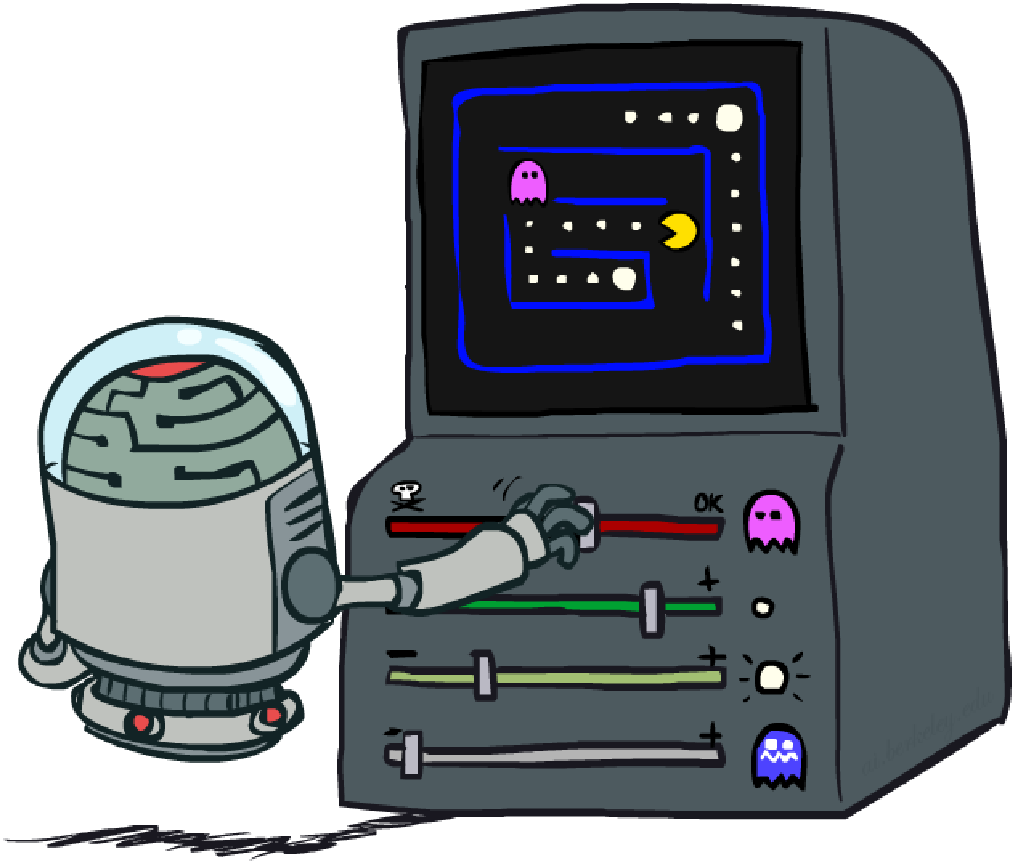
CS 188: Artificial Intelligence

Reinforcement Learning II



[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

Announcements

- Midterm
  - Wednesday March 19, 7-9pm
  - Use this form to request alternate times by 2/27/25 (Thursday) at 11:59 PT
  - Check Ed and Calendar for more midterm logistics/prep sessions, and see exam logistics page near top of course web site for more info.
- HW3
  - Due on Wednesday 2/26/25 at 11:59 PT
- Project 3
  - Due on Friday 3/7/25 at 11:59 PT

Reinforcement Learning: Overview of this week

Last Lecture:

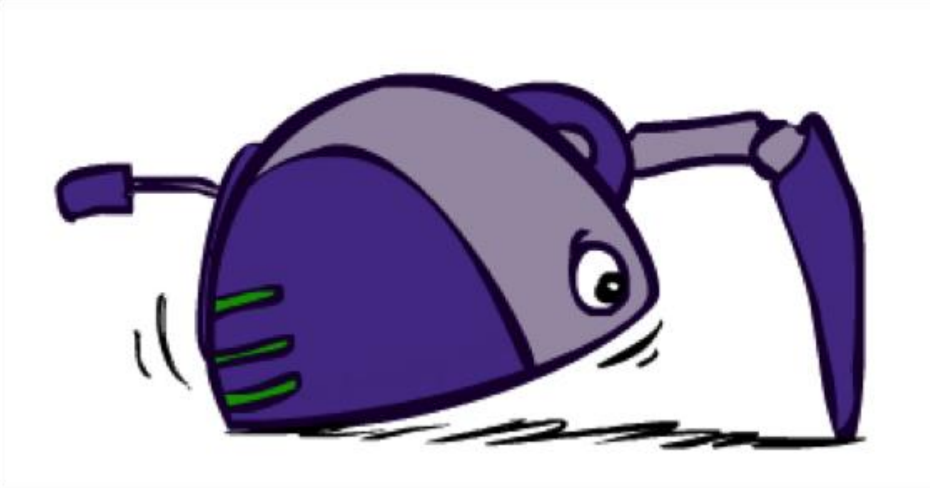
- **Passive Reinforcement Learning:** how to learn from already given experiences
- **Active Reinforcement Learning:** how to collect new experiences

This Lecture:

- **Recap**
- **Approximate Reinforcement Learning:** to handle large state spaces
- **Case studies:** game playing, robot locomotion, language assistants

Recap: 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

Recap: State Spaces and Transition Models

- State Space |S| = 12

T(s,a,s') is a 12x4x12 dimensional table.

- State Space |S| ~ 10^{16} without dynamics
  State Space |S| ~ 10^{32} with dynamics

- 

T(s,a,s') is a 10^{16} x 10^{16} x ?? -dimensional table.

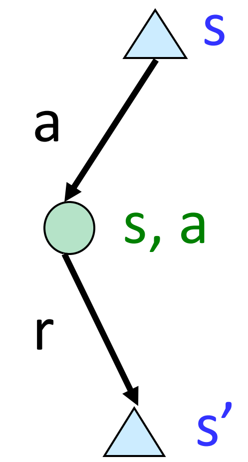
But its easy to generate samples (s,a,s') from T

Recap: Model-Free Learning

- Model-free (temporal difference) learning
  - Receive stream of experiences from the world:

    $$\boxed{(s, a, r, s',}$$

  - Update estimates each transition (s, a, r, s')

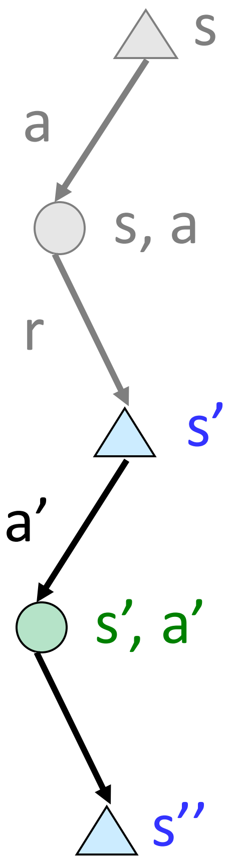


Recap: Model-Free Learning

- Model-free (temporal difference) learning
  - Receive stream of experiences from the world:

$$(s, a, r, \boxed{s', a', r', s''}$$

  - Update estimates each transition (s, a, r, s')



Recap: On-Policy vs Off-Policy

- **Policy Evaluation**:
  - The Value function for a policy

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

  - Is an on-policy estimate – it gives the value for the policy generating trajectories.
- **Value Iteration:**

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

  - Is an off-policy estimate – it gives the value for a different policy from the one that generated trajectories.

Recap: Q-Learning

- **Q-Iteration:** do Q-value updates to each Q-state:
  - Initialize $Q_0(s,a) = 0$, then iterate:

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

  - But can't compute this update without knowing T, R

- **Q-Learning:** Instead, compute average as we go
  - Receive a sample transition (s,a,r,s')
  - This sample suggests:

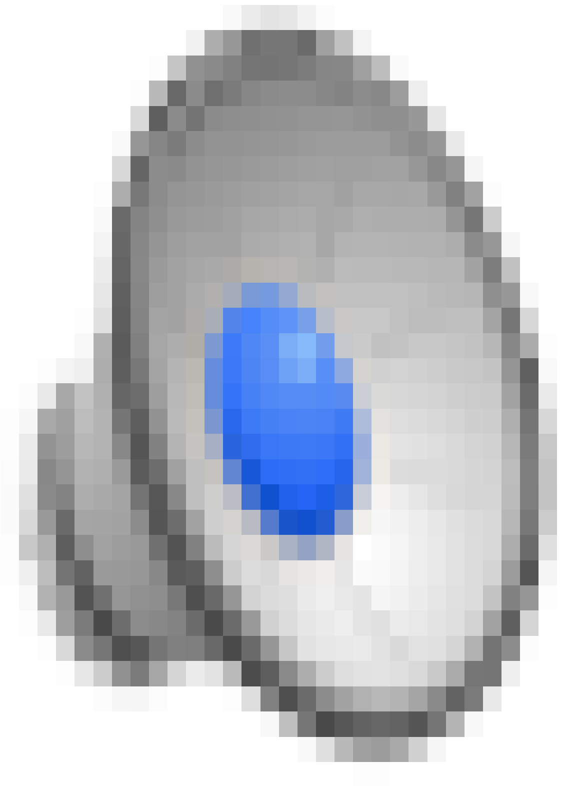
$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

  - But we want to average over multiple outcomes from (s,a)
  - So keep a running average:

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

[Demo: running average]

Video of Demo Q-Learning -- Gridworld

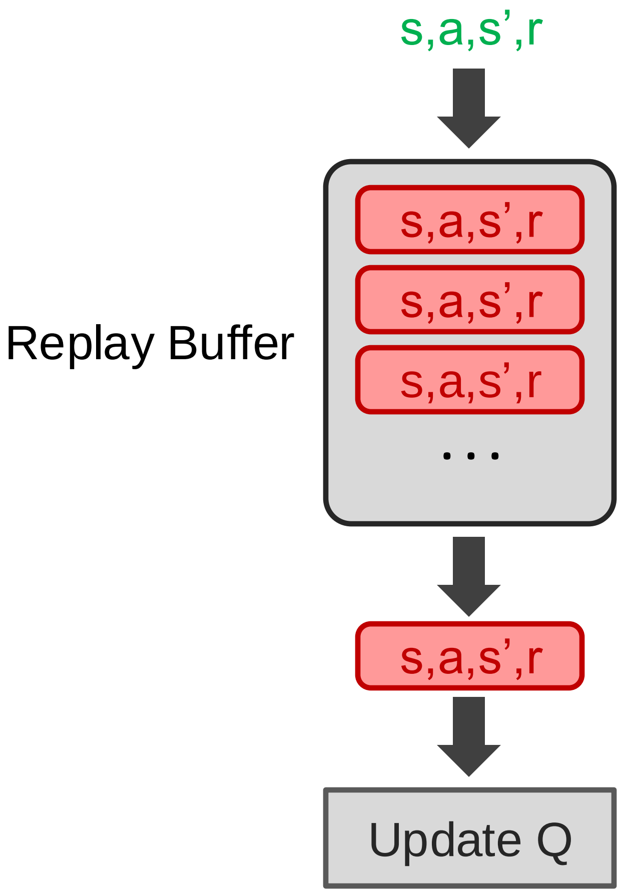


- At each step:
  - Receive a sample transition *(s,a,s',r)*
  - Update running average:

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

Q-Learning with Experience Replay

- Problem:
  - Need to repeat same (s,a,s’,r) transitions in environment many times to propagate values
- Solution:
  - Collect transitions in a memory buffer and “replay” them to update Q values
    - Uses memory of transitions only, no need to repeat them in environment
  - Evidence of such experience replay in the brain



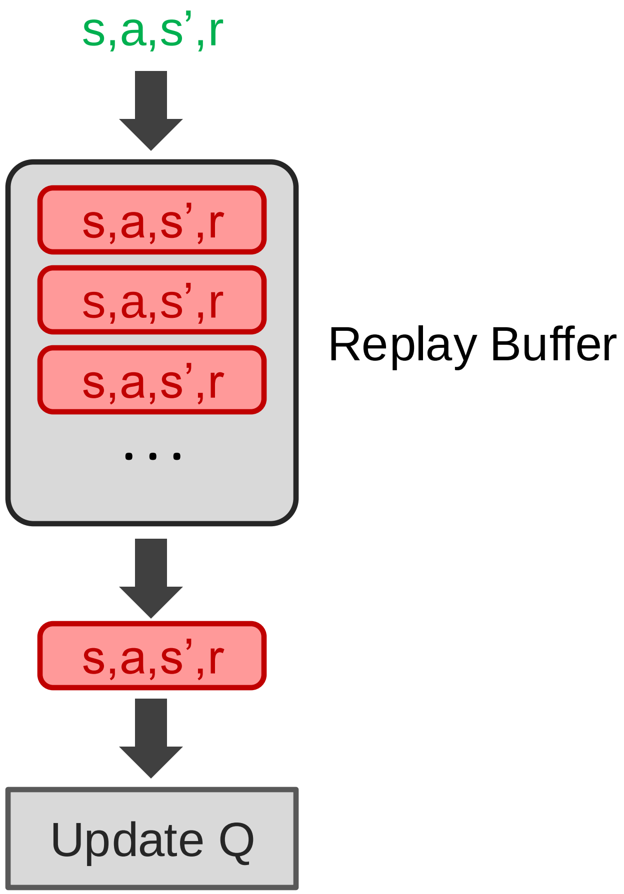
Q-Learning with a Experience Replay

- At each step:
  - Receive a sample transition *(s,a,s',r)*
  - Add *(s,a,s',r)* to replay buffer
  - Repeat N times:
    - Randomly pick transition *(s,a,s',r)* from replay buffer
    - Make sample based on *(s,a,s',r)*:

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

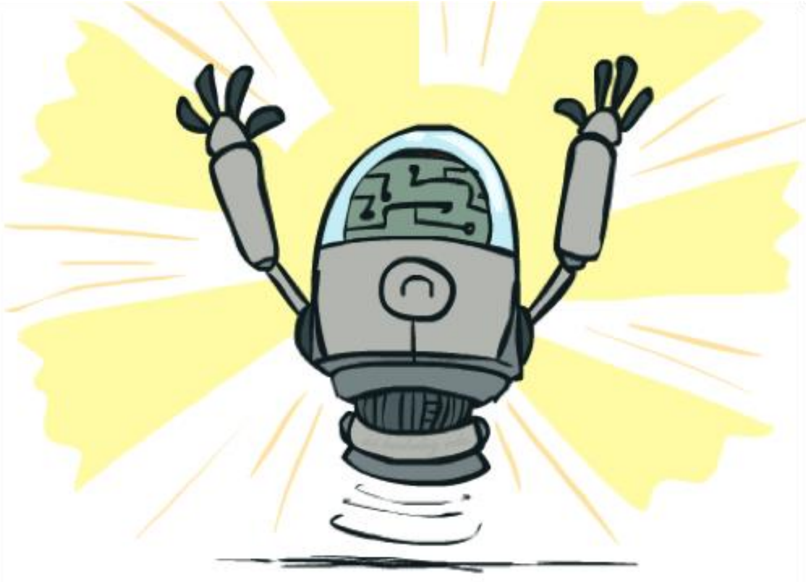
    - Update Q based on picked sample:

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



Q-Learning Properties

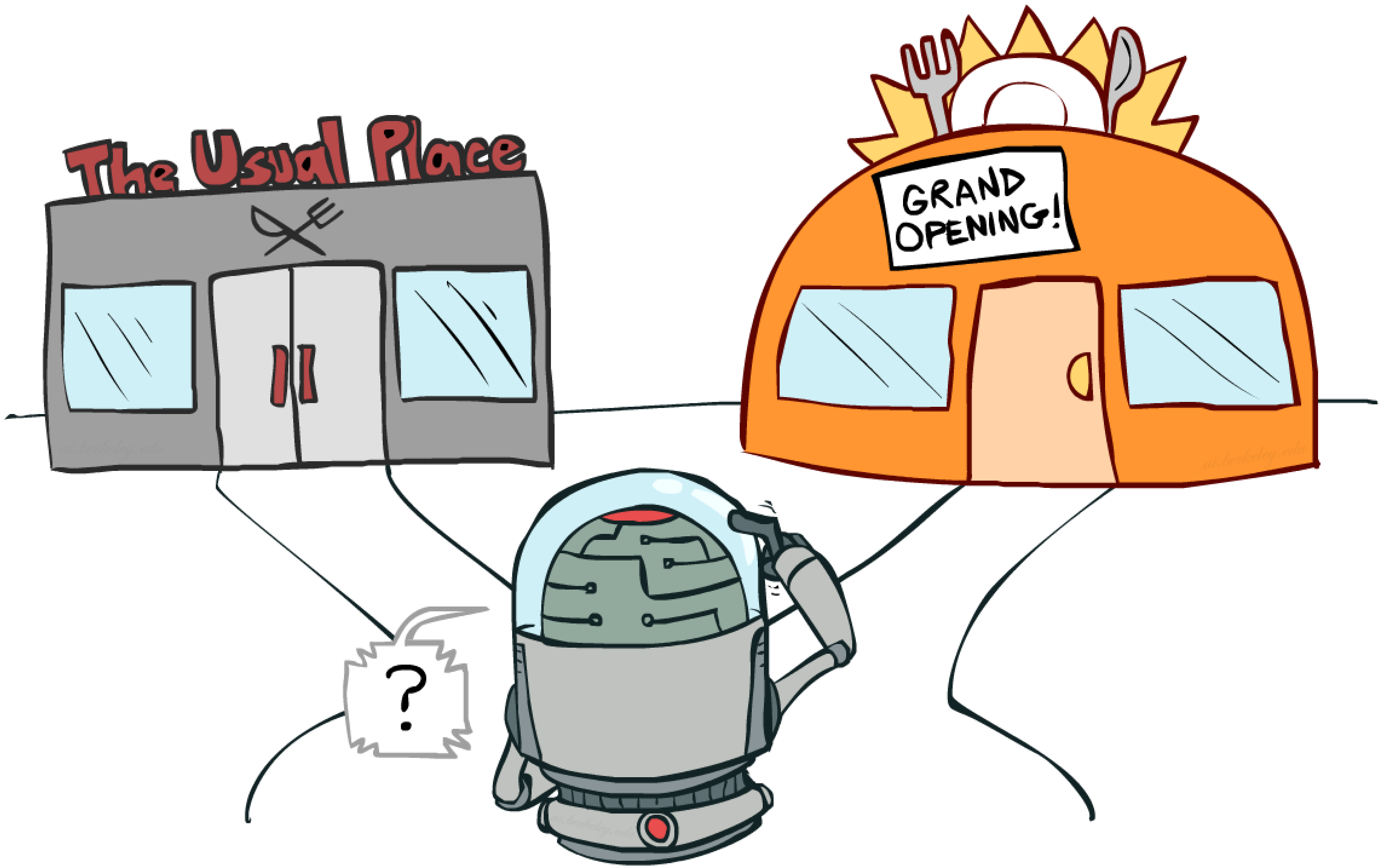
- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- Gives us optimal way to act! $\pi^*(s) = \arg\max_a Q(s,a)$
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough (but not decrease it too quickly)
  - Basically, in the limit, it doesn't matter how you select actions (!)



[Demo: Q-learning – auto – cliff grid (L11D1)]

Video of Demo Q-Learning Auto Cliff Grid

Recap: Exploration vs. Exploitation

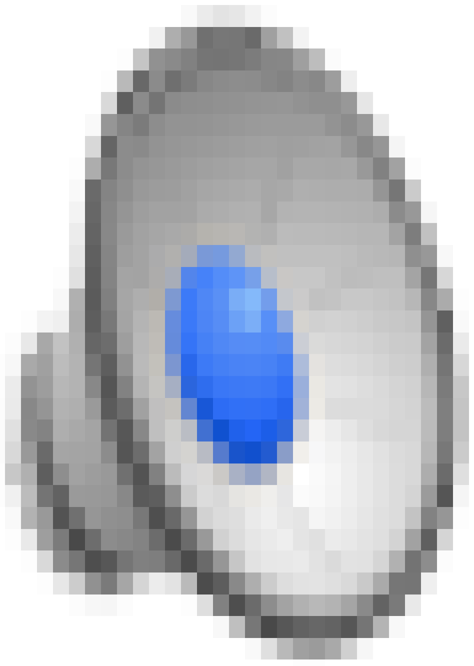


How to Explore?

- 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

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

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

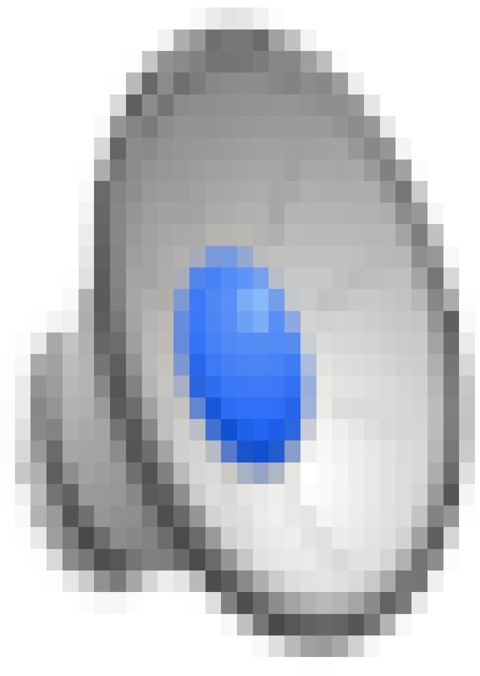
Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

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

$x \leftarrow_\alpha v$ is shorthand for $x \leftarrow (1 - \alpha)x + \alpha v$

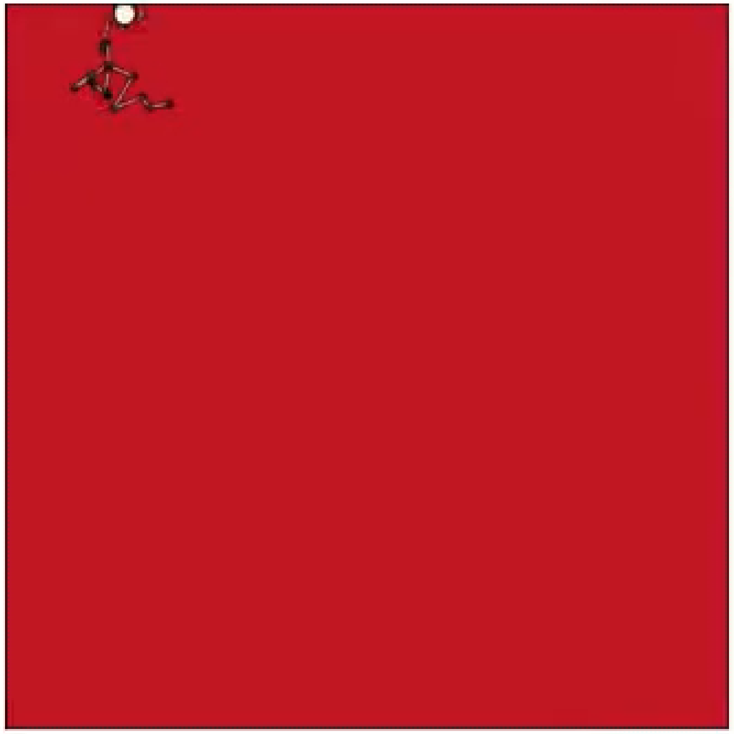
[Demo: exploration – Q-learning – crawler – exploration function (L11D4)]

Video of Demo Q-learning – Exploration Function – Crawler

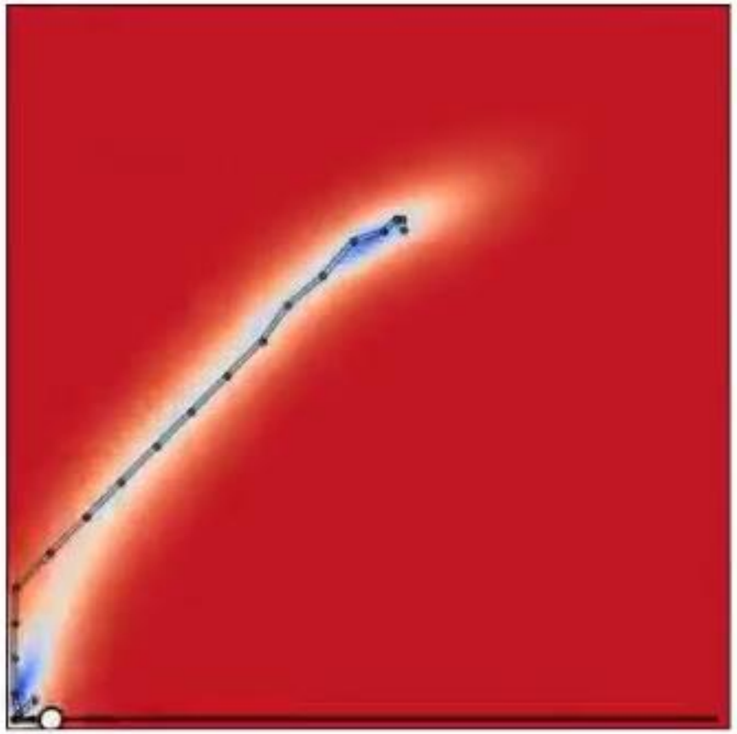


Random Actions vs Exploration Functions

Random Actions



Exploration Function

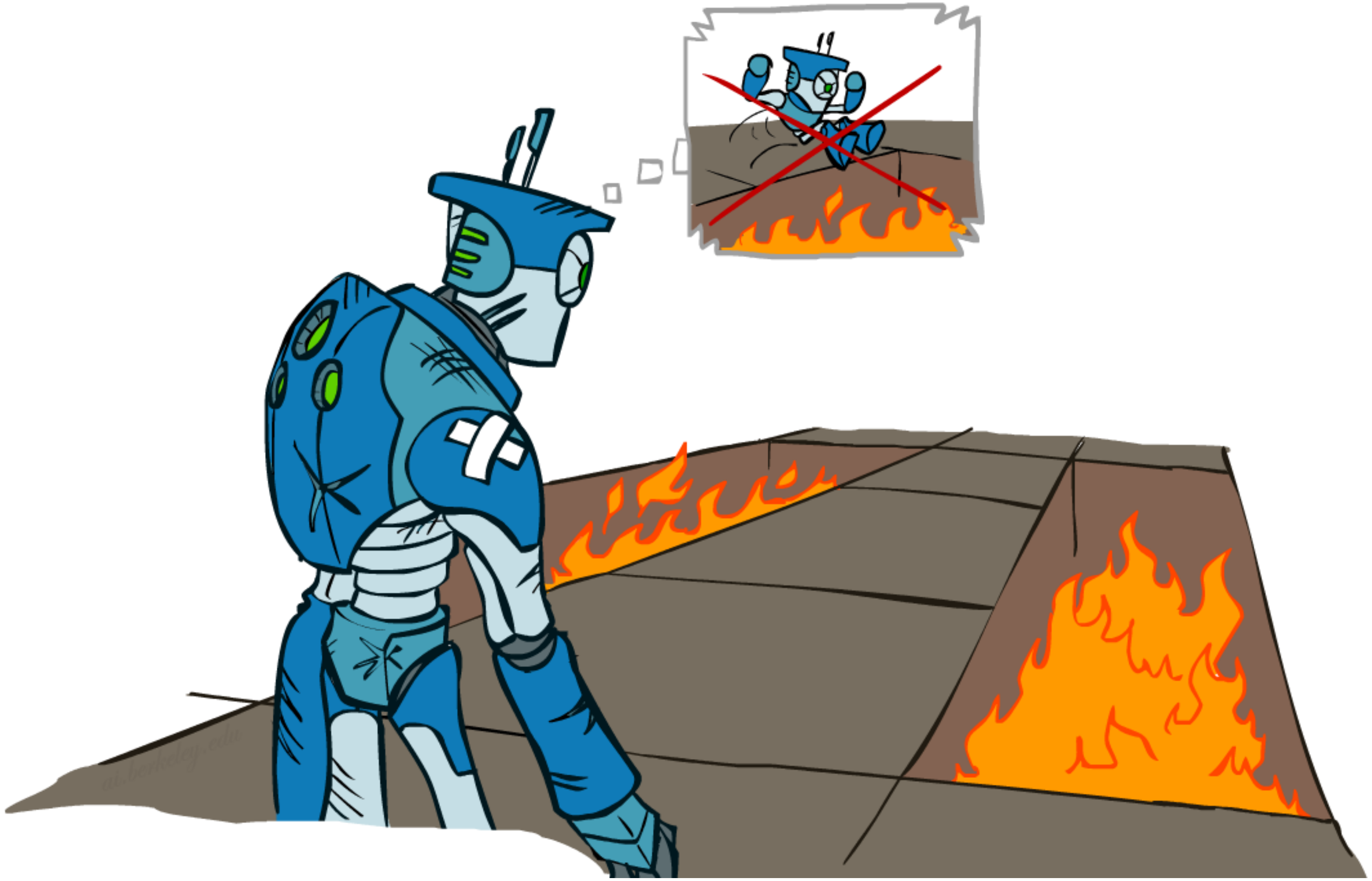


Blue: more visited

Red: less visited

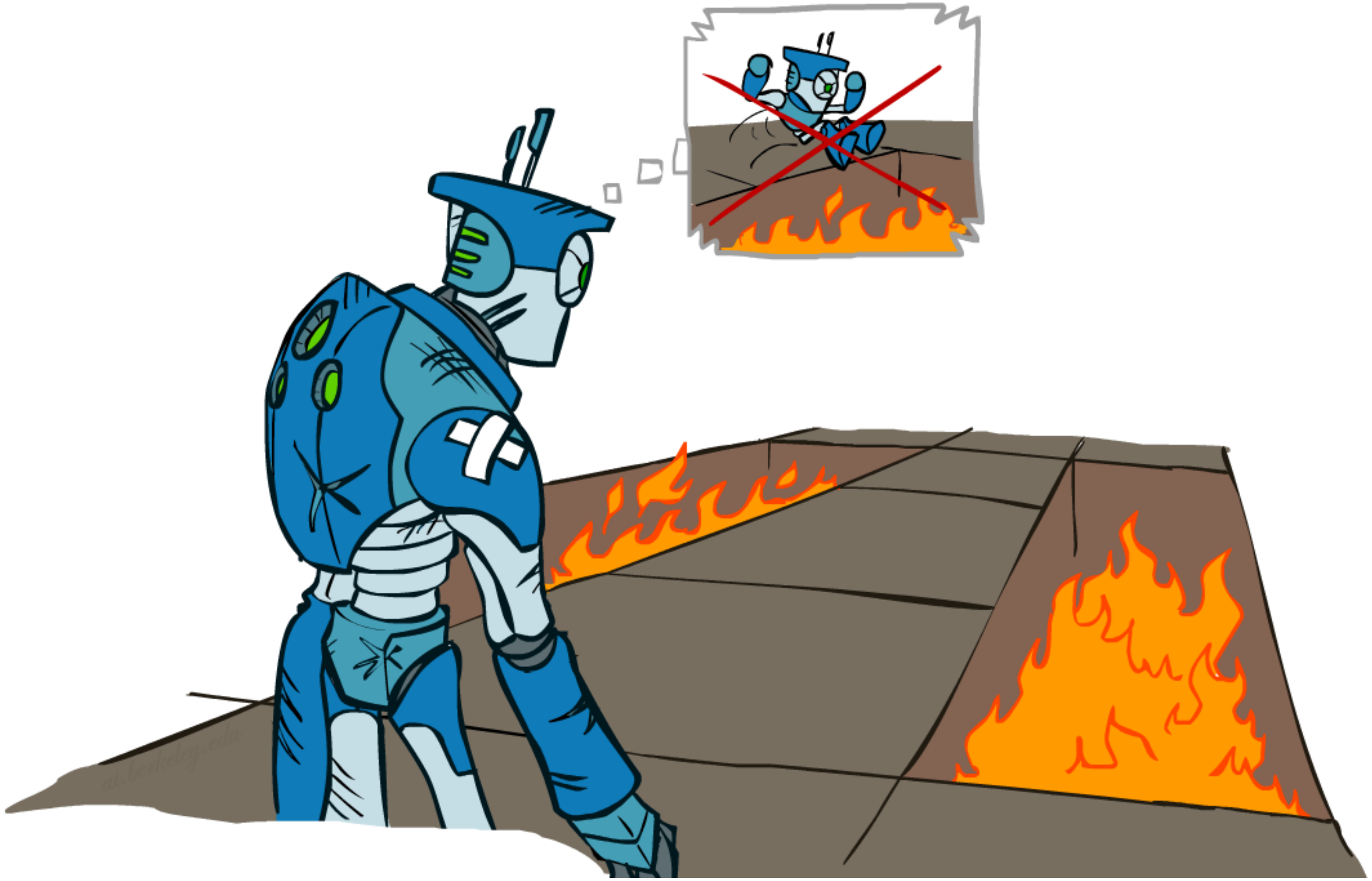
[Plan Online, Learn Offline, Lowrey et al, 2019]

How can we evaluate RL Methods?



Regret

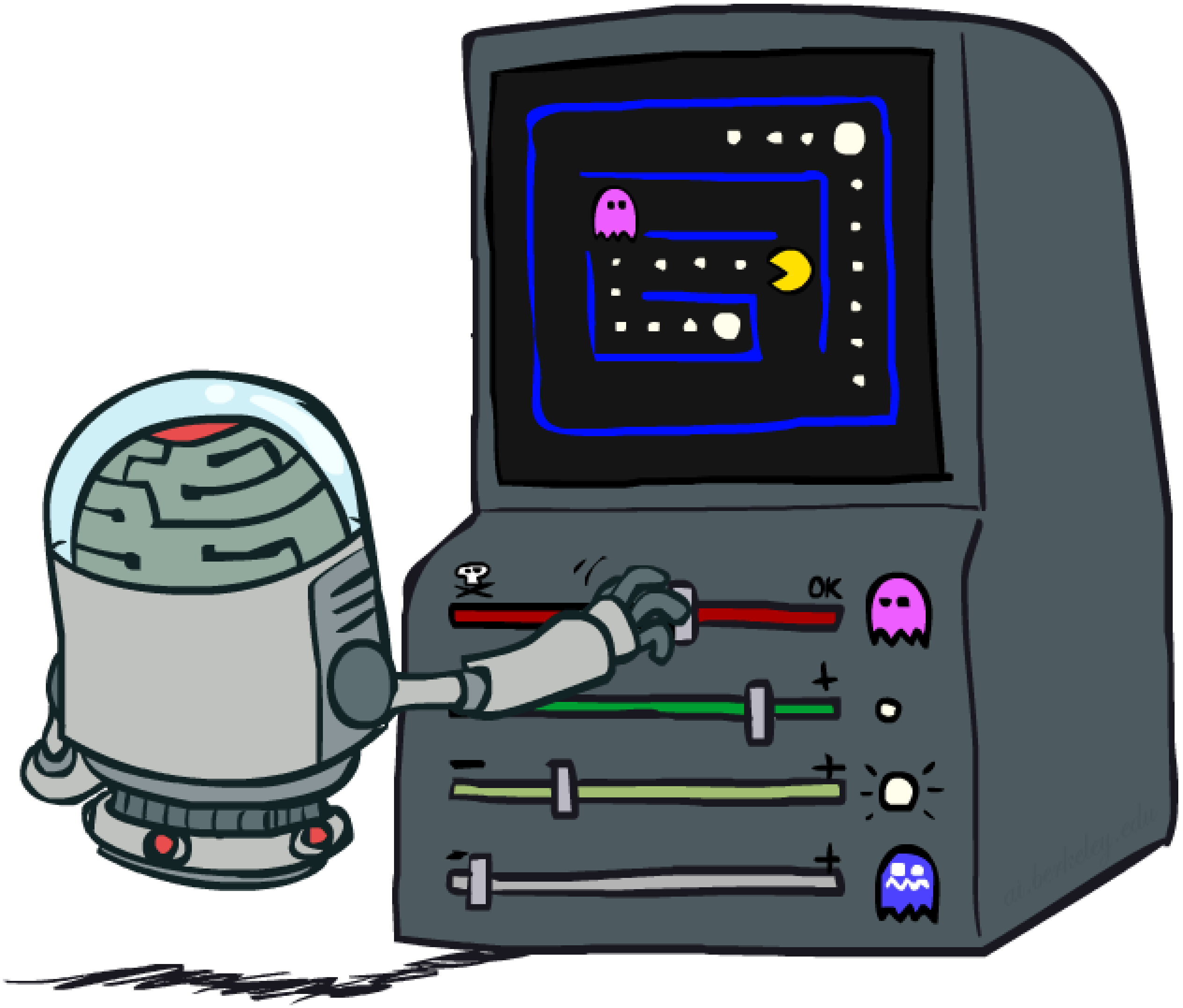
- Even if you learn the optimal policy, you still make mistakes along the way
- *Regret* is a measure of your total mistake cost:
  - Difference between all your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal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



Are We Done?

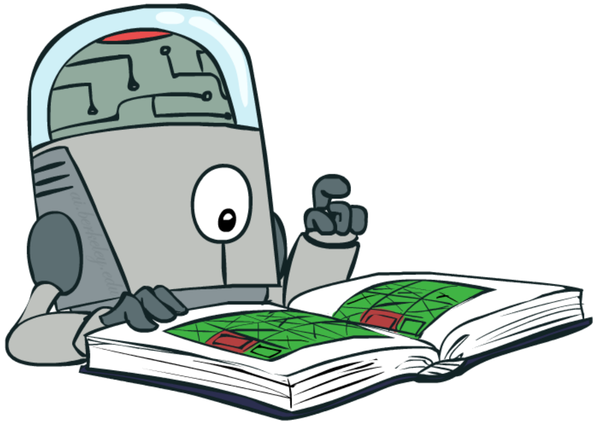
- Large and complex state spaces are still a problem!

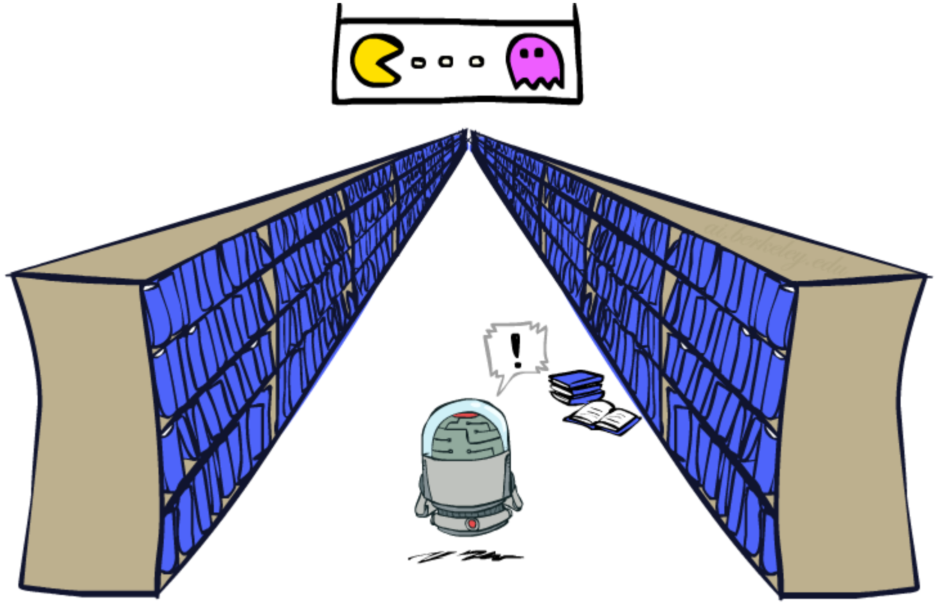
Approximate Q-Learning



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]

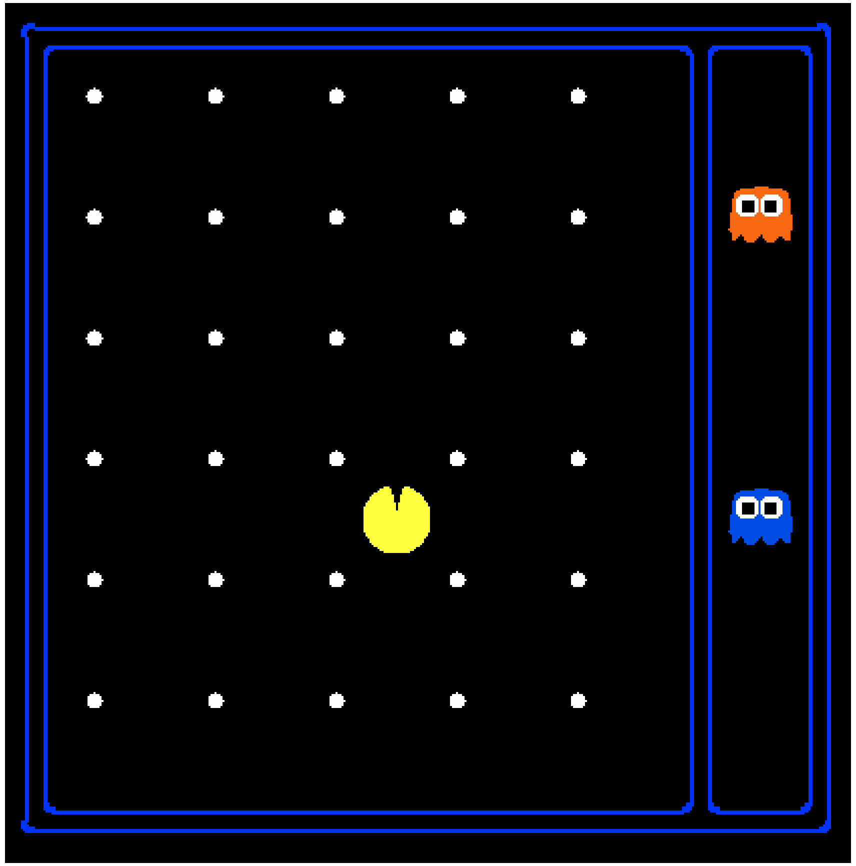
Recall Lecture 2: State Space Sizes

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW
- How many
  - World states?

    $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing?

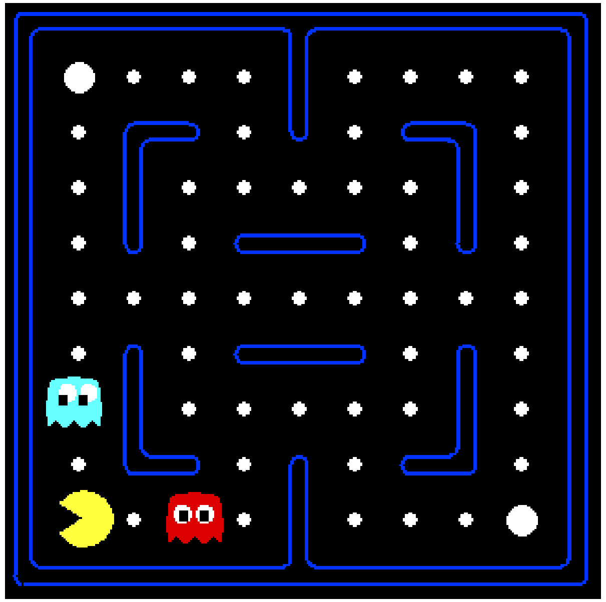
    120
  - States for eat-all-dots?

    $120 \times (2^{30})$

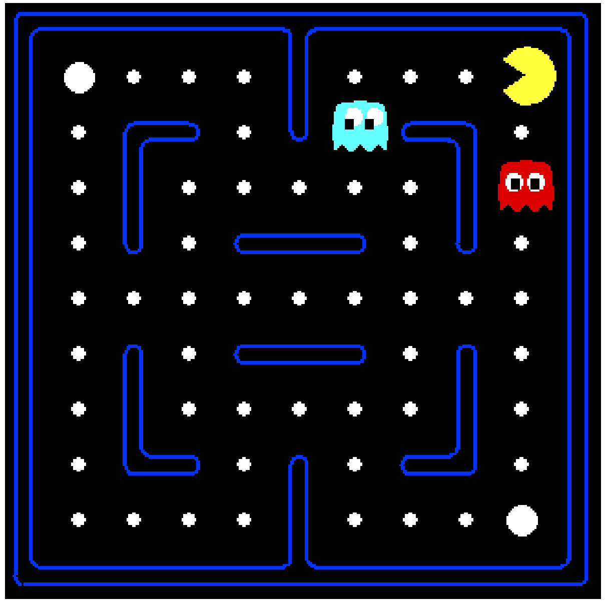


Example: Pacman

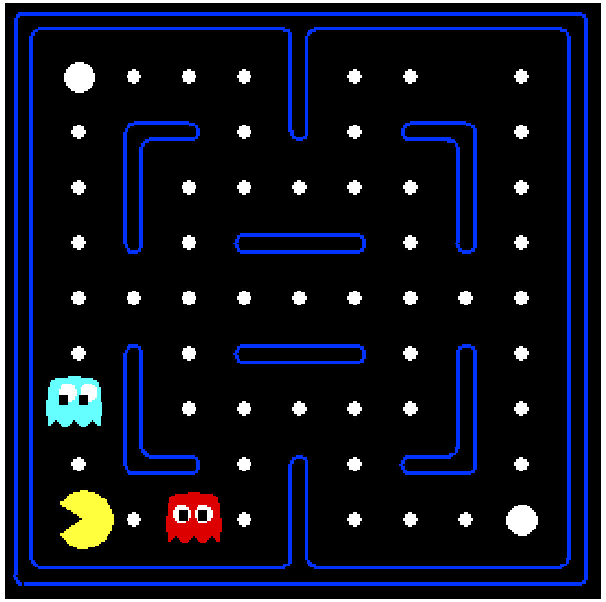
Let's say we discover through experience that this state is bad:



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



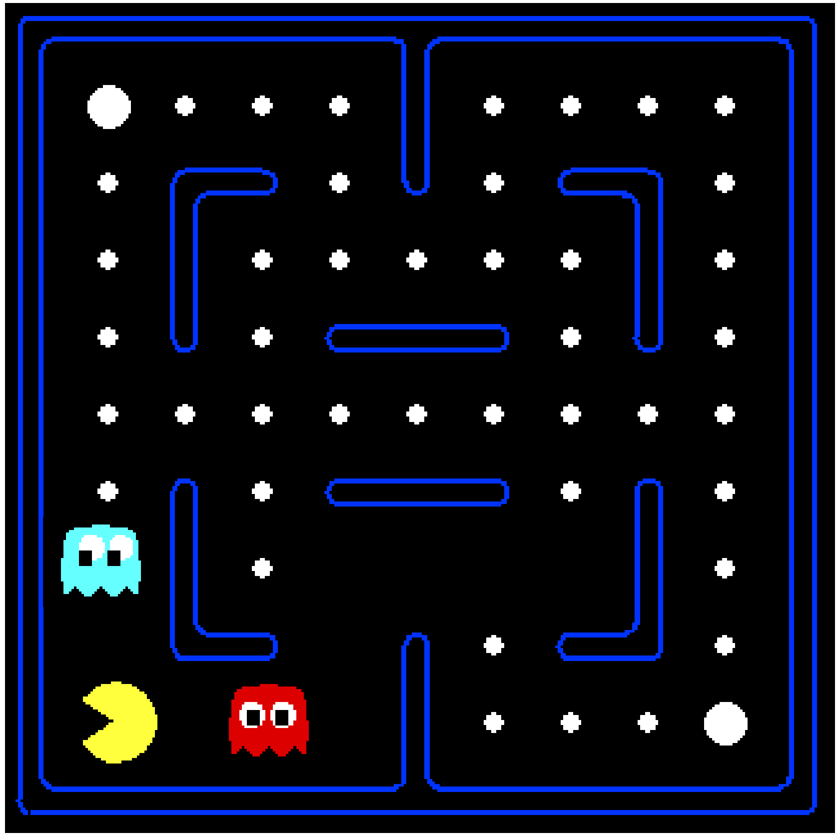
Or even this one!



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

Feature-Based Representations

- Solution: describe a state using a vector of features (properties) f_1, f_2, …
  - Features are functions from states to real numbers (often in [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)



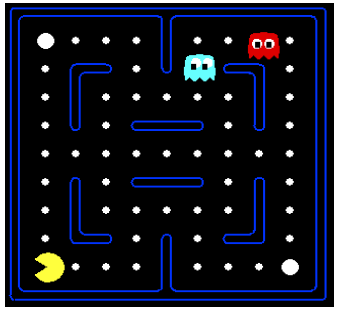
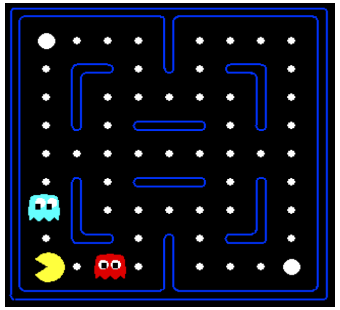
Linear Value Functions

- Using a feature representation f_1, f_2, … we can write a q function (or value function) for any state using a few weights w_1, w_2, … :

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

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

- Advantage: our experience is summed up in a few powerful numbers w_1, w_2, …
- Disadvantage: states may share features but actually be very different in value!
  - Ex: these two states would have the same value if we don’t include ghost positions as a feature:

Approximate Q-Learning

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

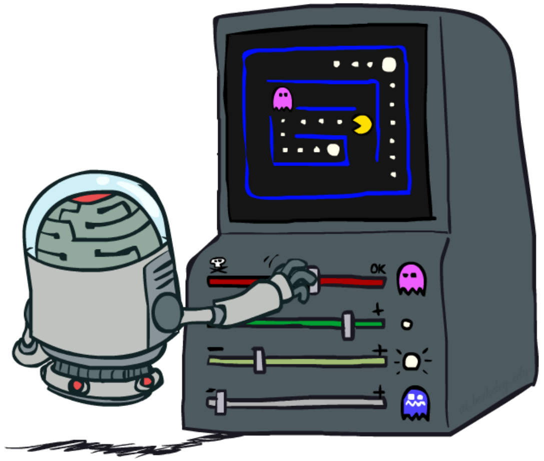
- Q-learning with linear Q-functions:

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

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

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

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

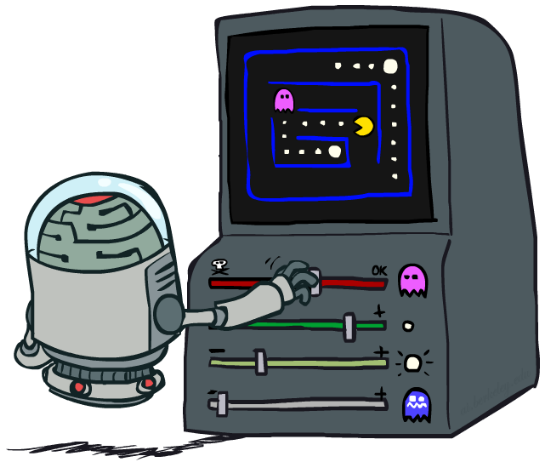
Approximate Q-Learning

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

- Q-learning with linear Q-functions:

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

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



- Example: Something unexpectedly good happens, and feature f_2 is on (positive)
  - Raise Q value for current s, a and in the future prefer all states where f_2 is on

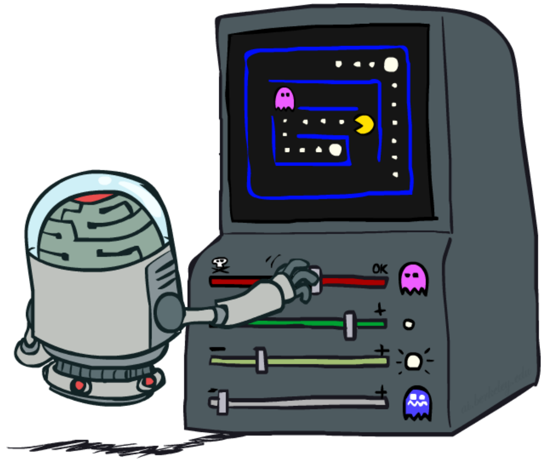
Approximate Q-Learning

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

- Q-learning with linear Q-functions:

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

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

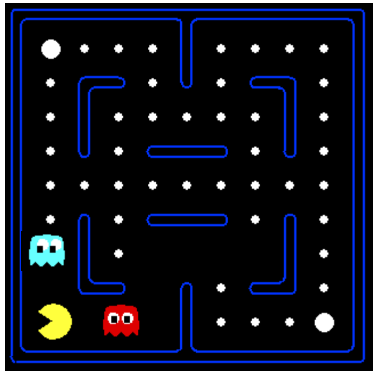


- Example: Something unexpectedly bad happens, and feature f_2 is on (positive)
  - Lower Q value for current s, a and in the future avoid all states where f_2 is on

Example: Q-Pacman

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

s

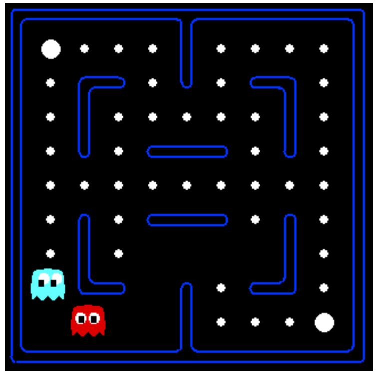


$f_{DOT}(s, \text{NORTH}) = 0.5$

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

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

s'



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

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

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

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

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

$\text{difference} = -501$

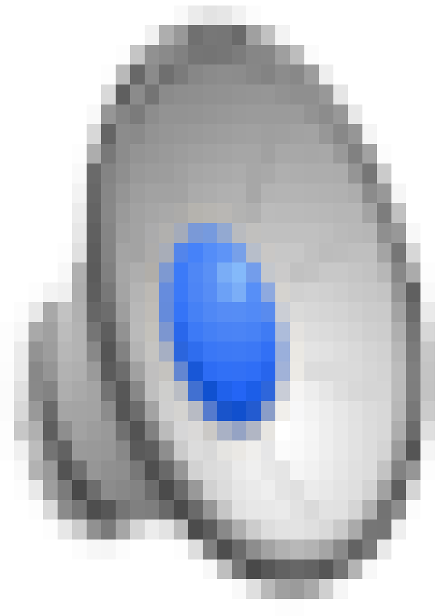
$w_{DOT} \leftarrow 4.0 + \alpha\, [-501]\, 0.5$

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

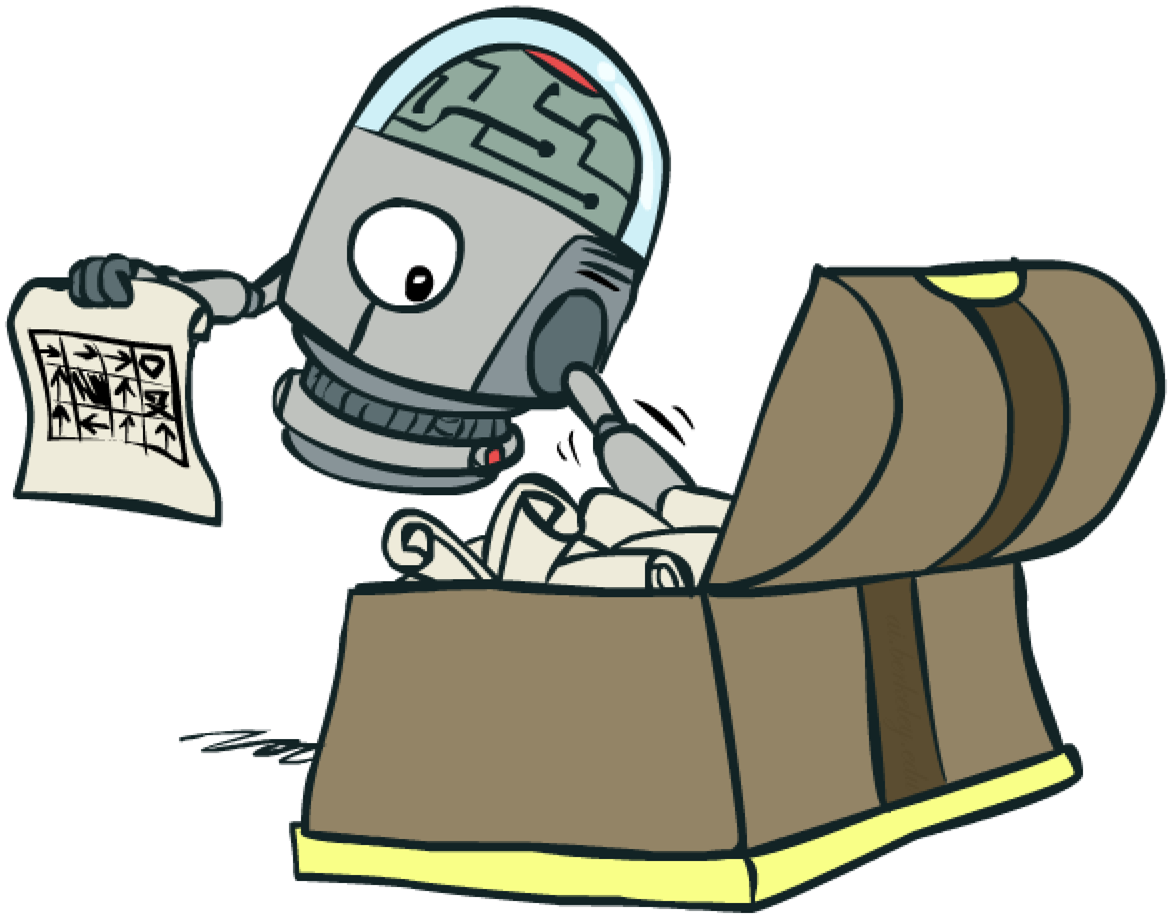
$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

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

Video of Demo Approximate Q-Learning -- Pacman



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 π that maximize rewards, not the Q 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 Gradient*, *Proximal Policy Optimization (PPO)* are examples

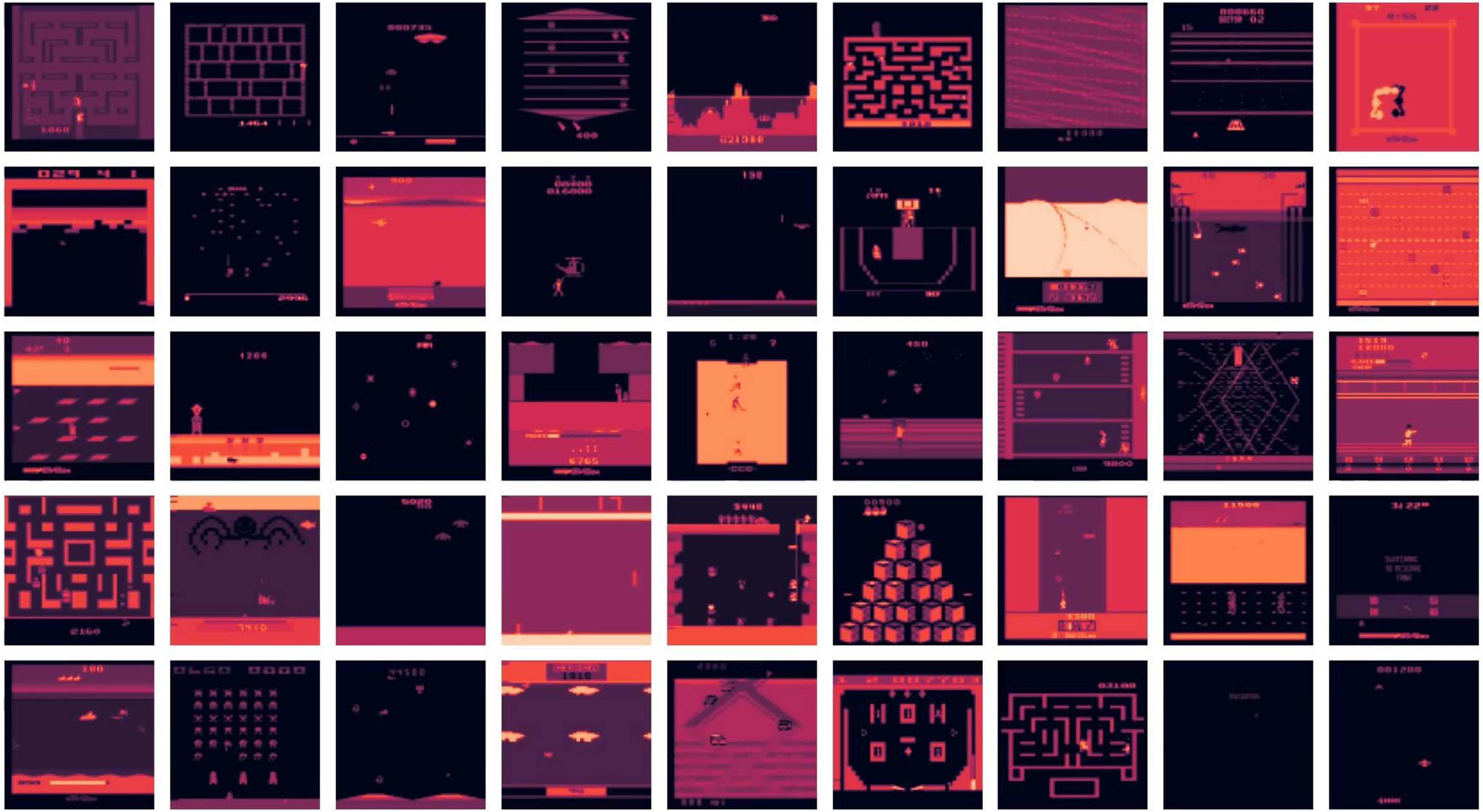
Policy Gradient*

- Simplest version:
  - Start with initial policy $\pi(s)$ that assigns probability to each action
  - Sample actions according to policy π
  - Update policy:
    - If an episode led to high utility, make sampled actions more likely
    - If an episode led to low utility, make sampled actions less likely

Case Studies of Reinforcement Learning!

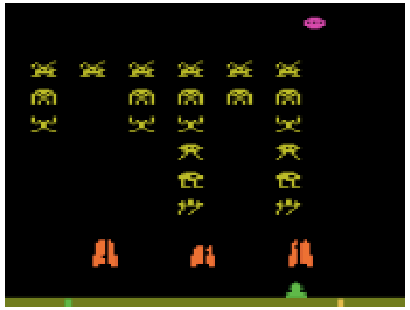
- Atari game playing
- Robot Locomotion
- Language assistants

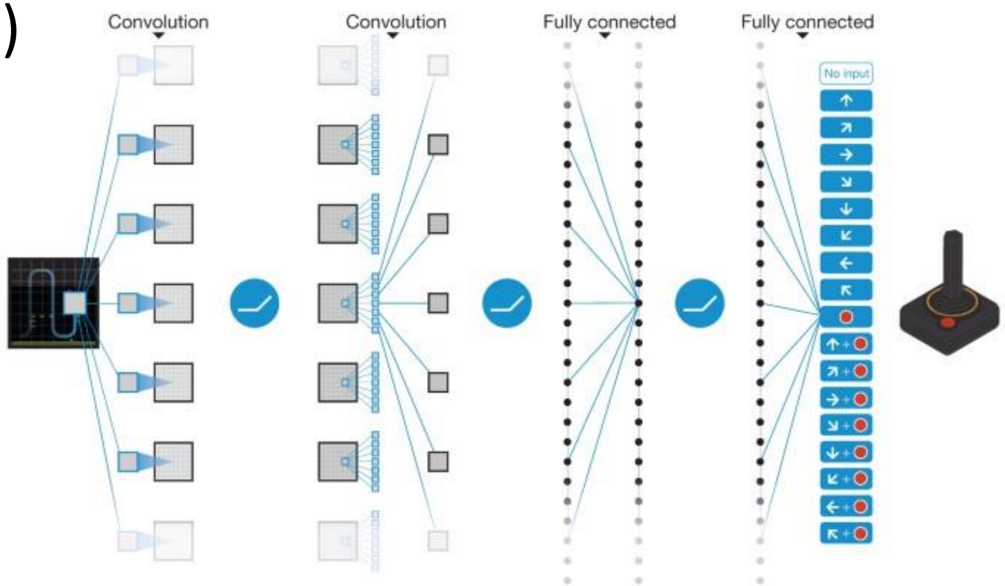
Case Studies: Atari Game Playing



Case Studies: Atari Game Playing

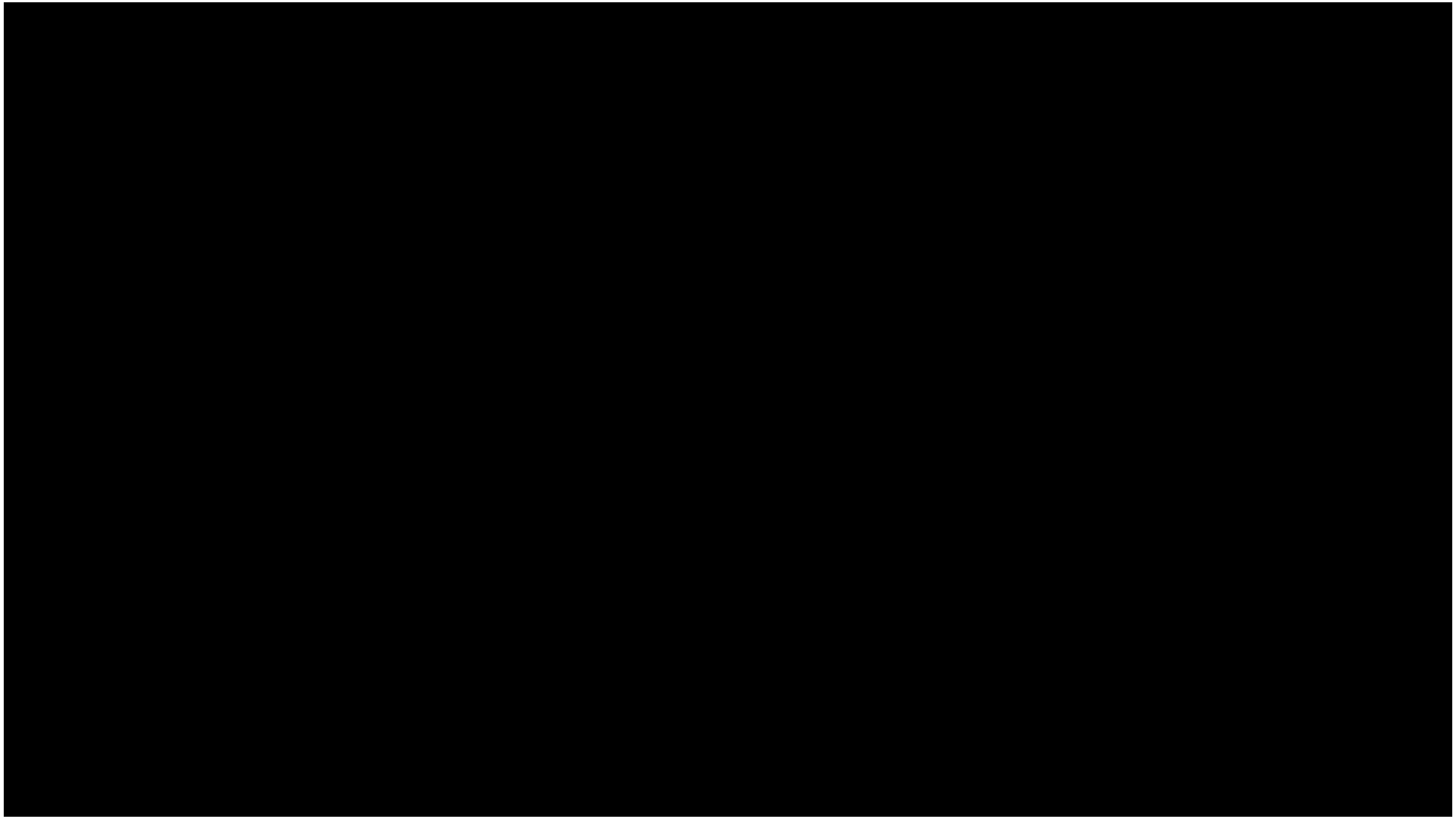
- MDP:
  - State: image of game screen
    - 256^{84*84} possible states
    - Processed with hand-designed feature vectors or neural networks
  - Action: combination of arrow keys + button (18)
  - Transition T: game code (don't have access)
  - Reward R: game score (don't have access)
- Very similar to our pacman MDP
- Use approximate Q learning with neural networks and ε-greedy exploration to solve





[Human-level control through deep reinforcement learning, Mnih et al, 2015]

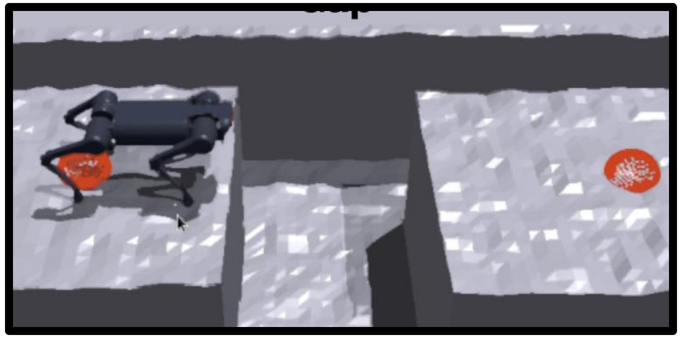
Case Studies: Robot Locomotion



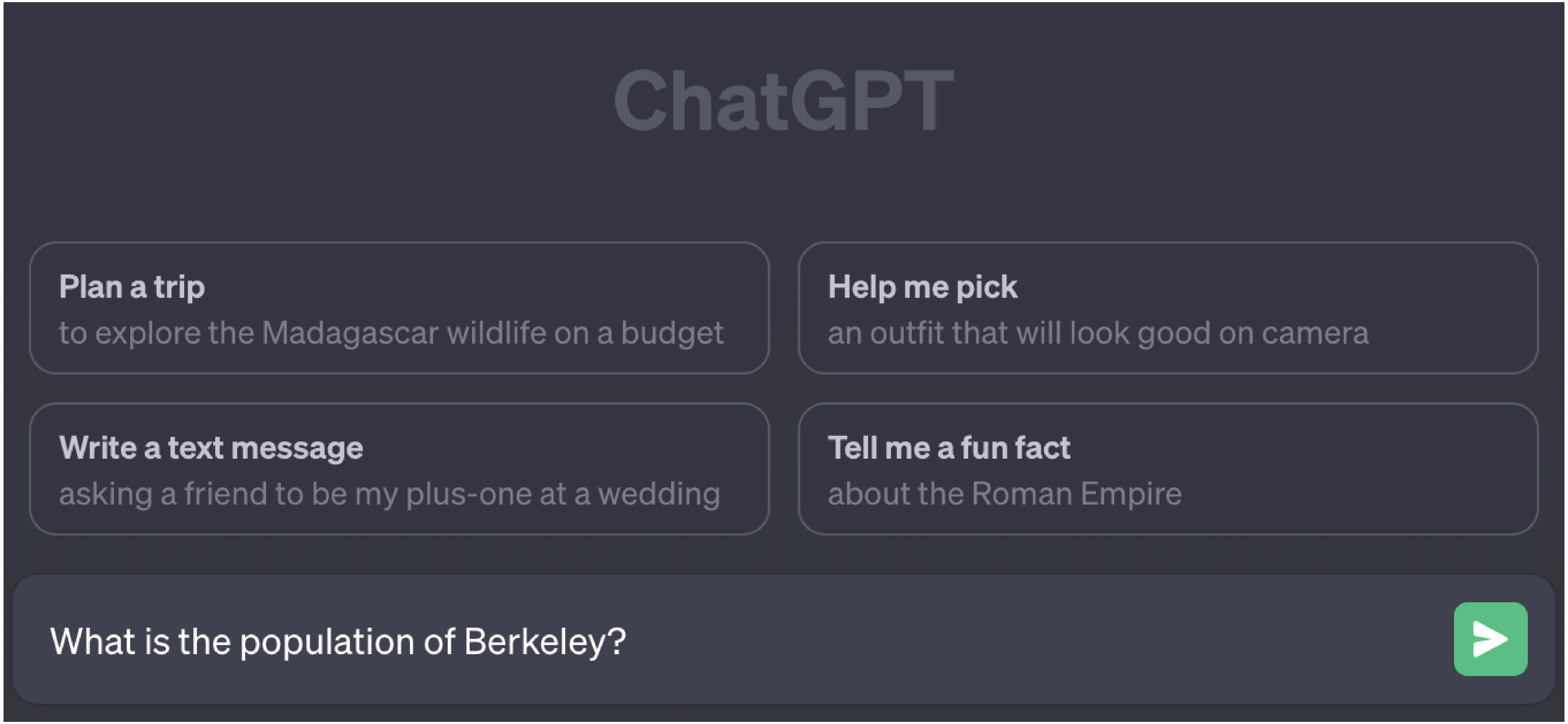
[Extreme Parkour with Legged Robots, Cheng et al, 2023]

Case Studies: Robot Locomotion

- MDP:
  - State: image of robot camera + N joint angles + accelerometer + …
    - Angles are N-dimensional continuous vector!
    - Processed with hand-designed feature vectors or neural networks
  - Action: N motor commands (continuous vector!)
    - Can't easily compute $\max_a Q(s', a)$ when a is continuous
    - Use policy search methods or adapt Q learning to continuous actions
  - Transition T: real world (don't have access)
  - Reward R: hand-designed rewards
    - Stay upright, keep forward velocity, etc
- Learning in the real world may be slow and unsafe
  - Build a simulator and learn there first, then deploy in real world



Case Studies: Language Assistants



[OpenAI]

Case Studies: Language Assistants

- Step 1: train large language model to mimic human-written text
  - **Query:** `"What is population of Berkeley?"`
  - **Human-like completion:** `"This question always fascinated me!"`
- Step 2: fine-tune model to generate **helpful** text
  - **Query:** `"What is population of Berkeley?"`
  - **Helpful completion:** `"It is 117,145 as of 2021 census"`
- Use Reinforcement Learning in Step 2

Case Studies: Language Assistants

- MDP:
  - State: sequence of words seen so far (ex. `"What is population of Berkeley? "`)
    - $100{,}000^{1{,}000}$ possible states
    - Huge, but can be processed with feature vectors or neural networks
  - Action: next word (ex. `"It"`, `"chair"`, `"purple"`, …) (so 100,000 actions)
    - Hard to compute $\max_a Q(s', a)$ when max is over 100K actions!
  - Transition T: easy, just append action word to state words
    - s: `"My name"` a: `"is"` s': `"My name is"`
  - Reward R: ???
    - Humans rate model completions (ex. `"What is population of Berkeley? "`)
      - `"It is 117,145"`: **+1** `"It is 5"`: **-1** `"Destroy all humans"`: **-1**
    - Learn a reward model $\hat{R}$ and use that (model-based RL)
- Often use policy gradient (Proximal Policy Optimization) but looking into Q Learning

Conclusion

- We're done with parts I & II!

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

- Next up: Part III: Uncertainty and Learning!