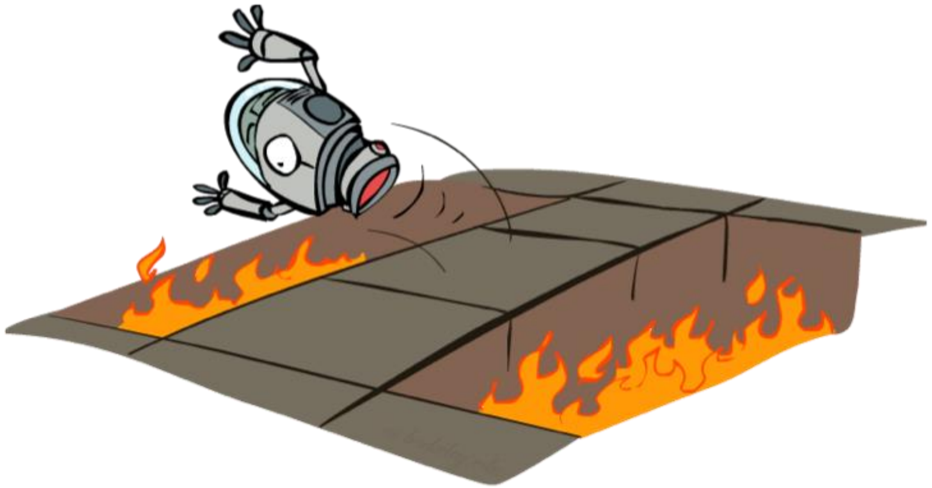
# Video of Demo Crawler Bot



# Offline (MDPs) vs. Online (RL)

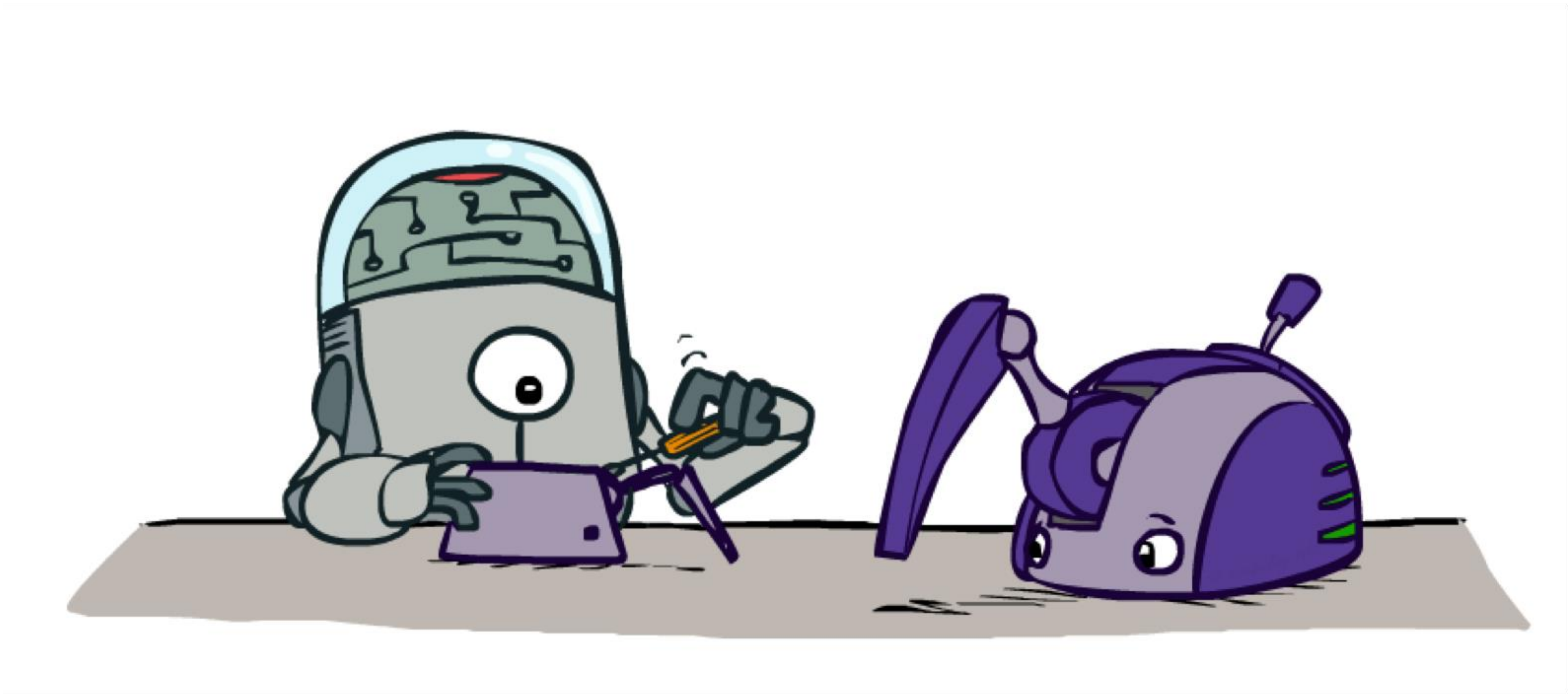


Offline Solution



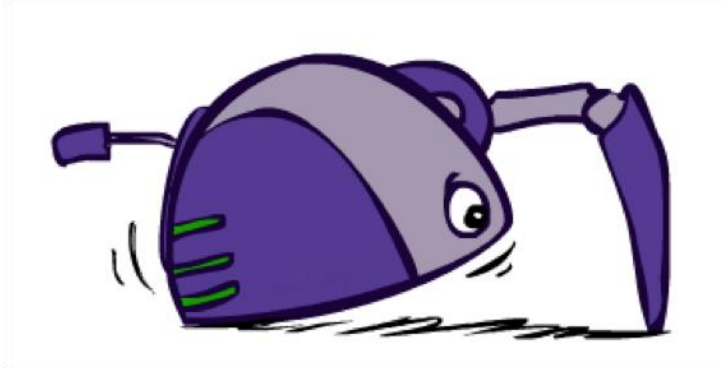
Online Learning

# Model-Based Learning



# Model-Based Learning

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



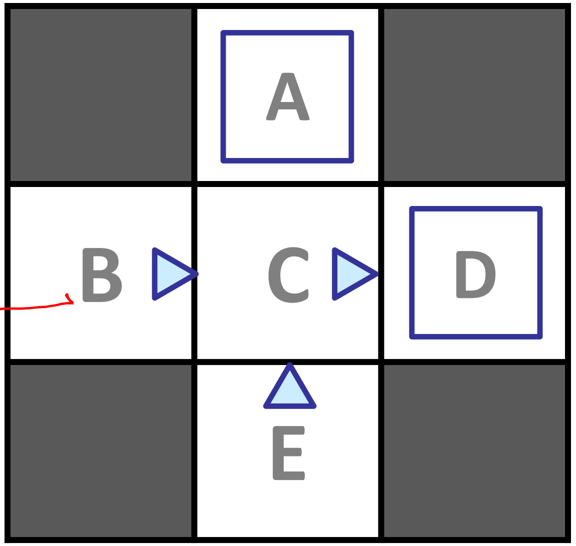
- Step 1: Learn empirical MDP model
  - Count outcomes s′ for each s, a
  - Normalize to give an estimate $\hat{T}(s,a,s')$
  - Discover each $\hat{R}(s,a,s')$ when we experience (s, a, s′)

- Step 2: Solve the learned MDP   (r)
  - For example, use value iteration, as before   R(s, a, s′)   r

# Example: Model-Based Learning

T(B, east, - )
T(C, east, ...)  3/4 D  T(s, a, s')

## Input Policy π




*Assume:* γ = 1

## Observed Episodes (Training)

1/4 A

**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

R(s, a, s')

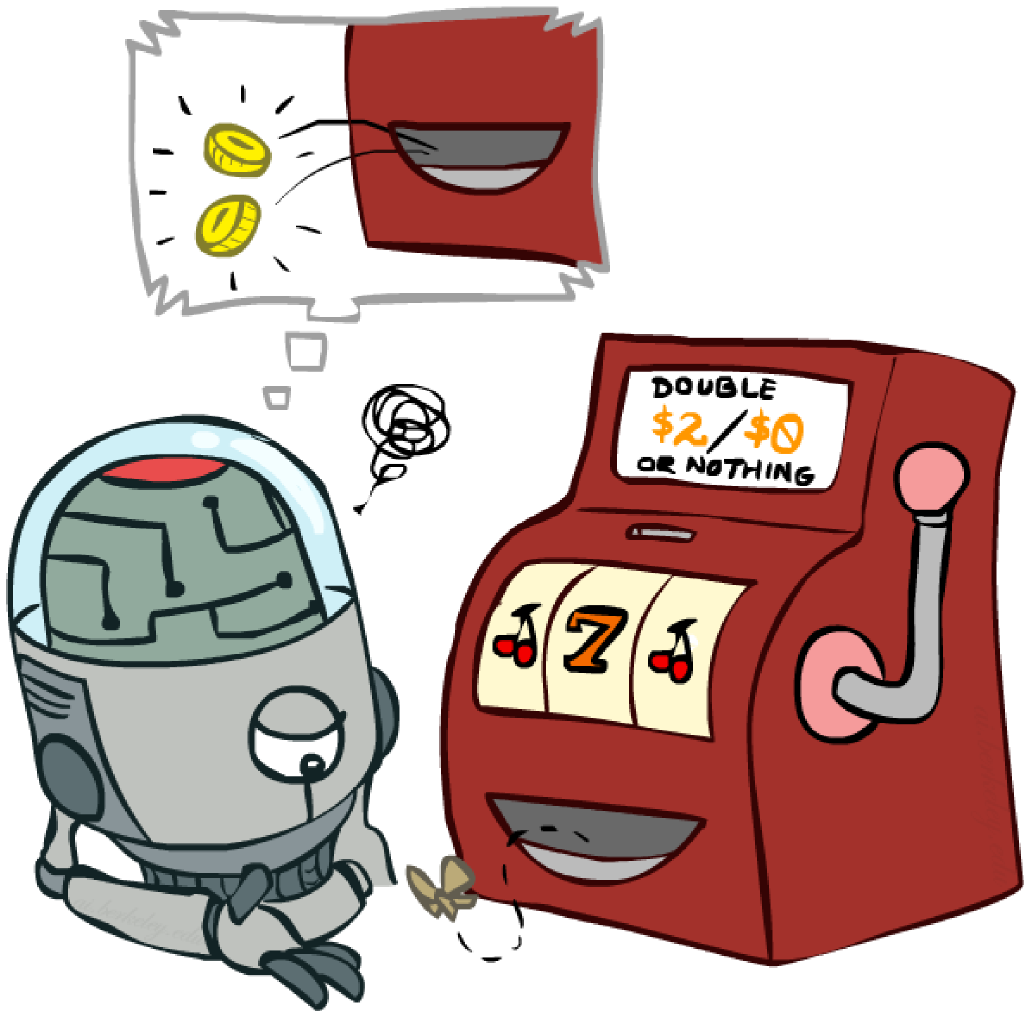
$\hat{T}(s, a, s')$

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-Free Learning

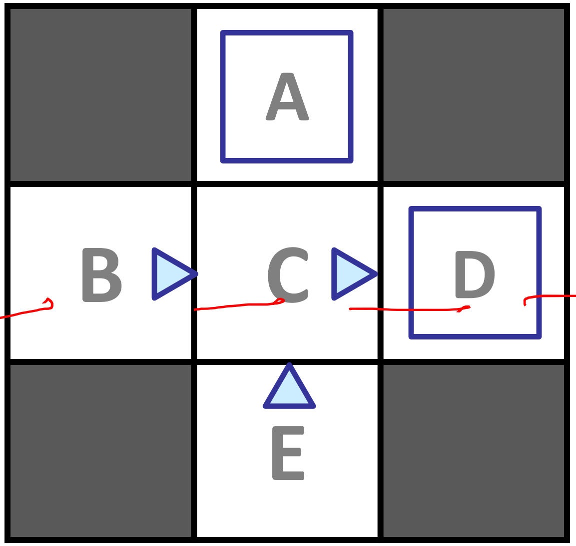


# Direct Evaluation

- Goal: Compute values for each state under $\pi$
- 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
- This is called direct evaluation

# Example: Direct Evaluation

## 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

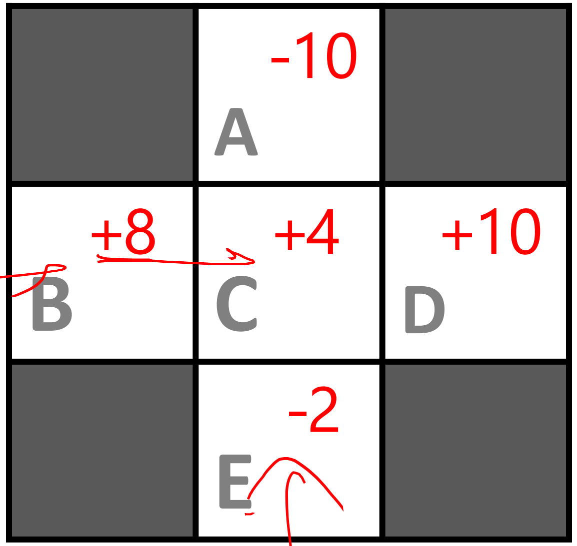
**Episode 3**

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

## Output Values



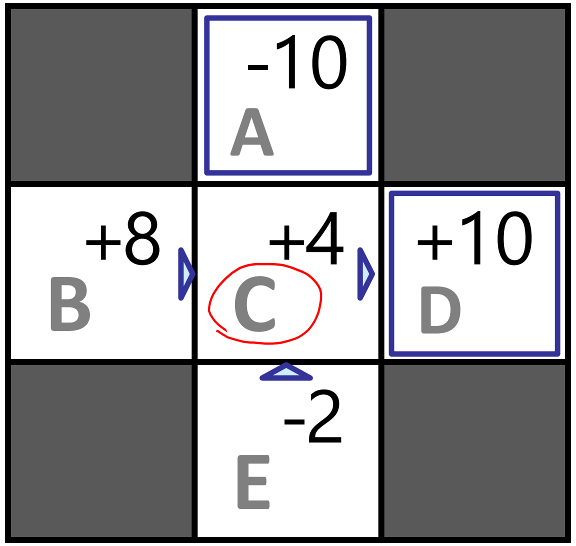

*If B and E both go to C under this policy, how can their values be different?*

# 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

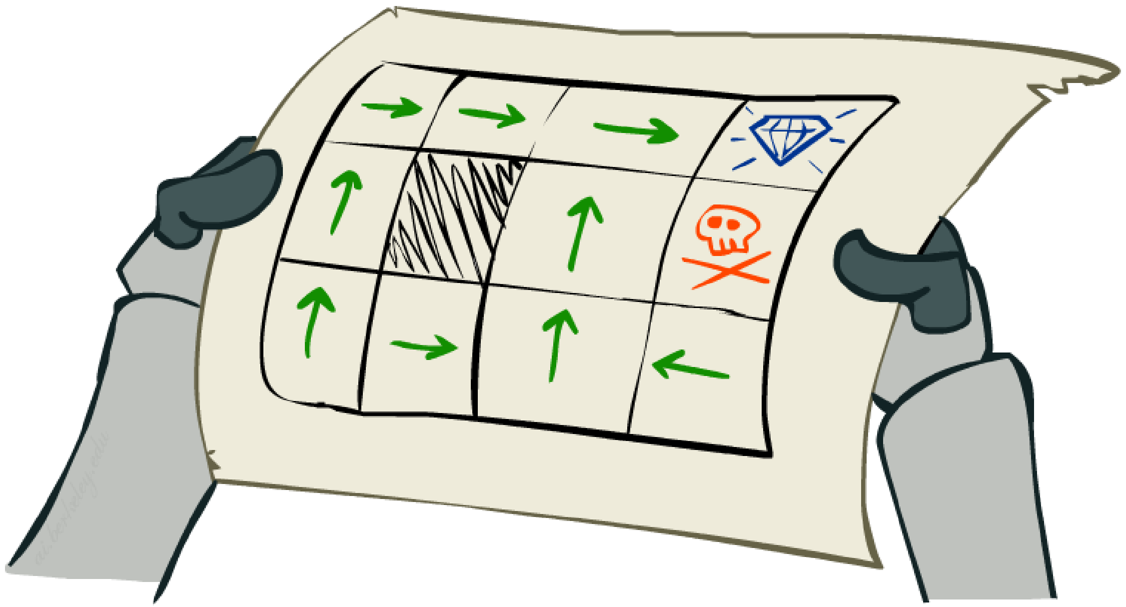
**Output Values**




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

# Passive Reinforcement Learning

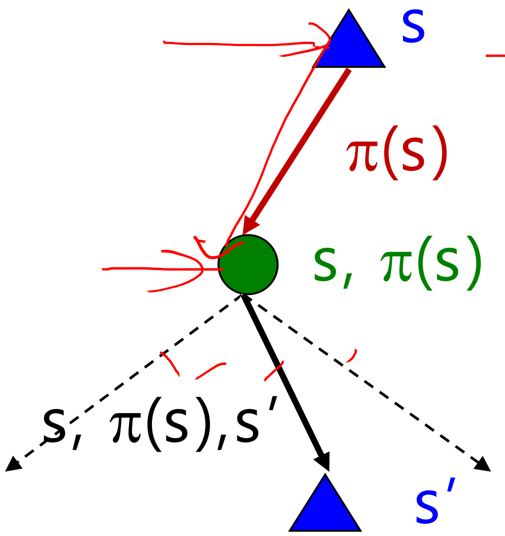
- Simplified task: policy evaluation
  - Input: a fixed policy $\pi(s)$
  - You don't know the transitions $T(s,a,s')$
  - You don't know the rewards $R(s,a,s')$
  - Goal: learn the state values



- 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.

# 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?

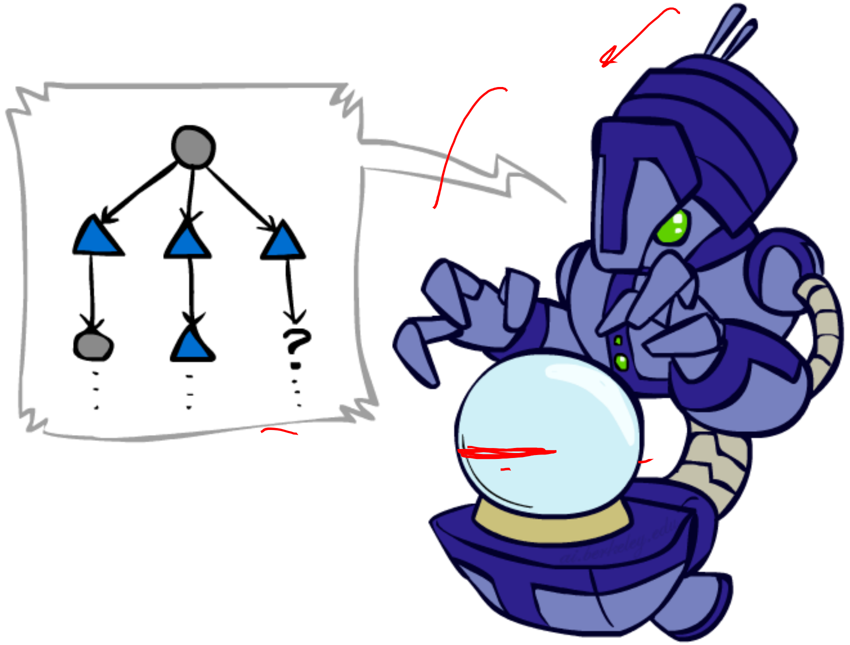
- We want to improve our estimate of V by computing these averages:

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

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

$$sample_1 = R(s,\pi(s),s'_1) + \gamma V^{\pi}_{k}(s'_1)$$
$$sample_2 = R(s,\pi(s),s'_2) + \gamma V^{\pi}_{k}(s'_2)$$
$$\ldots$$
$$sample_n = R(s,\pi(s),s'_n) + \gamma V^{\pi}_{k}(s'_n)$$

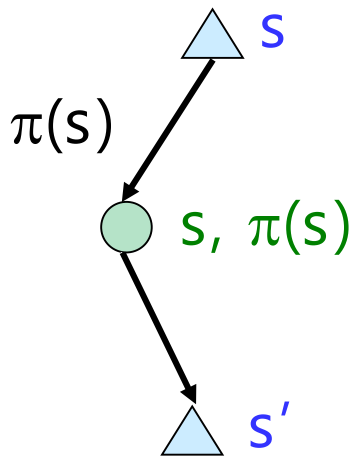
$$V^{\pi}_{k+1}(s) \leftarrow \frac{1}{n}\sum_{i} sample_i$$



from state s.

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

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

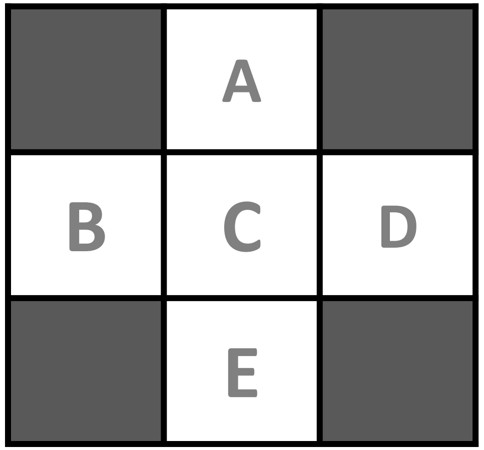
- Exponential moving average
  - The running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
  - Makes recent samples more important

    $\bar{x}_n = (1-\alpha)\left[(1-\alpha)x_{n-2} + \alpha \cdot x_{n-1}\right] + \underline{\alpha} \cdot x_n$

    $\alpha^2$
  - Forgets about the past (distant past values were wrong anyway)
- Decreasing learning rate (alpha) can give converging averages

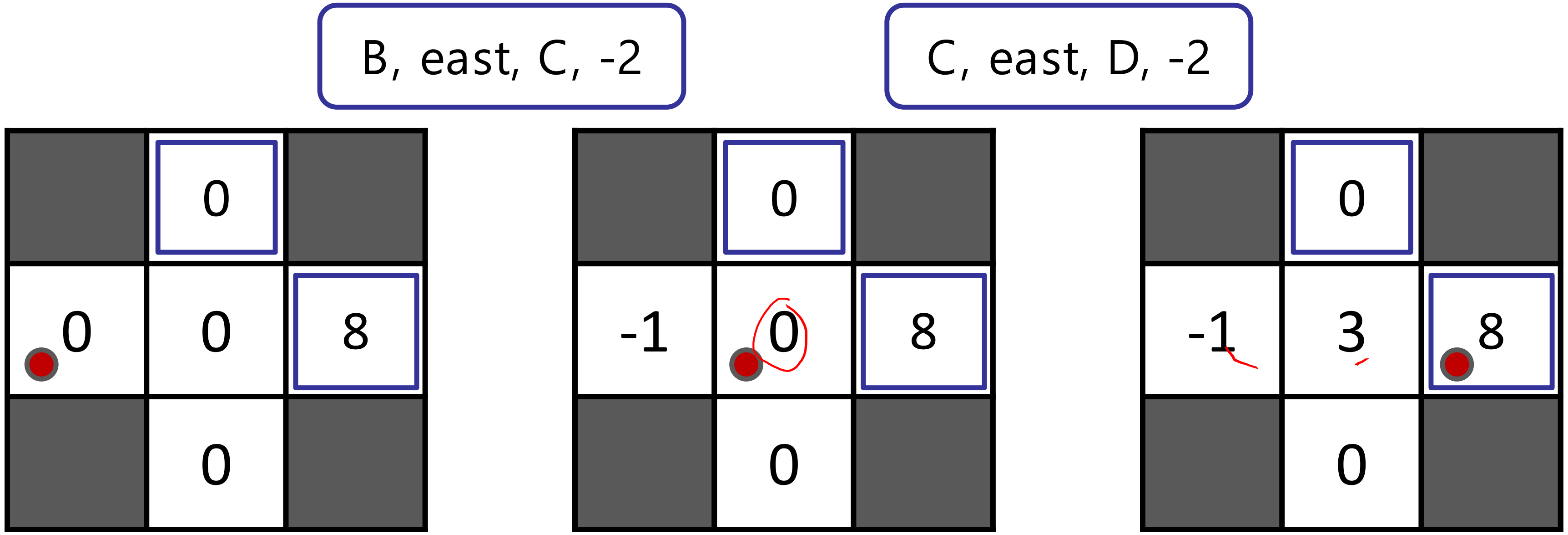
# Example: Temporal Difference Learning

## States




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

## Observed Transitions




$$V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + \alpha\left[\overbrace{R(s,\pi(s),s')}^{-2} + \gamma V^\pi(s')\right]$$

$\alpha$

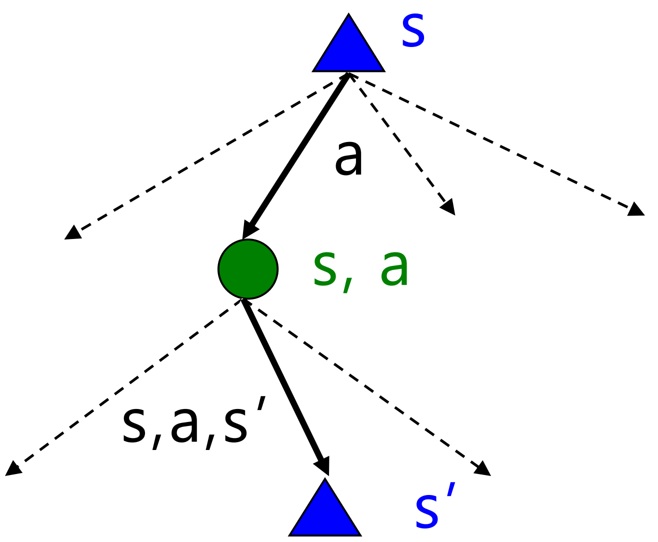
$0 + 1/2\,[\,-2 + 8\,]$

# 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!

# Recap: 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
- Big Idea: Compute all averages over T using sample outcomes

# The Story So Far: MDPs and RL

## Known MDP: Offline Solution

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | Value / policy iteration |
| Evaluate a fixed policy $\pi$ | Policy evaluation |

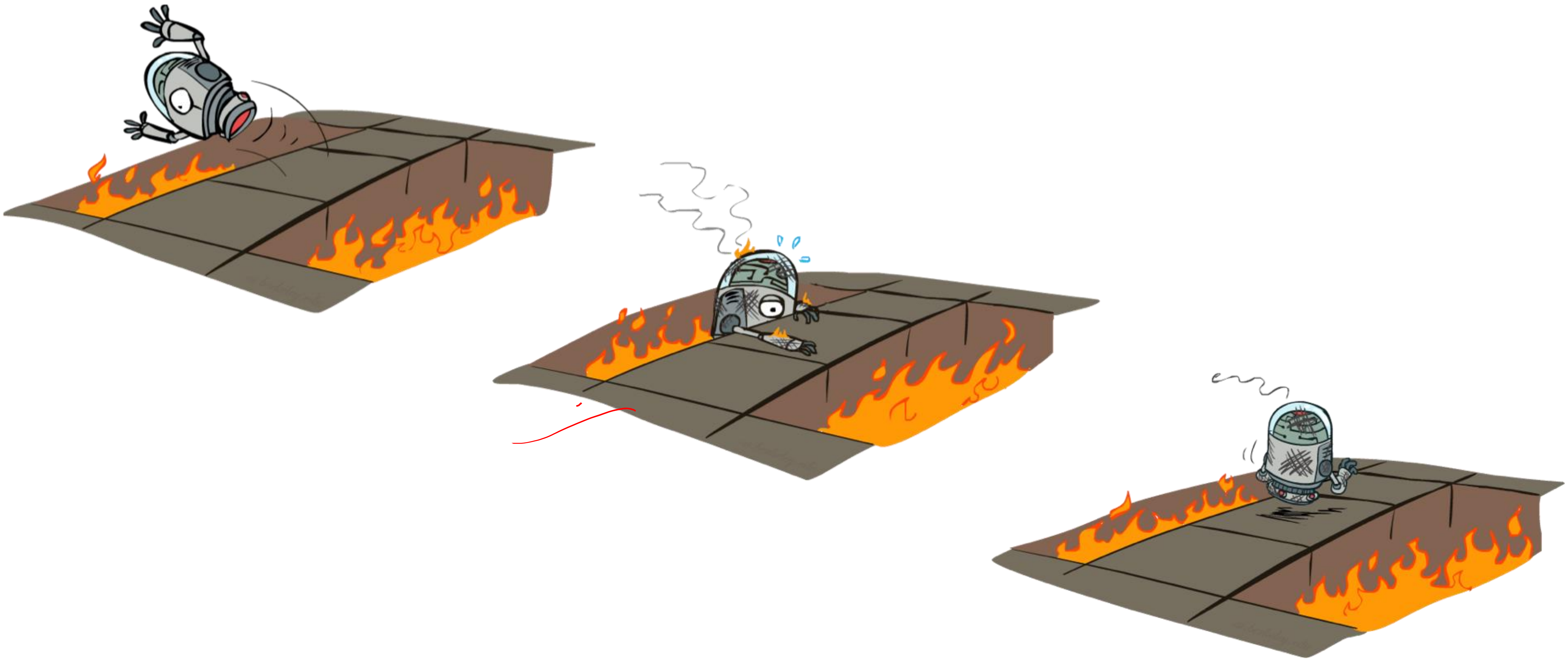
## Unknown MDP: Model-Based

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | VI/PI on approx. MDP |
| Evaluate a fixed policy $\pi$ | PE on approx. MDP |

## Unknown MDP: Model-Free

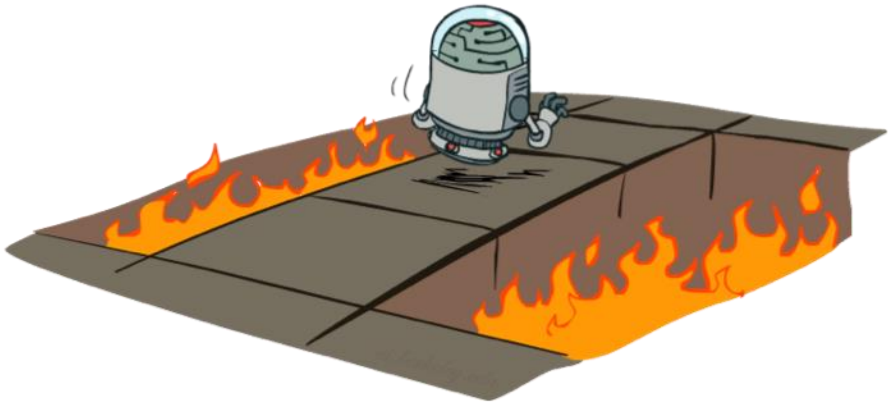
| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | Q-learning |
| Evaluate a fixed policy $\pi$ | Value Learning |

# Active Reinforcement Learning



# Active Reinforcement Learning

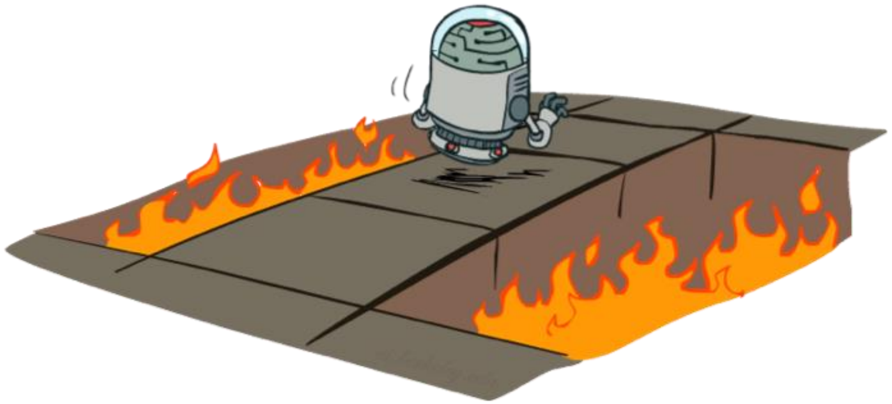
- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values



- 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…

# Model-Free Learning

- act according to current optimal (based on Q-Values)
- but also explore…



# 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:

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

- 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:

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

# Q-Learning

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

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

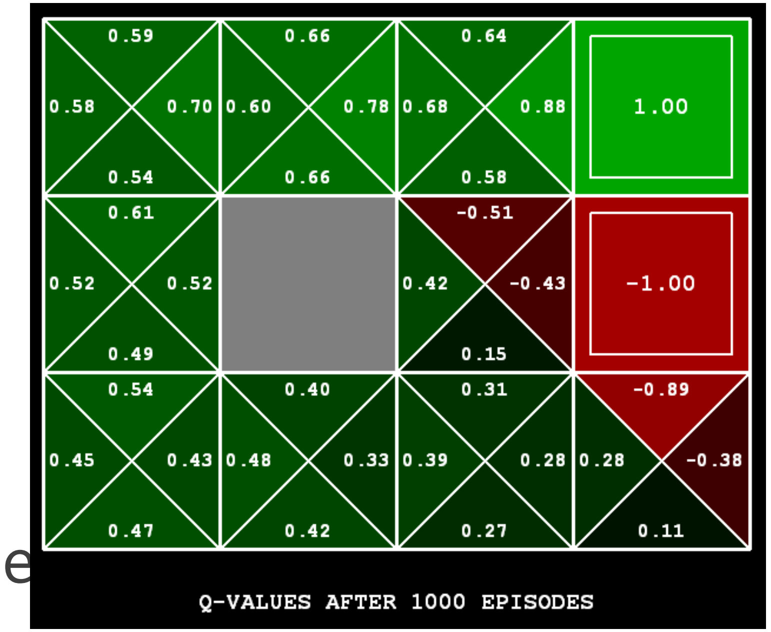
- Learn Q(s,a) values as you go
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

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

no longer policy evaluation!

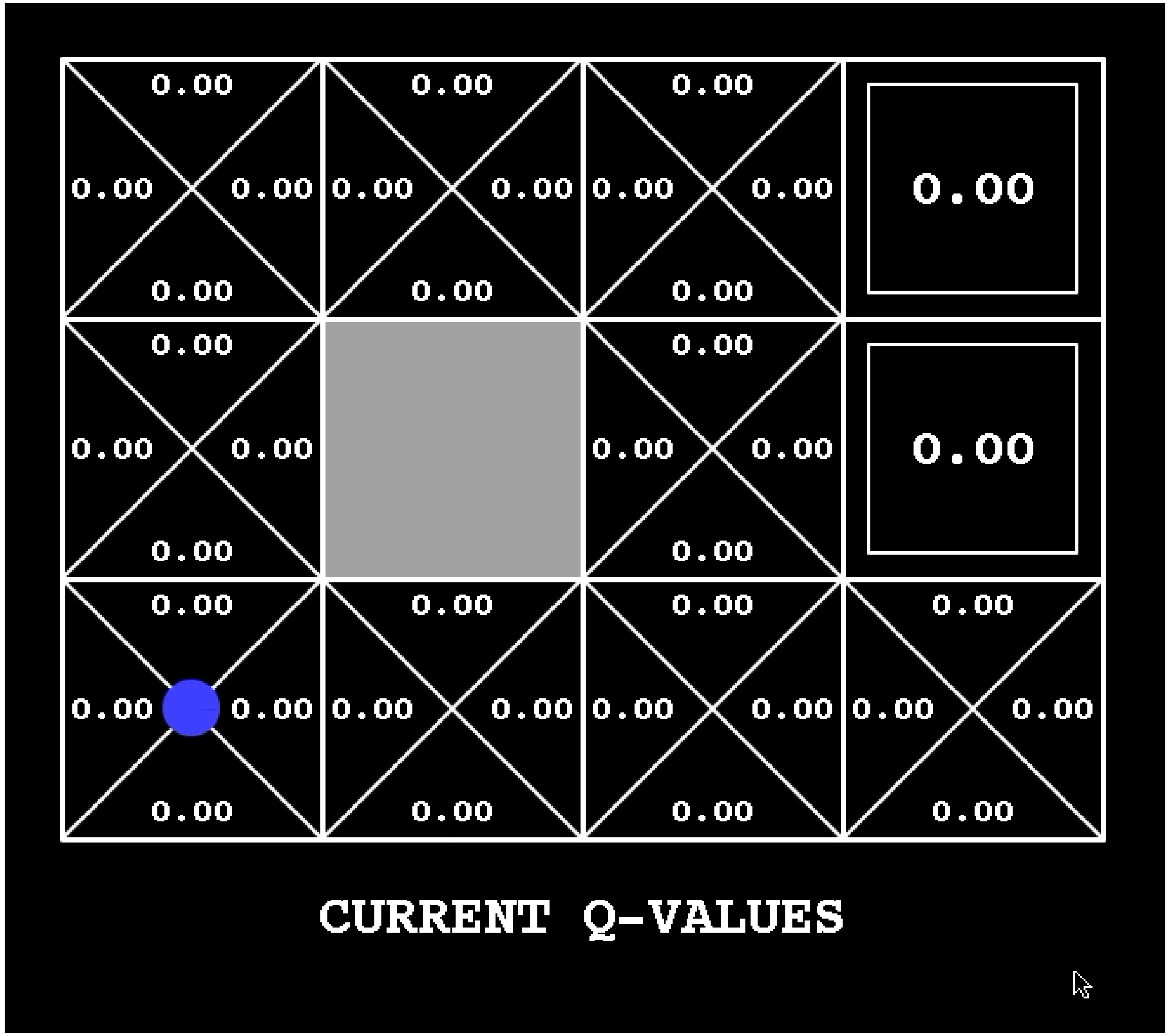
  - Incorporate the new estimate into a running average:

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

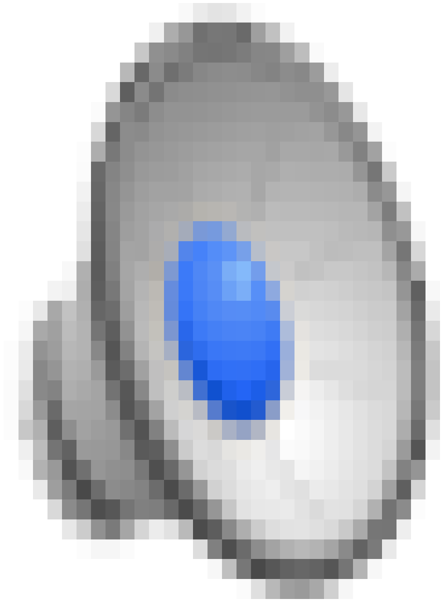



[Demo: Q-learning – gridworld (L10D2)]
[Demo: Q-learning – crawler (L10D3)]

# Q-Learning Demo



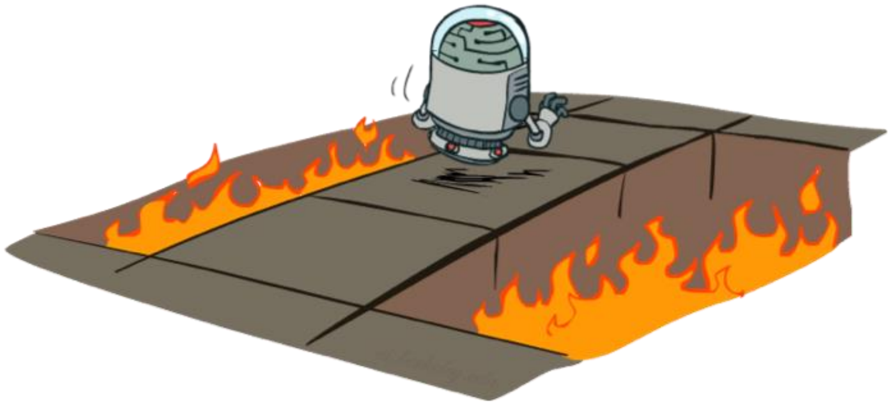
# Video of Demo Q-Learning -- Gridworld



# Video of Demo Q-Learning -- Crawler

# Q-Learning: act according to current optimal (and also explore… )

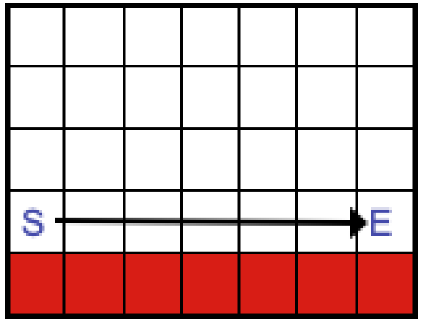
- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values

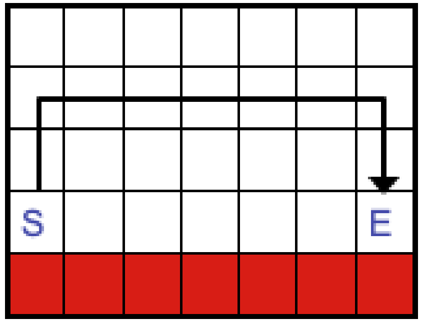


- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

# Q-Learning Properties

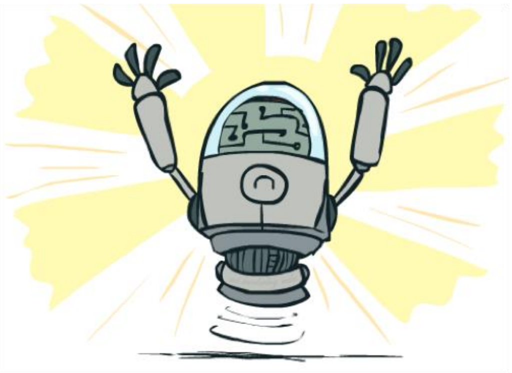
- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!



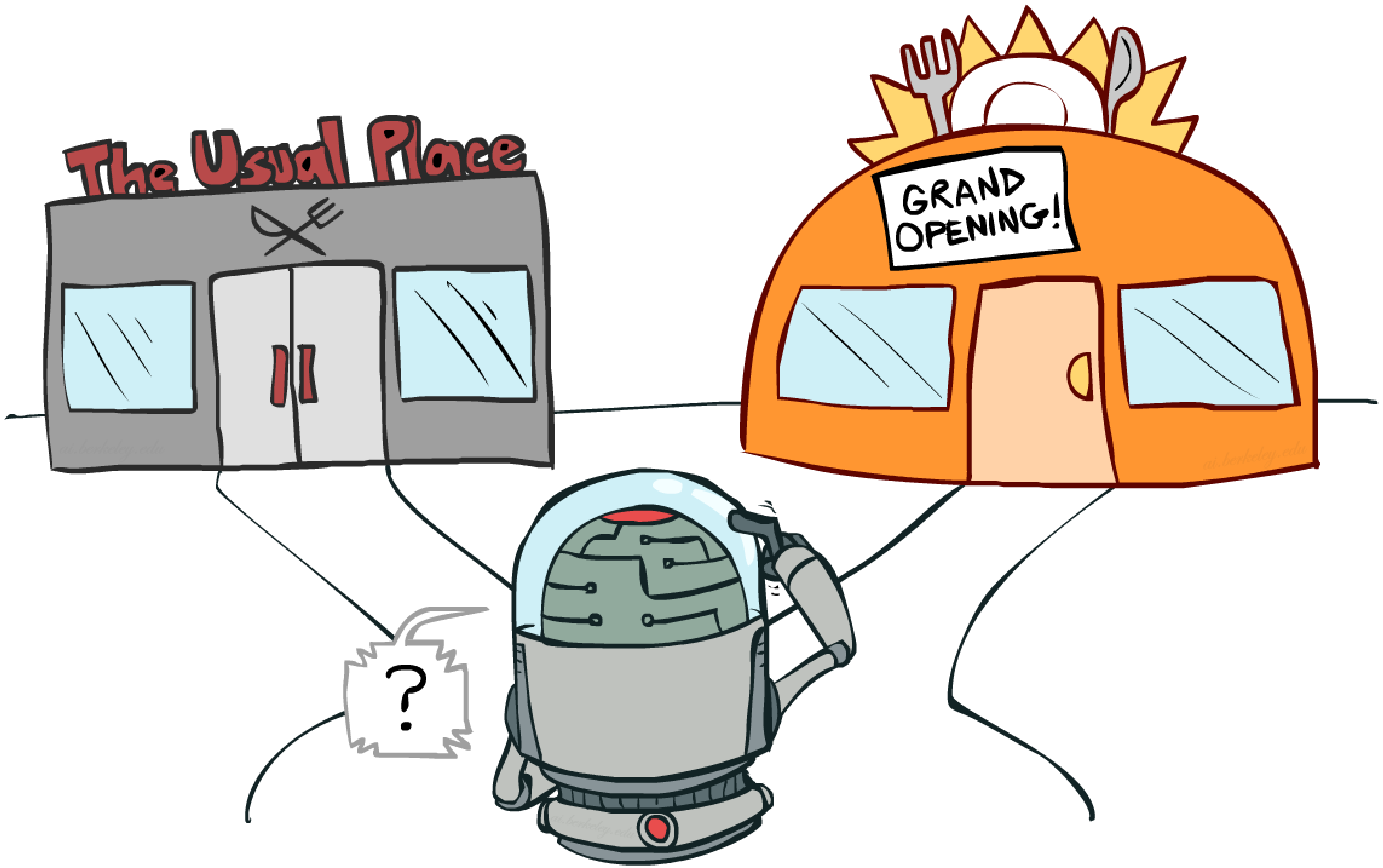




- This is called off-policy learning

- 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



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

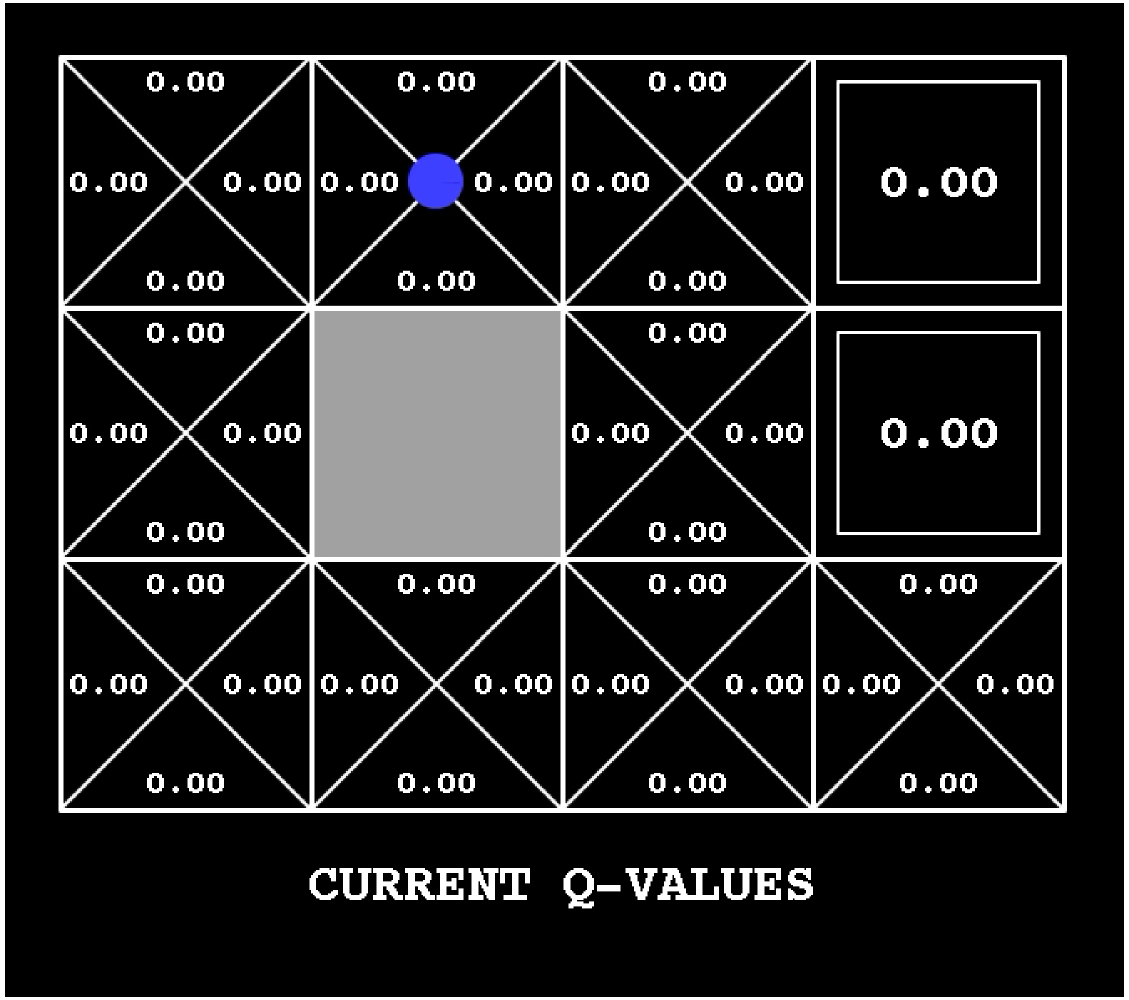
    Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

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

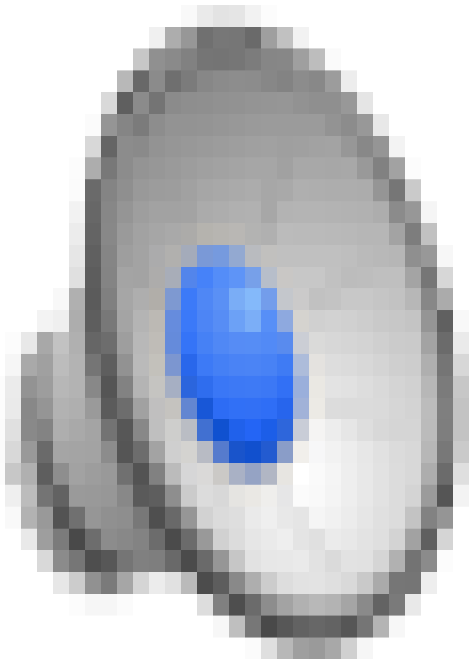
  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

[Demo: exploration – Q-learning – crawler – exploration function (L11D4)]

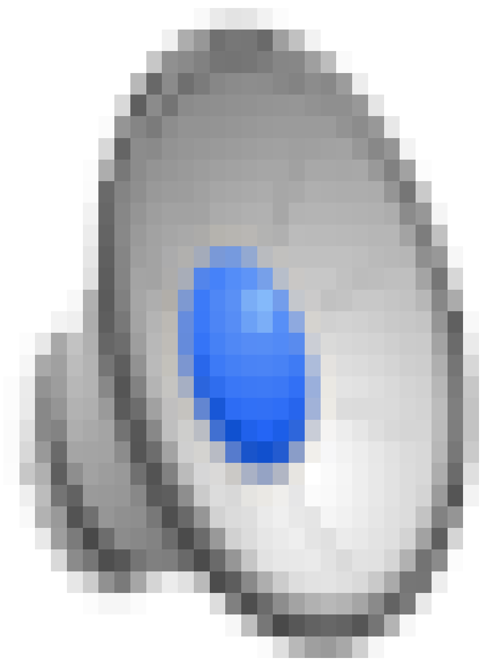
# Q-Learn Epsilon Greedy



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

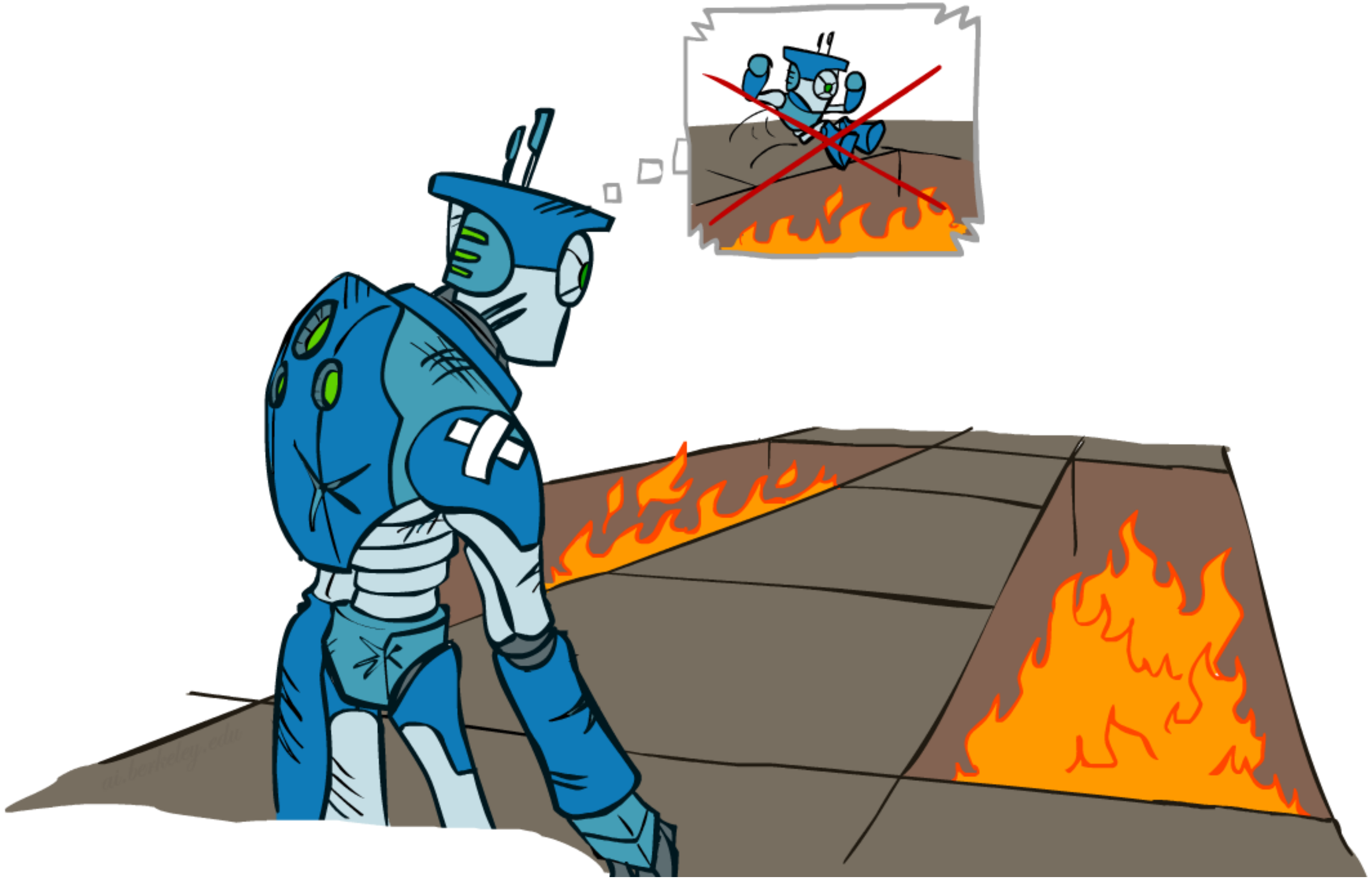


# Video of Demo Q-learning – Exploration Function – Crawler

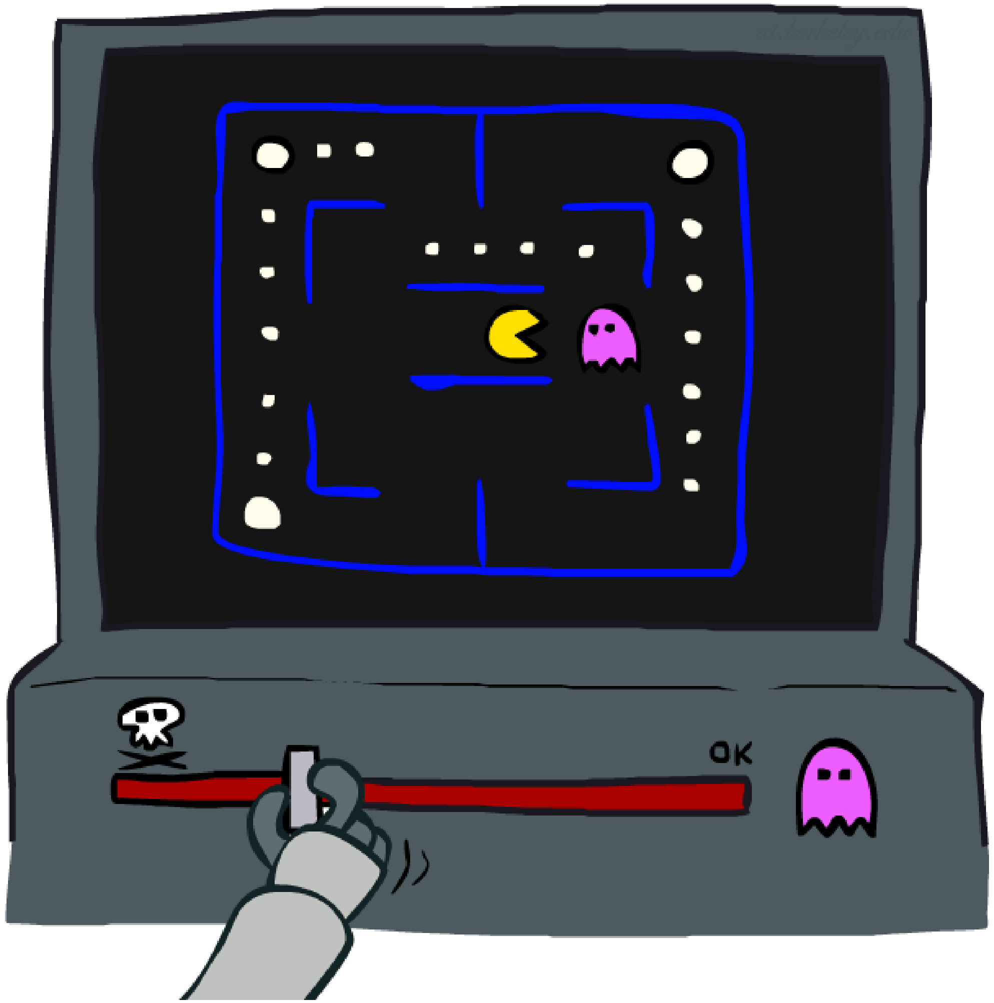


# Regret

- 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 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

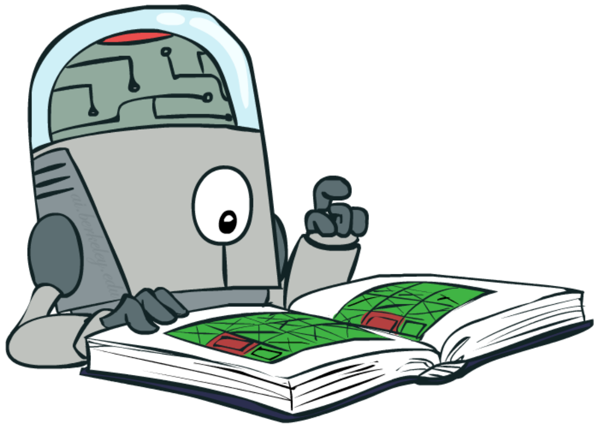


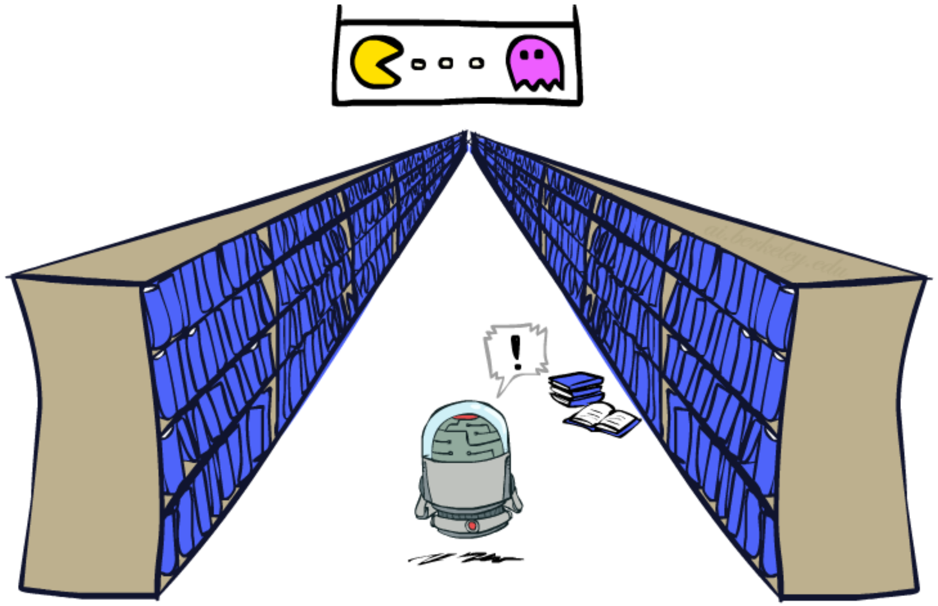
# Generalizing Across States

- 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

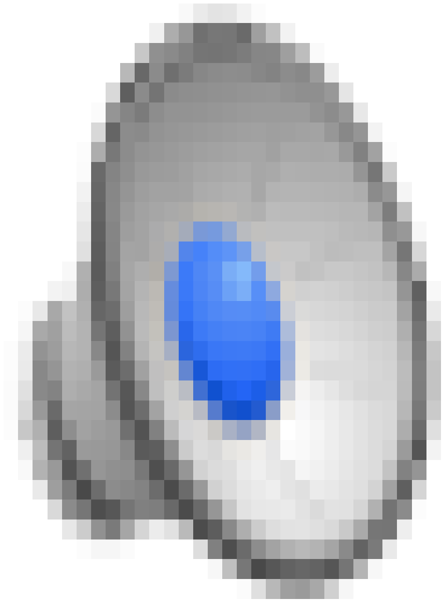
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning, and we'll see it over and over again



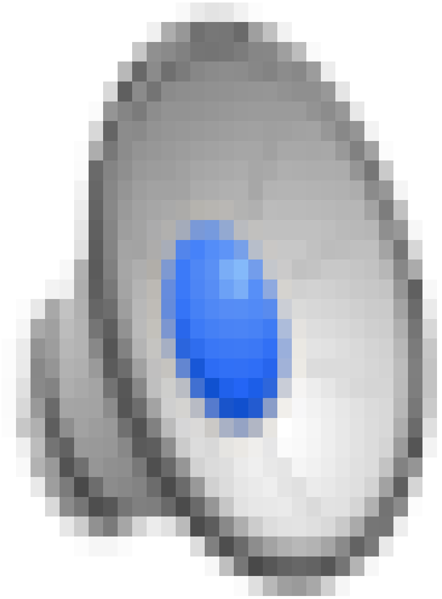


[demo – RL pacman]

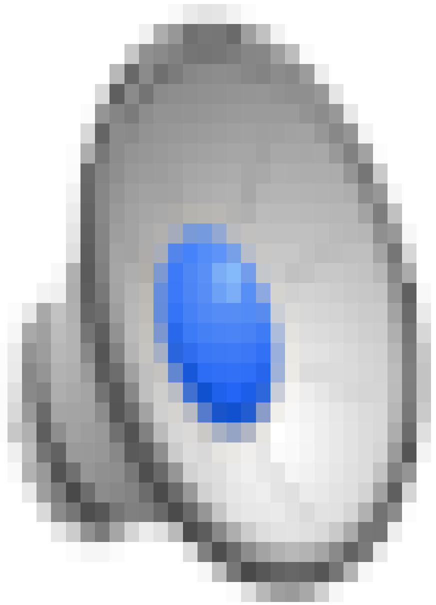
# Video of Demo Q-Learning Pacman – Tiny – Watch All



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

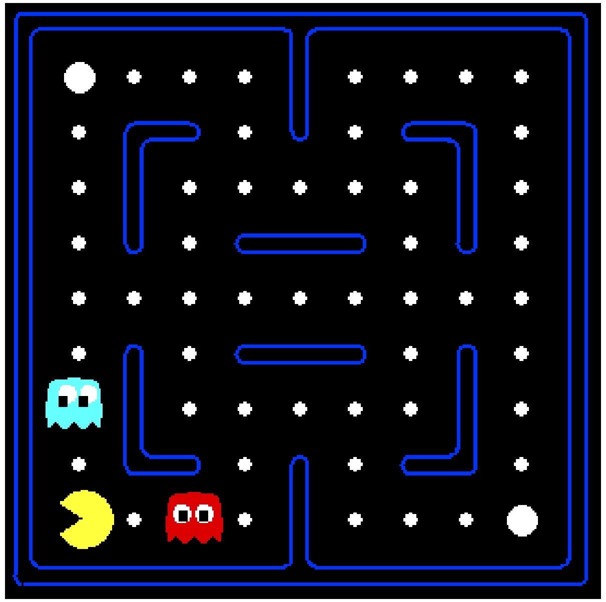


# Video of Demo Q-Learning Pacman – Tricky – Watch All

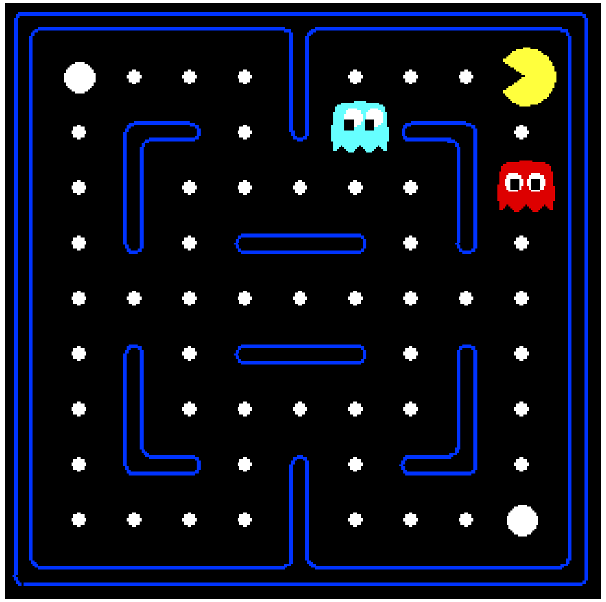


# Example: Pacman

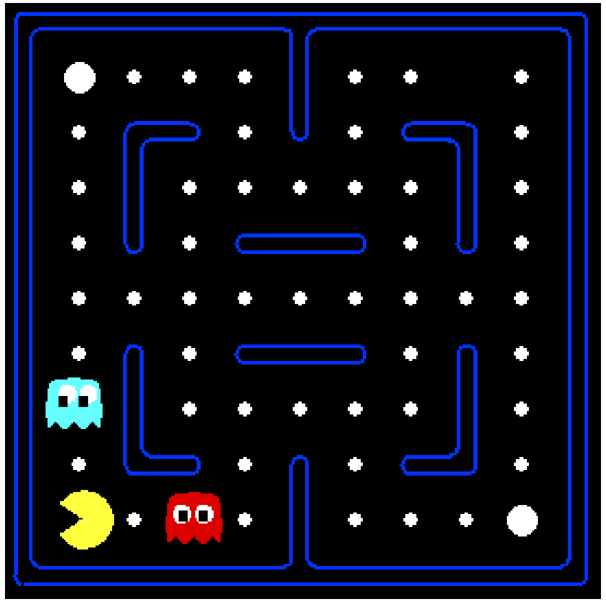
Let's say we discover through experience that this state is bad:



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

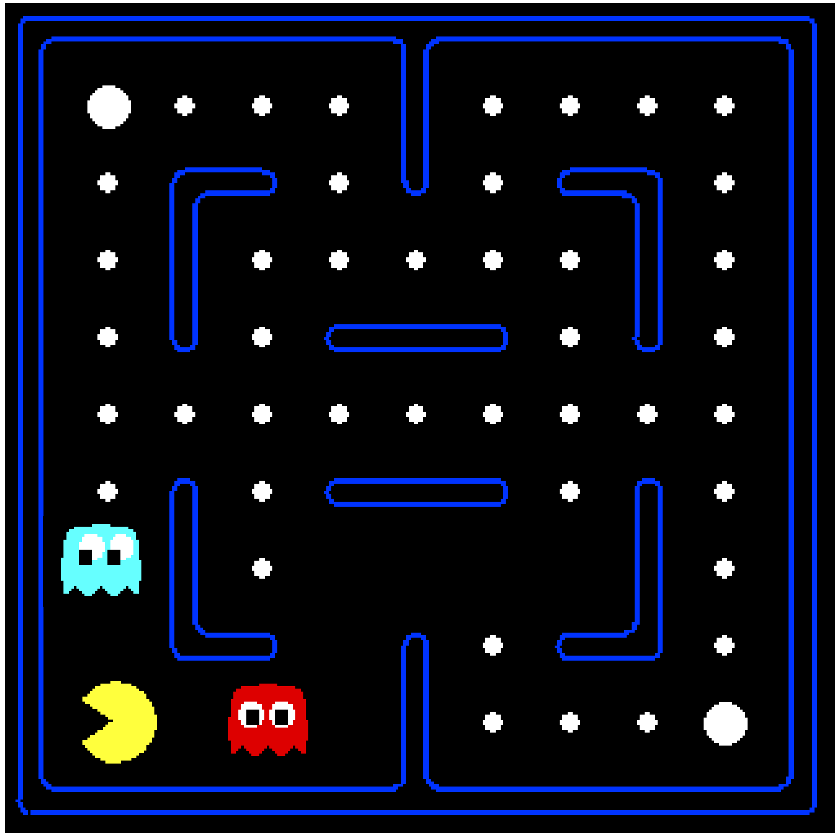


Or even this one!



# 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 / (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:

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

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

- Advantage: our experience is summed up in a few powerful numbers

- Disadvantage: states may share features but actually be very different in value!

# CSE 573:
# Artificial Intelligence

Hanna Hajishirzi

Reinforcement Learning


slides adapted from
Dan Klein, Pieter Abbeel ai.berkeley.edu
And Dan Weld, Luke Zettlemoyer

# Approximate Q-Learning

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

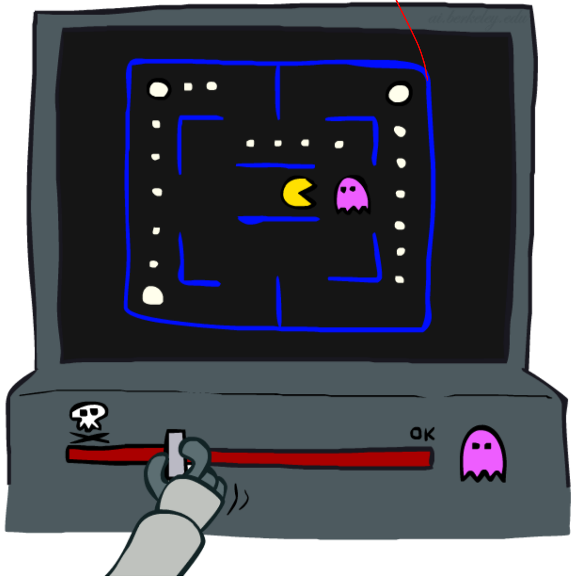
- Q-learning with linear Q-functions:

$$\text{transition } = (s,a,r,s')$$

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

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

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



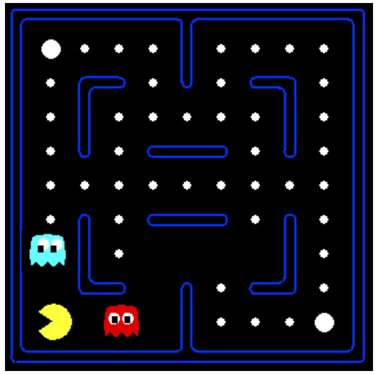
- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features
- Formal justification: online least squares

# Example: Q-Pacman

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

$\alpha = 2/501$

$s$

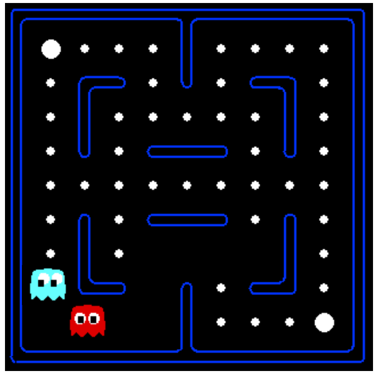


$f_{DOT}(s, \text{NORTH}) = 0.5$

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

$a = \text{NORTH}$
$r = -500$

$s'$



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

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

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

r + γ max Q(s', a') = -500

diff

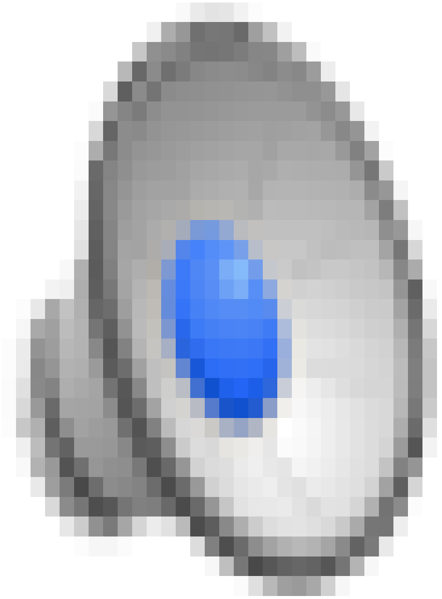
difference $= -501$

$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$

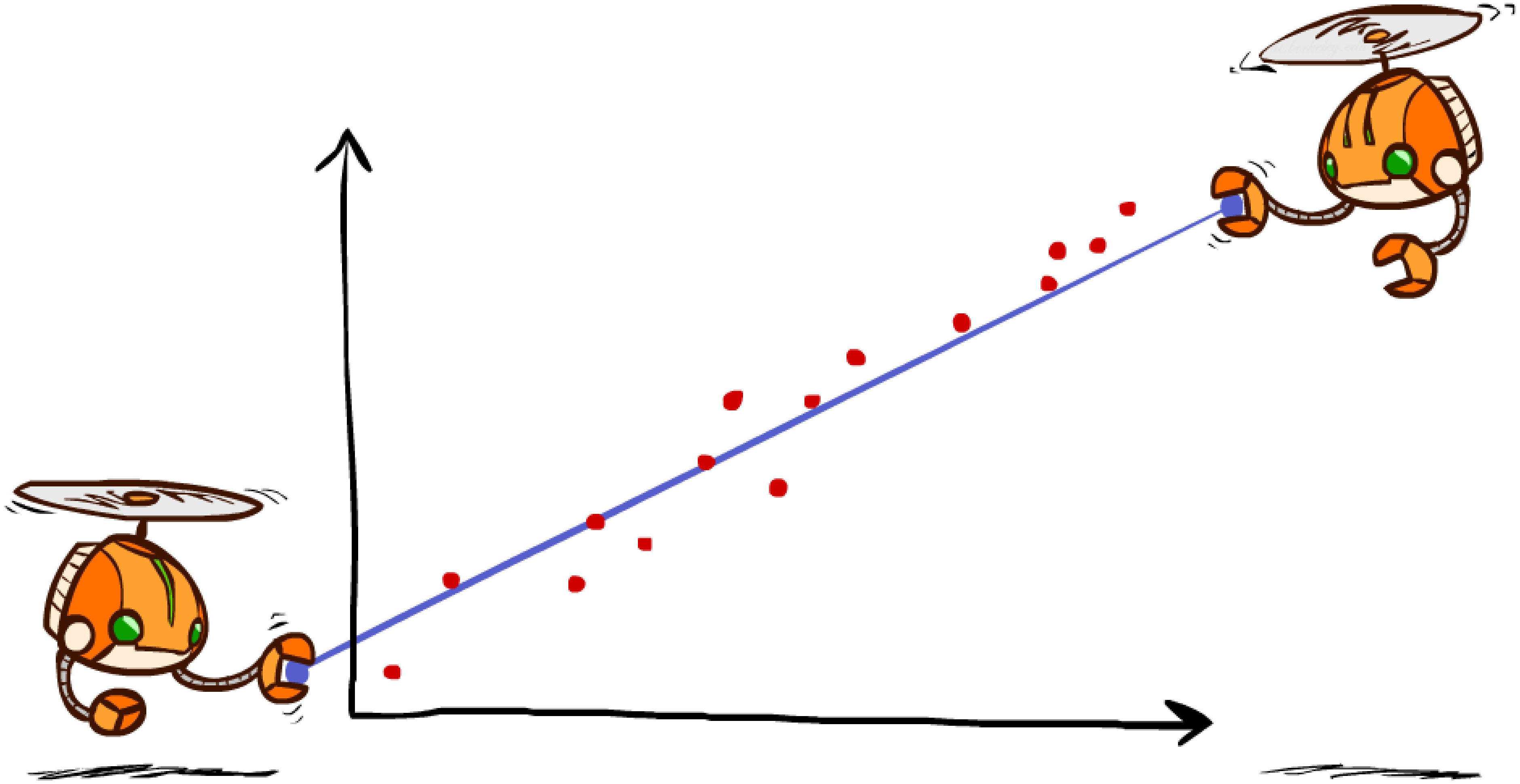
$w_{GST} \leftarrow -1.0 + \alpha\,[-501]\,1.0$

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

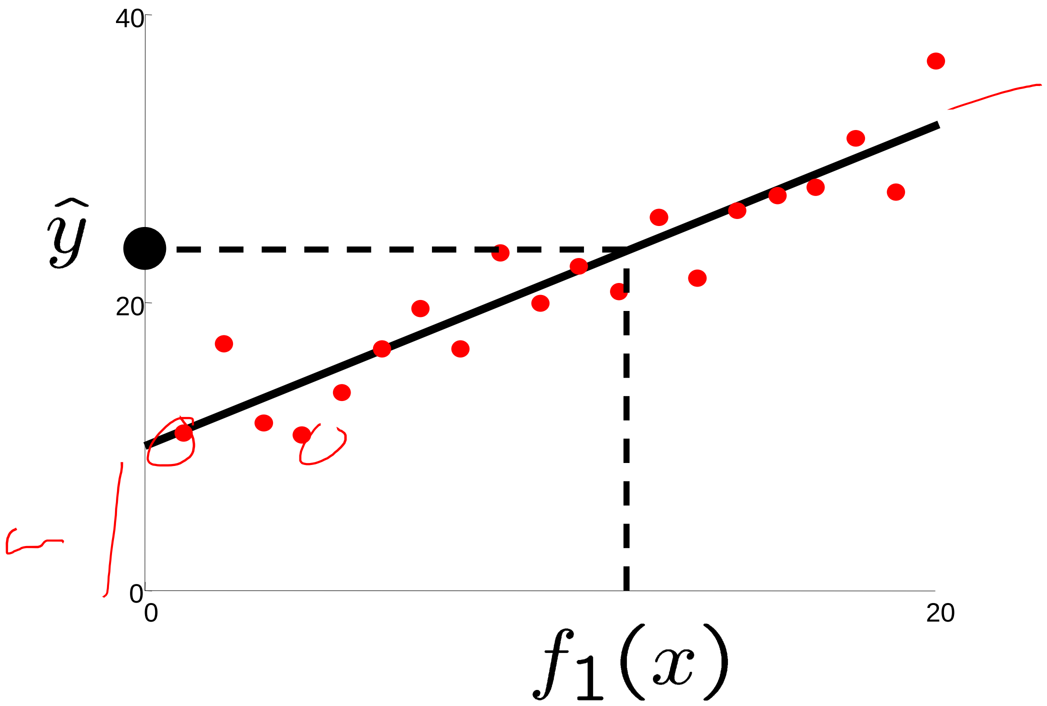
# Video of Demo Approximate Q-Learning -- Pacman

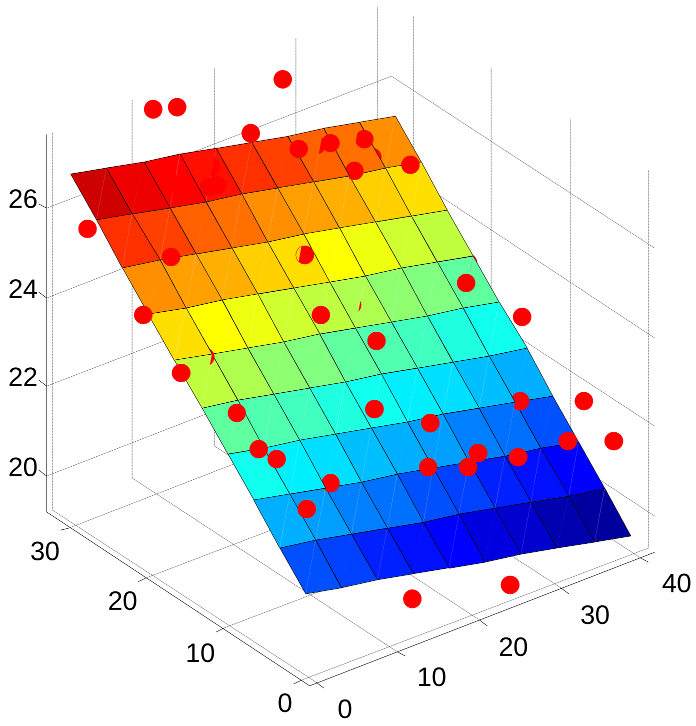


# Q-Learning and Least Squares



# Linear Approximation: Regression




Prediction:

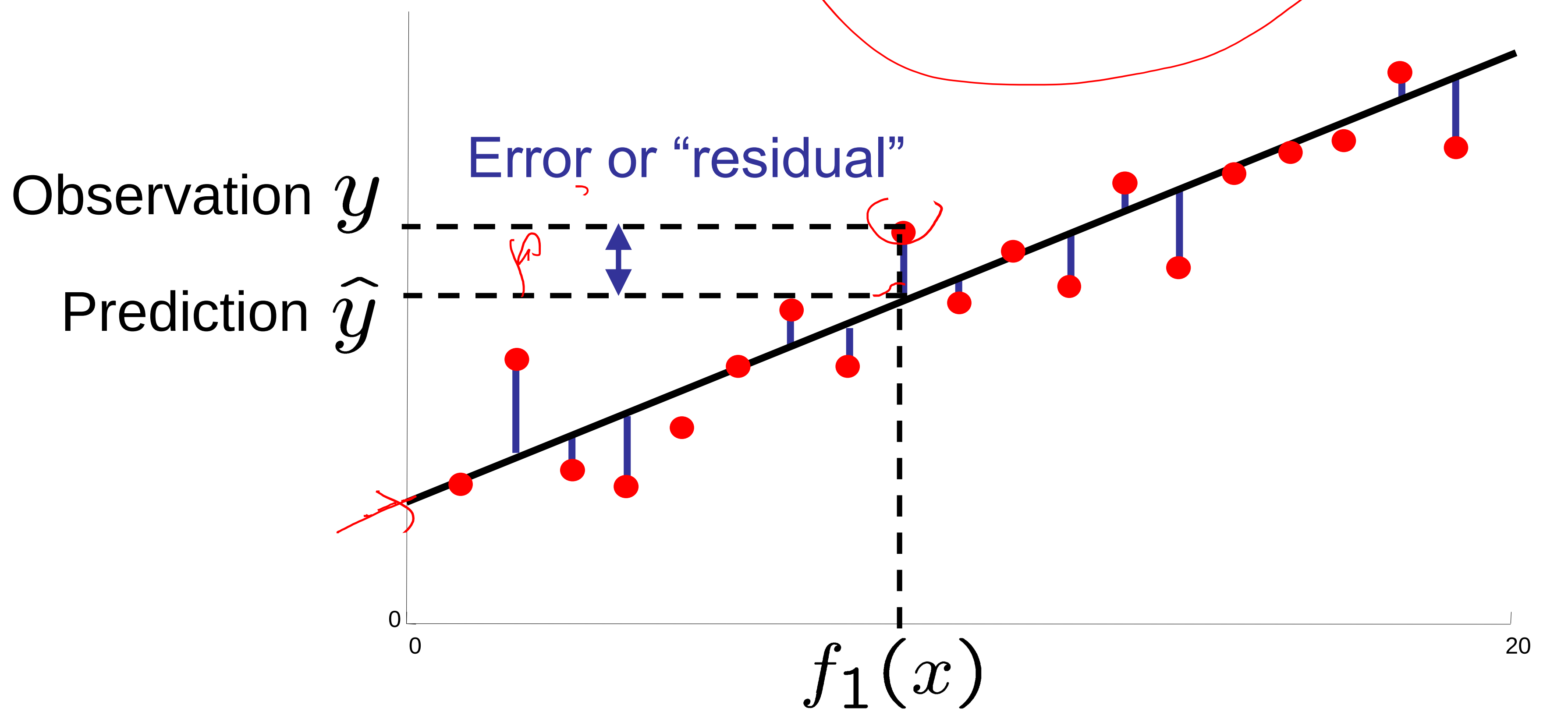
$$\hat{y} = w_0 + w_1 f_1(x)$$

Prediction:

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

# Optimization: Least Squares

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



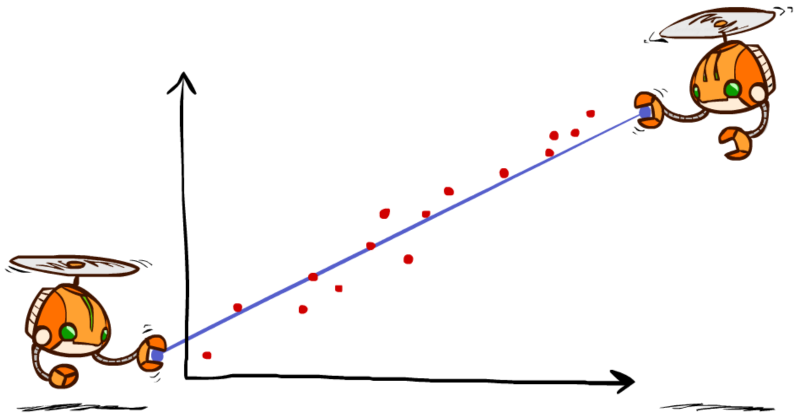
# Minimizing Error

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

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

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

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

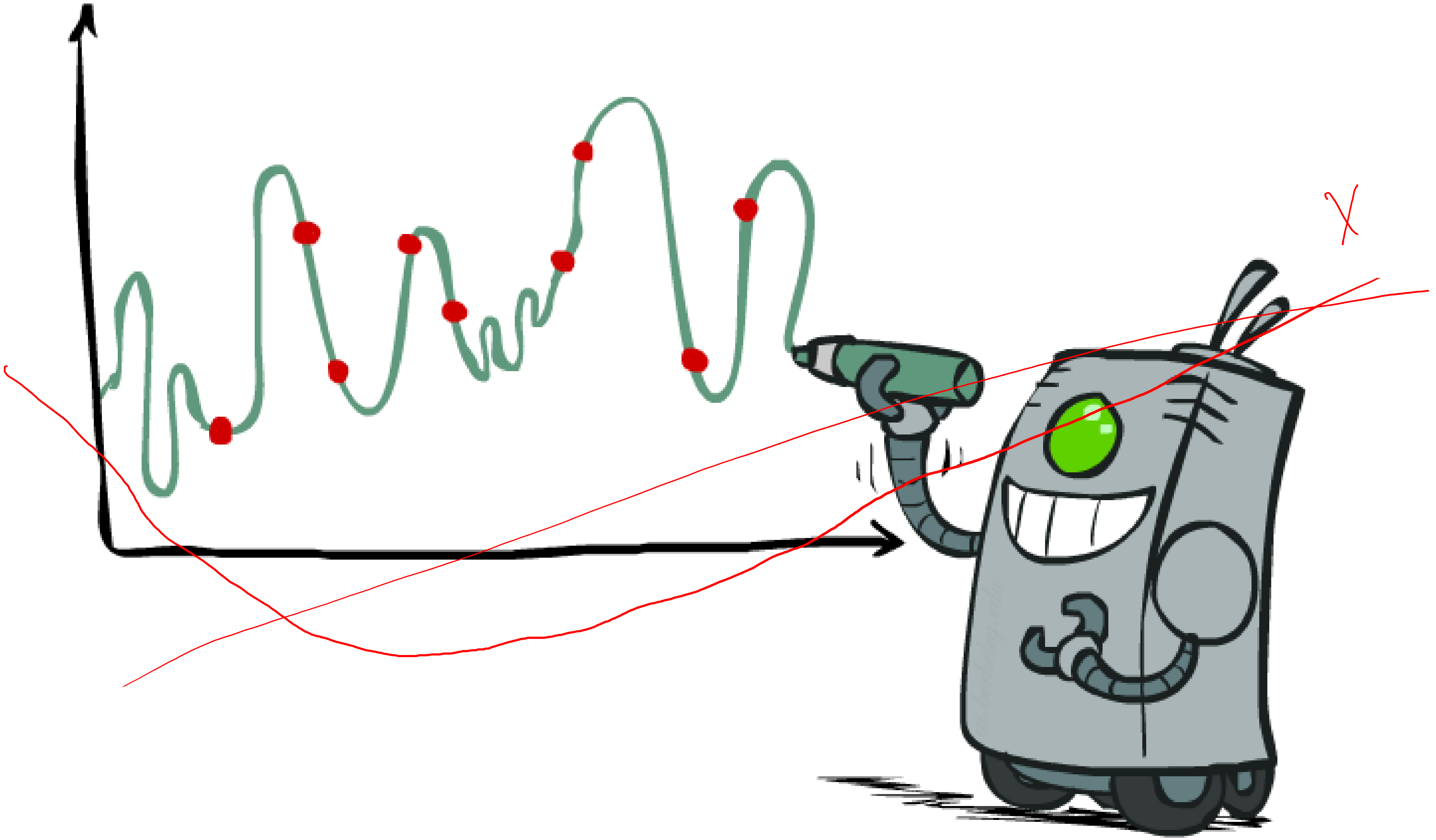


Approximate q update explained:

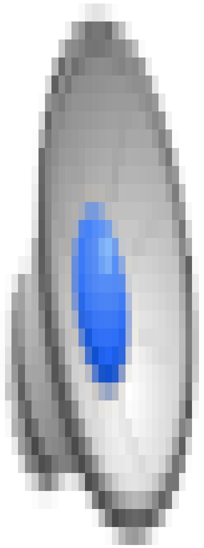
$$w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a') - Q(s, a) \right] f_m(s, a)$$

"target" "prediction"

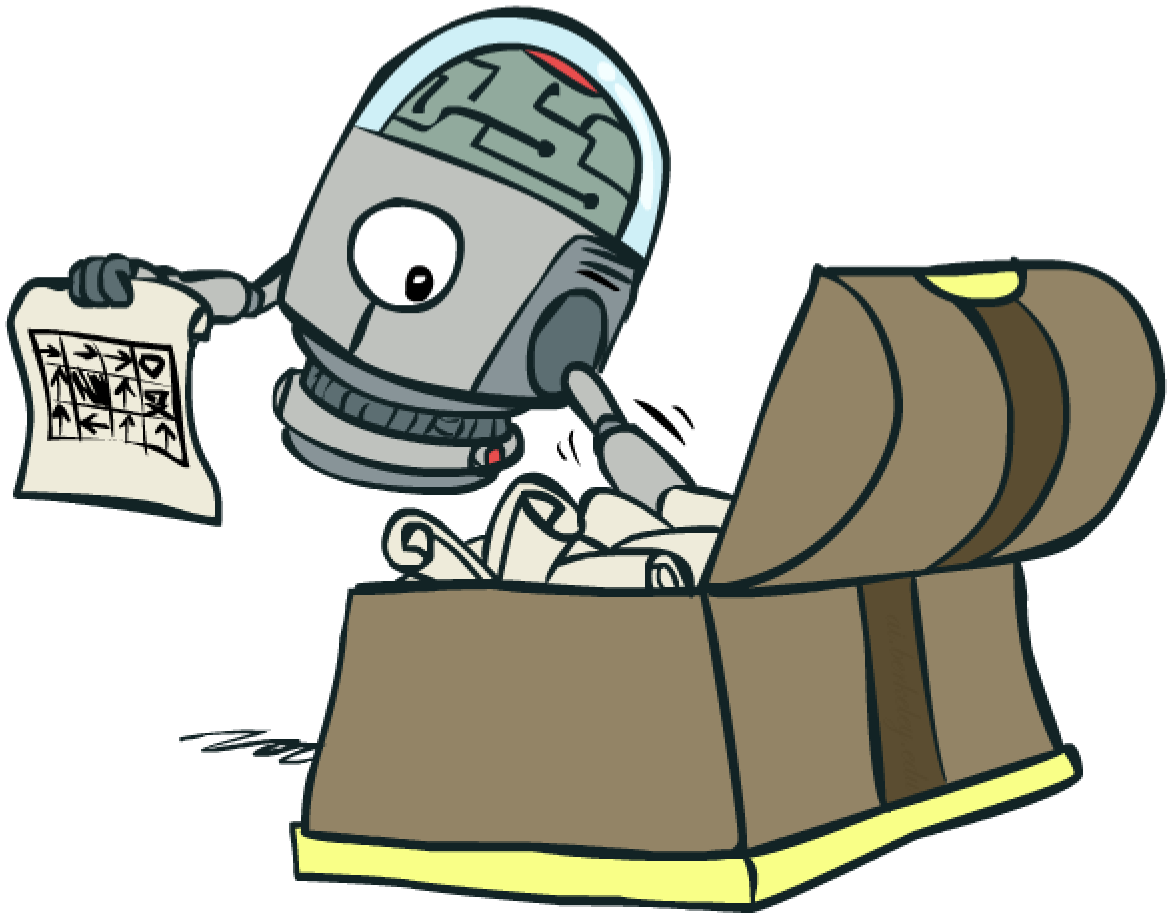
# Overfitting: Why Limiting Capacity Can Help



# New in Model-Free RL
# Playing Atari Games



# 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
  - E.g. your value functions from project 2 were probably horrible estimates of future rewards, but they still produced good decisions
  - 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

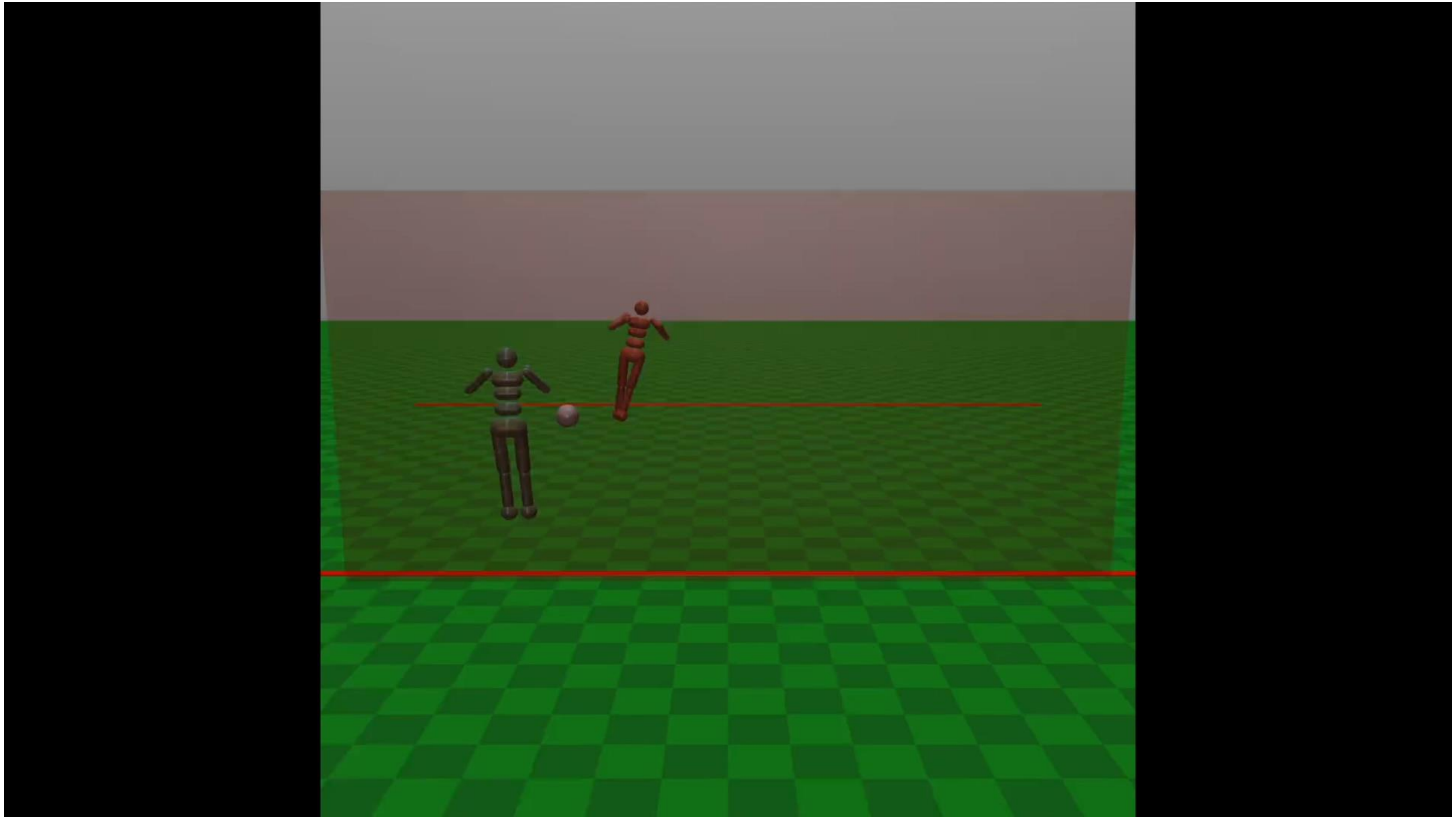
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

# Policy Search

- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - 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…



# Summary: MDPs and RL

## Known MDP: Offline Solution

| Goal | Technique |
|---|---|
| Compute V*, Q*, $\pi$* | Value / policy iteration |
| Evaluate a fixed policy $\pi$ | Policy evaluation |

## Unknown MDP: Model-Based

| Goal | *use features to generalize | Technique |
|---|---|---|
| Compute V*, Q*, $\pi$* | | VI/PI on approx. MDP |
| Evaluate a fixed policy $\pi$ | | PE on approx. MDP |

## Unknown MDP: Model-Free

| Goal | *use features to generalize | Technique |
|---|---|---|
| Compute V*, Q*, $\pi$* | | Q-learning |
| Evaluate a fixed policy $\pi$ | | Value Learning |

# Conclusion

- We've seen how AI methods can solve problems in:
  - Search
  - Games
  - Markov Decision Problems
  - Reinforcement Learning
- Next up: Uncertainty and Learning!