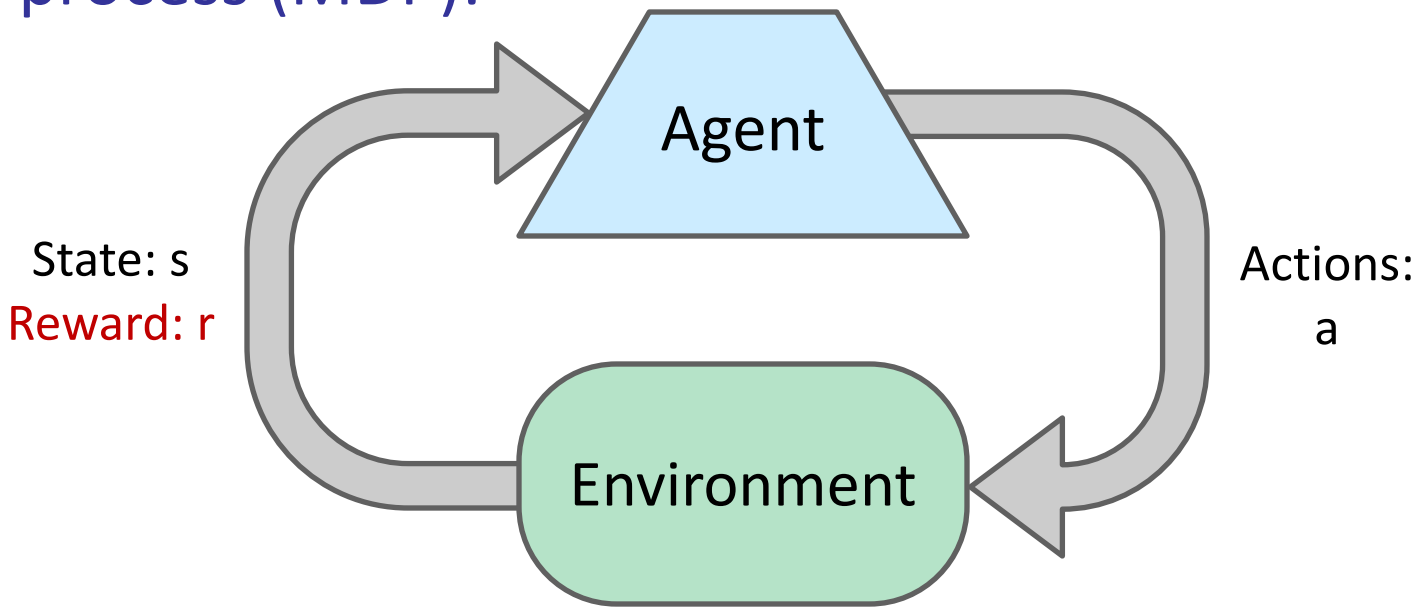
Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) $A(s)$
  - A transition 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 the consequence of actions and goodness of states
  - Must explore new states and actions
    - -- to bravely go where no robot has gone before

Approaches to reinforcement learning

1. Model-based learning

   Learn the model, solve it, execute the solution

2. Value-based methods

   Learn values from experiences, use them to make decisions

3. Policy-based methods

   Directly learn policies from experiences

Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. (available on Canvas)

Model-Free Methods: Mathematical Insight

Goal: Compute expected age of CS3317 students

Known P(A)

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

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

"Model Based": estimate P(A):

$$\hat{P}(A=a) = N_a/N$$

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

=

"Model Free": estimate expectation

$$E[A] \approx 1/N \sum_i a_i$$

Value Learning

- Value is an expectation over *returns*, estimate it using samples!
- Direct evaluation
  - $V^{\pi}(s)$ = E[total discounted rewards starting in s and following π]
  - *Sample = return =* total discounted rewards observed in one trial
  - Retrieve relevant samples from experience and take average (offline)
- Temporal difference (TD) learning
  - $V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s') \, [R(s,\pi(s),s') + \gamma V^{\pi}(s')]$ (recursive definition)
  - Sample = $R(s,\pi(s),s') + \gamma V^{\pi}(s')$ (Bootstrapped return)
  - Update $V^{\pi}(s)$ every time when observing a transition (s,a,s',r) (online)

TD Value Learning

- Model-free temporal difference learning
  - Experience world through episodes

$$(s, a, r, s', a', r', s'', a'', r'', s'''' \ldots)$$

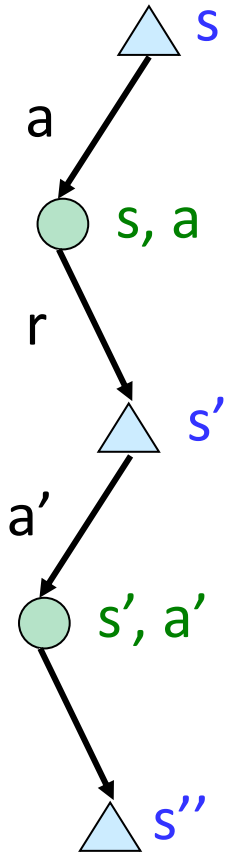
  - Update estimates after each transition (s, a, r, s')

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

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

  - Over time, updates will mimic Bellman updates

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

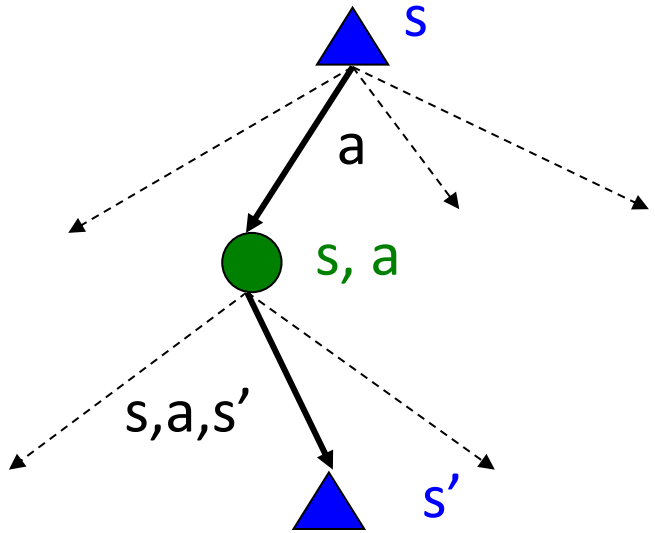


Problems with TD Value Learning

- We need *T* and *R* to turn values into a (new) policy!
- Recap: policy extraction operator:

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

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



- **Solution**: learn Q-values, not values
  - New algorithm coming up: *Q-learning*
  - It makes action selection model-free too!

Detour: Q-Value Iteration (Planning)

- Value iteration: compute (depth-limited) values recursively
  - Start with $V_0(s) = 0$ (always true)
  - Recursion: 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 convenient, so compute them instead
  - Start with $Q_0(s,a) = 0$ (always true)
  - Recursion: 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 as approximate Q-iteration

- Q-value iteration:

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

- Approximate the expectation using samples and running average:
  - $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a')]$

- We obtain a policy from learned Q(s,a), with no model!
  - $\pi^*(s) = \text{argmax}_a Q^*(s,a)$
  - (No free lunch: Q(s,a) table is |A| times bigger than V(s) table)

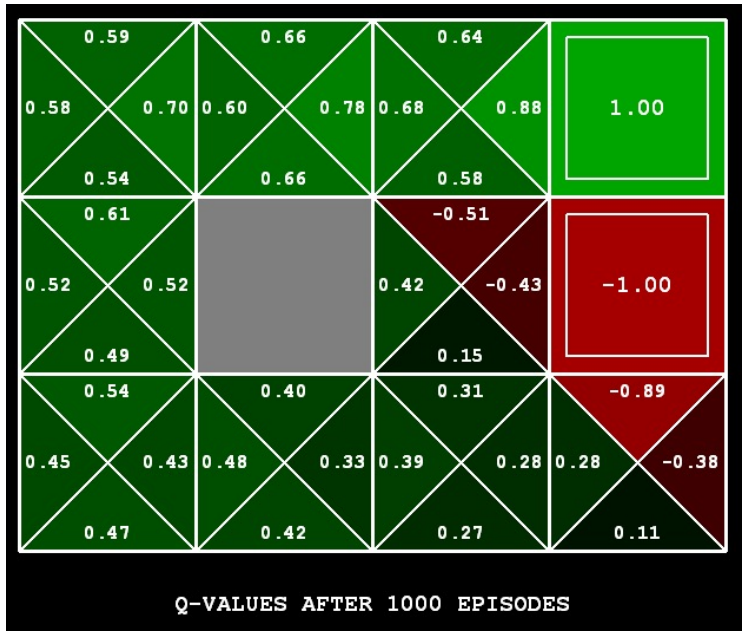
Q-Learning Algorithm

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

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

  - Incorporate the new sample into the running average:

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



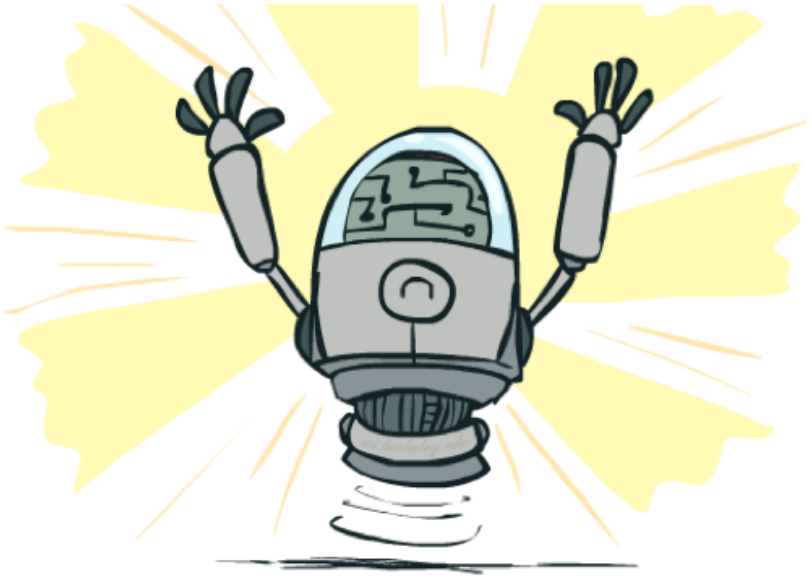
[Demo: Q-learning – gridworld (L10D2)]
[Demo: Q-learning – crawler (L10D3)]

Video of Demo Q-Learning -- Gridworld

Video of Demo Q-Learning -- Crawler

Q-Learning Properties

- Theoretical guarantee: Q-learning converges to optimal policy -- even if samples are generated from a suboptimal policy!
- This is called ***off-policy learning***
- Caveats:
  - You have to explore sufficiently
    - Eventually try every state/action pair infinitely often
  - You have to decrease the learning rate appropriately
    - Requirements: $\sum_t \alpha(t) = \infty$, $\sum_t \alpha^2(t) < \infty$
    - Satisfied by: $\alpha(t) = 1/t$ or (better) $\alpha(t) = K/(K+t)$



Summary

- RL solves MDPs via direct experience of transitions and rewards
- There are several approaches:
  - Learn the MDP model and solve it
  - Learn V directly from sums of rewards, or by TD updates
    - Still need a model to make a policy
  - Learn Q by local Q updates
    - Can directly select actions

The Story So Far: MDPs and RL

Known MDP: Planning

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration / MCTS
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

Unknown MDP: Model-Free

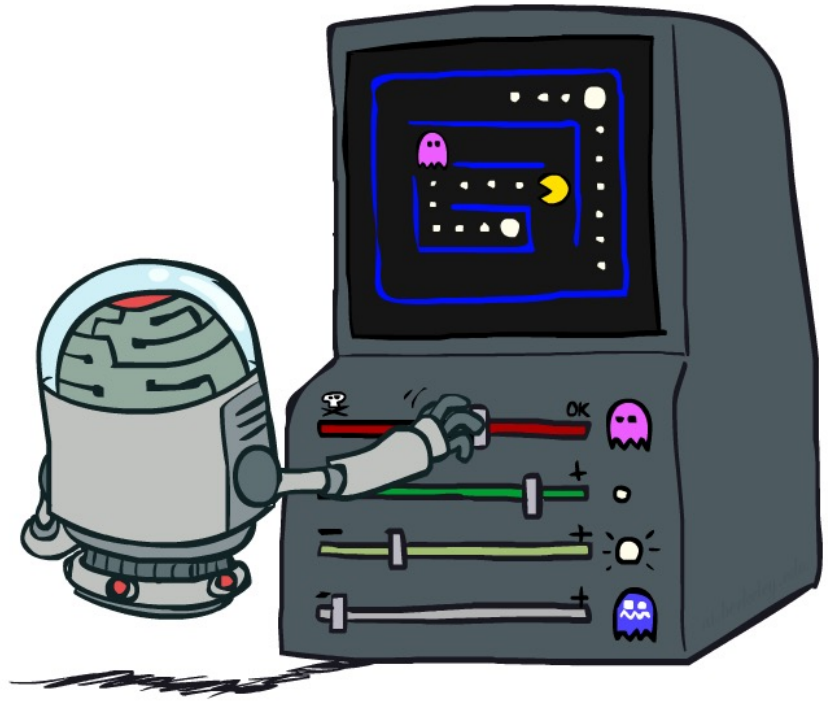
Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	TD Learning

Big Missing Pieces

- How to explore without too much regret?
- How to scale this up to large state and action spaces?
  - Tetris ($|S|=10^{60}$), Go ($|S|=10^{172}$), StarCraft ($|A|=10^{26}$)?

CS 3317: Artificial Intelligence

Reinforcement Learning II



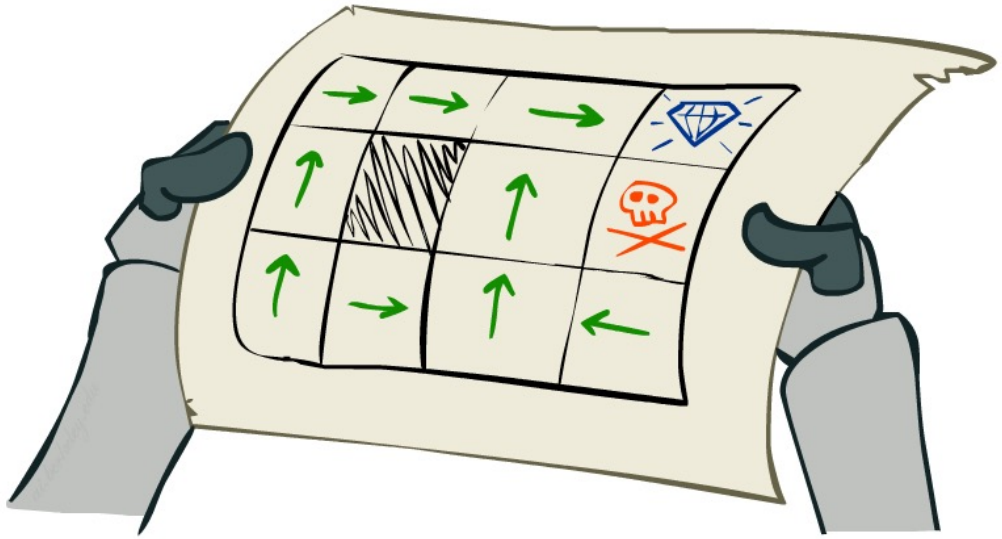
Instructors: **Cai Panpan**

Shanghai Jiao Tong University

(slides adapted from UC Berkeley CS188)

Till Now: Passive Reinforcement Learning

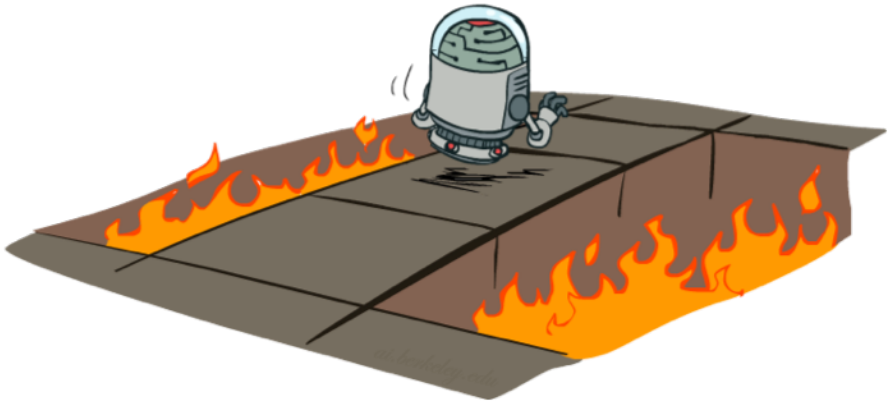
- Goal: learn policy values / optimal values
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - Input: a fixed policy $\pi(s)$



- In this case:
  - Learner is passively observing
  - No choice about what actions to take
  - Just execute a fixed policy and learn from its experience

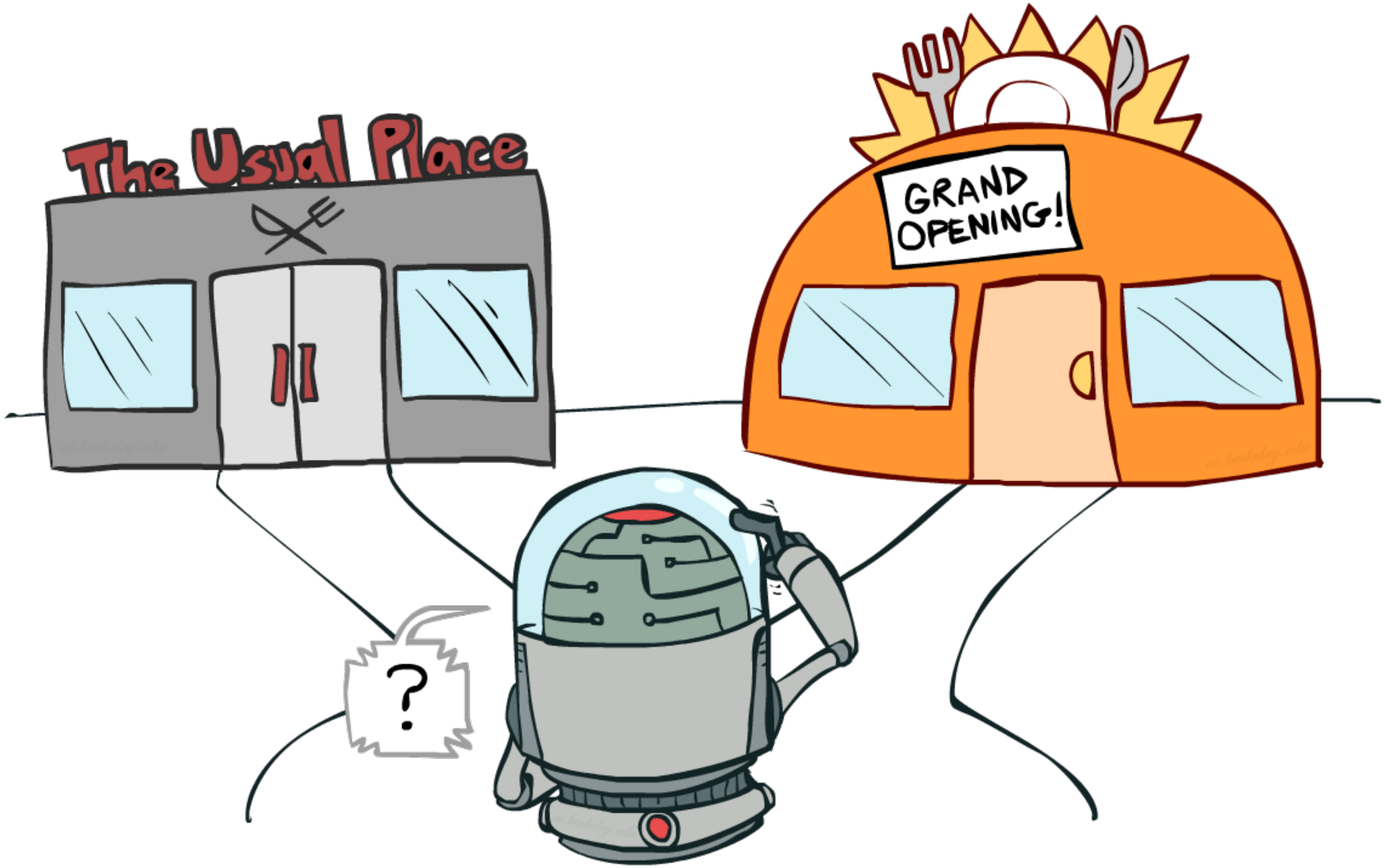
Next: Active Reinforcement Learning

- Goal: learn optimal policies / values
  - 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*



- In this case:
  - Learner actively choose actions!
  - Learn from our own experience
    - Can turn Q-learning into active RL if using the learned policy to act
  - Fundamental tradeoff: *exploration* vs. *exploitation*

Exploration vs. Exploitation



Exploration vs exploitation

- ***Exploration***: try new things (states and actions)
  - Collect information to help later decisions
- ***Exploitation***: do what's best given what you've learned so far
  - Collect rewards by leveraging the current information

- Insight:
  - Exploitation is often appealing: we human like immediate rewards
  - But pure exploitation often leads to ***local-optima***!
  - To be optimal, need to combine exploration and exploitation
  - ***When and how to perform exploration?***

Exploration method 1: ε-greedy

- ε-greedy exploration
  - Every time step, flip a biased coin
  - With (small) probability ε, act *randomly*
  - With (large) probability 1-ε, act on current policy

- Properties of ε-greedy exploration
  - Every s,a pair is tried infinitely often
  - Does a lot of stupid things
    - Jumping off a cliff *lots of times*
  - Keeps doing stupid things for ever
    - **Solution**: decay ε towards 0

Demo Q-learning – Epsilon-Greedy – Crawler

Better Idea? Bandits

A Tries: 1000
Winnings: 900

B Tries: 100
Winnings: 90

C Tries: 5
Winnings: 4

D Tries: 100
Winnings: 0

- Which arm to try next?
- Most people would choose C > B > A > D
- Basic intuition: higher mean is better; more uncertainty is better
- Multi-armed bandit problem
  - UCB is a good solution!

Exploration Functions

- Exploration functions implement this tradeoff
  - Takes a value estimate u and a visit count n, and returns an optimistic utility:
  - e.g., $f(u,n) = u$ (*value*) $+ k/\sqrt{n}$ (*exploration bonus*)

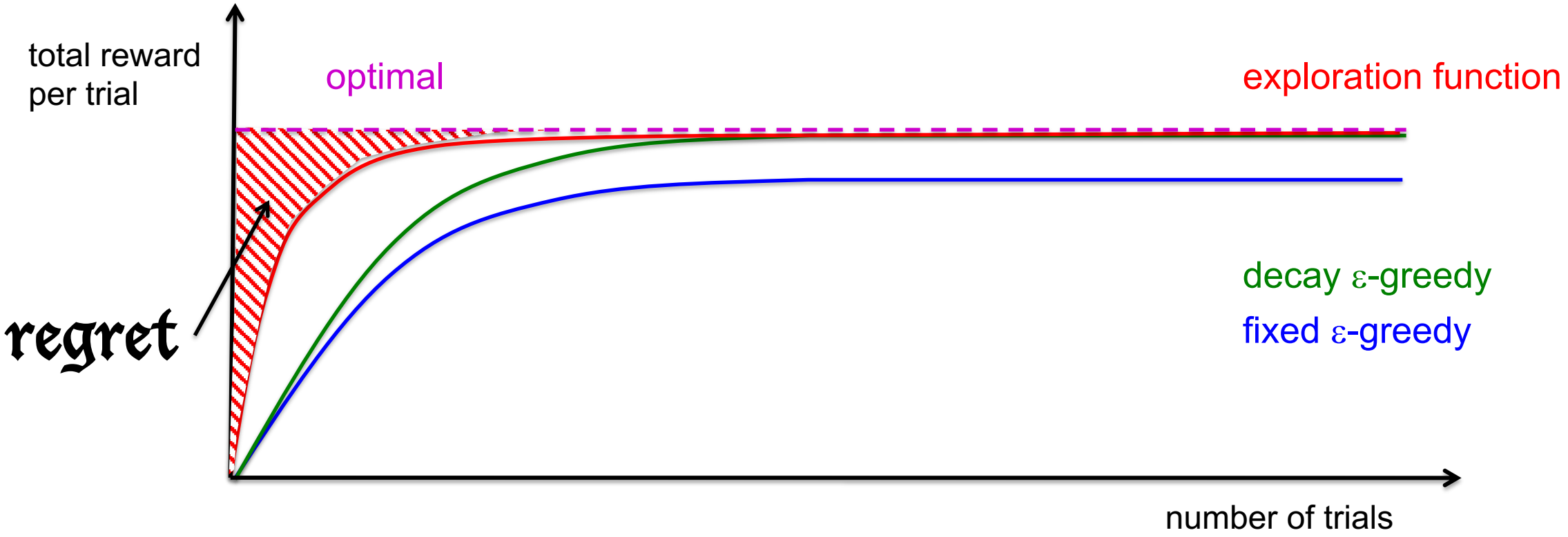
- Q-learning + exploration function
  - Regular Q-update:

    $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_a Q(s',a)]$
  - Modified Q-update:

    $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_a f(Q(s',a),n(s',a'))]$
  - Note: also propagated the "bonus" back to states that lead to unknown states
    - e.g., s (known state) ->s' (unknown state)

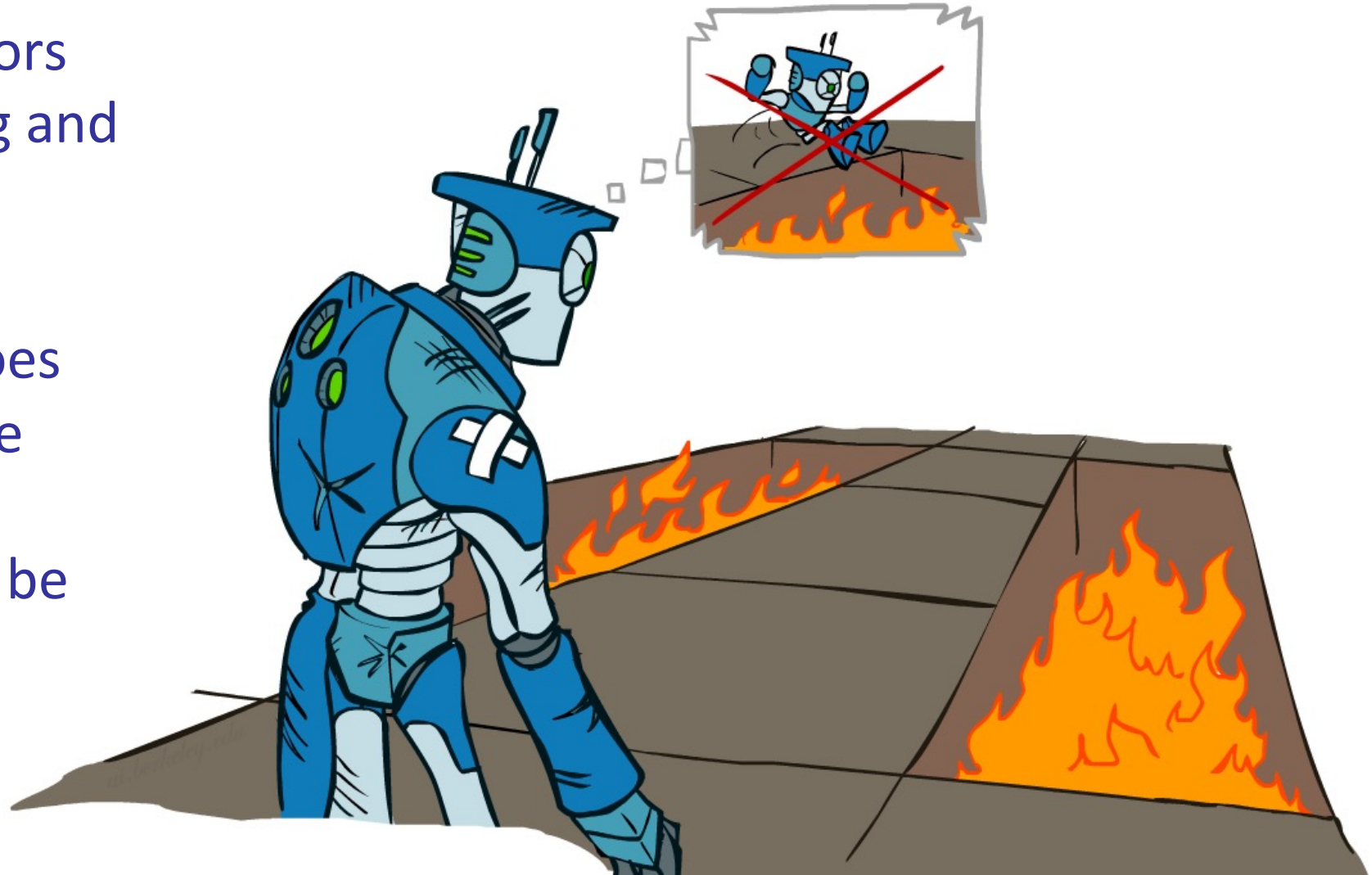
Demo Q-learning – Exploration Function – Crawler

Exploration and regret



Regret

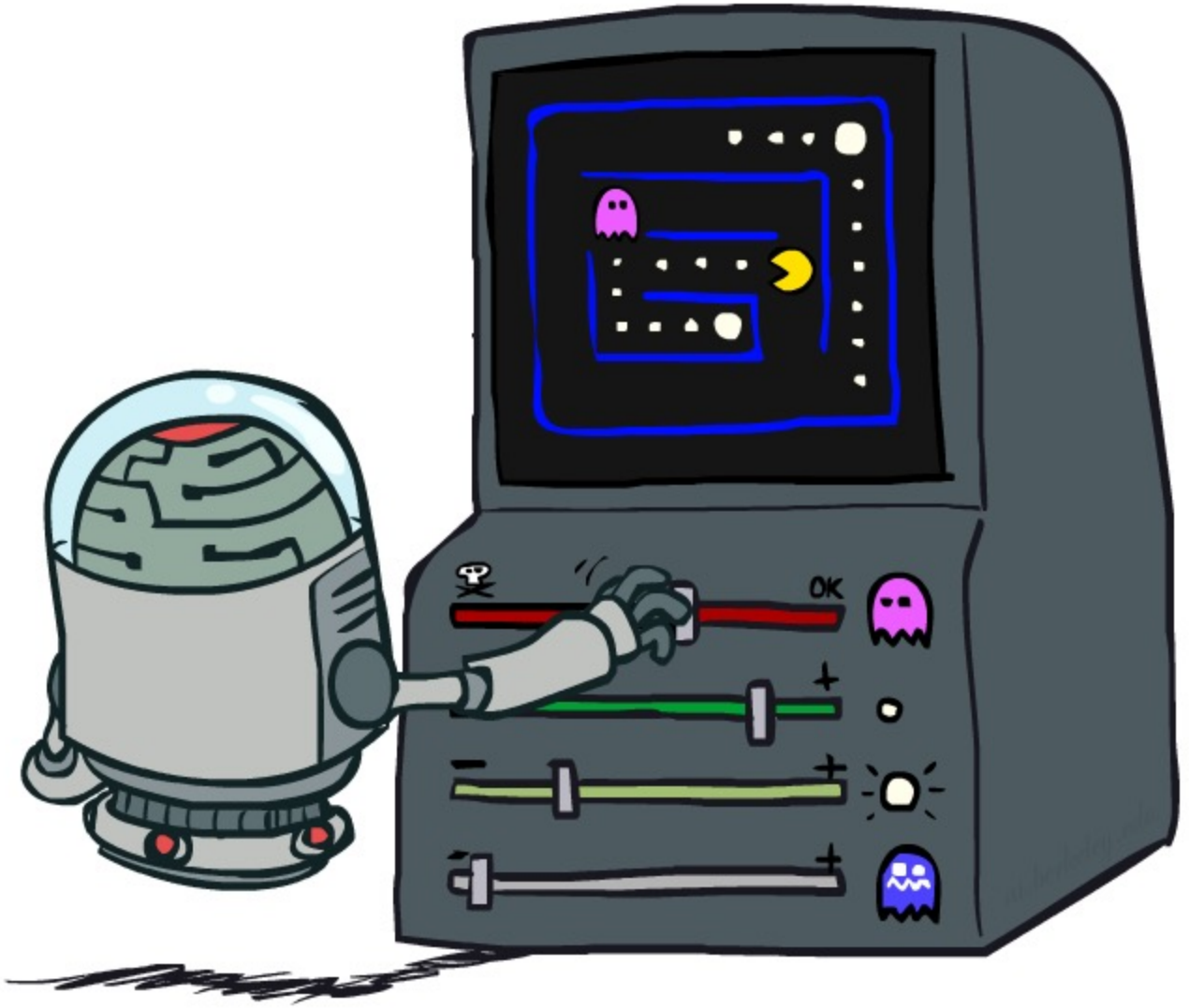
- ***Regret*** measures the total cost of inevitable errors made while exploring and learning instead of behaving optimally
- ***Minimizing regret*** goes beyond learning to be optimal – it requires *optimally* learning to be optimal



Big Missing Pieces

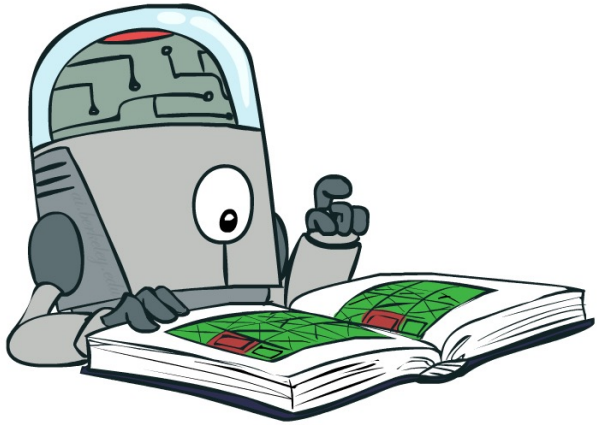
- (Done) How to explore without too much regret?
- How to scale this up to large state and action spaces?
  - Tetris ($|S|=10^{60}$), Go ($|S|=10^{172}$), StarCraft ($|A|=10^{26}$)?

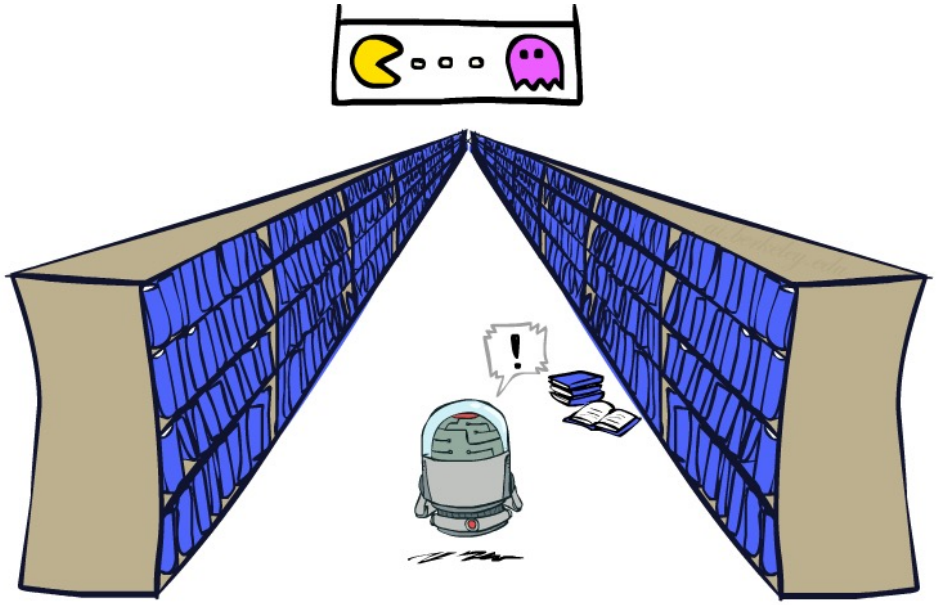
Approximate Q-Learning



Generalizing Across States

- Basic Q-Learning keeps a *table* of all Q-values
- In realistic situations, we cannot possibly learn for every single state!
  - Too many states to visit in training
  - Too many states to hold the Q-tables
- Instead, we want to *generalize*:
  - Learn for some small number of training states
  - Generalize that experience to new, similar situations
  - *Can we apply some machine learning tools to do this?*





[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!

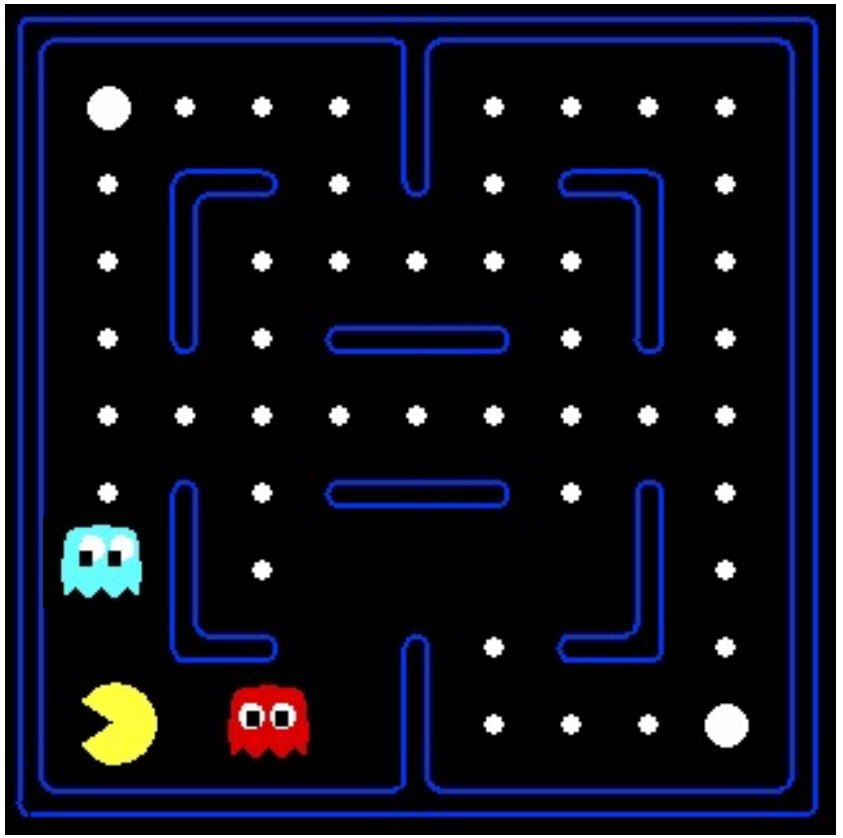
Demo Q-Learning Pacman – Tiny – Watch All

Demo Q-Learning Pacman – Tiny – Silent Train

Demo Q-Learning Pacman – Tricky – Watch All

Feature-Based Representations

- Describe a state using a vector of ***features***
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost f_{GST}
    - Distance to closest dot
    - Number of ghosts
    - 1 / (distance to closest dot) f_{DOT}
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
  - Can also describe a q-state (s, a) with features
    - e.g., action moves closer to food, f_{DOT} gets higher



Linear Value Functions

- We can express V and Q (approximately) as weighted linear functions of feature values:
  - $V_{\mathbf{w}}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
  - $Q_{\mathbf{w}}(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$

- Advantage: our experience is summarized in a few powerful numbers
  - Generalize over states with similar features
- Disadvantage:
  - With wrong features, the best possible approximation can be terrible
  - With wrong features, states with same features can have very different values
  - But in practice we can compress a value function for chess (10^{43} states) down to about 30 weights and get decent play!!!

Updating a linear value function

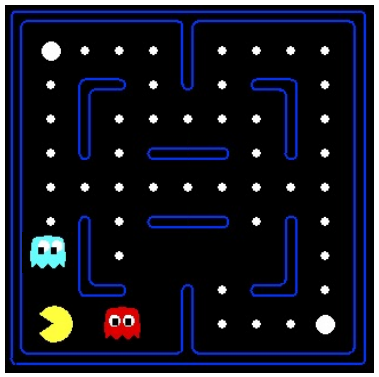
- Original Q-learning directly updates *Q's* to reduce the error at s,a:
  - $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
- Instead, we update the *weights* to reduce the error at s,a:
  - $w_i \leftarrow w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)] \partial Q_{\mathbf{w}}(s,a)/\partial w_i$

    $= w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)] f_i(s,a)$
- Intuitively:
  - Pleasant surprise: *sample > old Q*
    - Increase weights on positive features, decrease on negative ones
    - *(Reinforce contributing features, weaken inconsistent features)*
  - Unpleasant surprise: *sample < old Q*
    - Decrease weights on positive features, increase on negative ones
    - *(Blame contributing features, reinforce counter features)*

Example: Q-Pacman

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

s

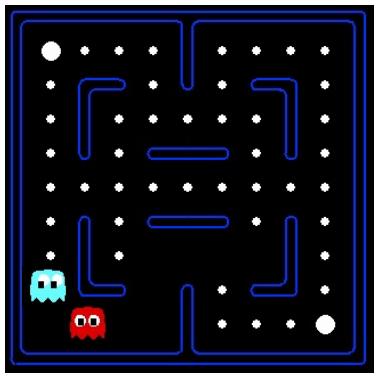


$f_{DOT}(s,\text{NORTH}) = 0.5$

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

a = NORTH
r = –500

s’



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

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

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

difference = –501

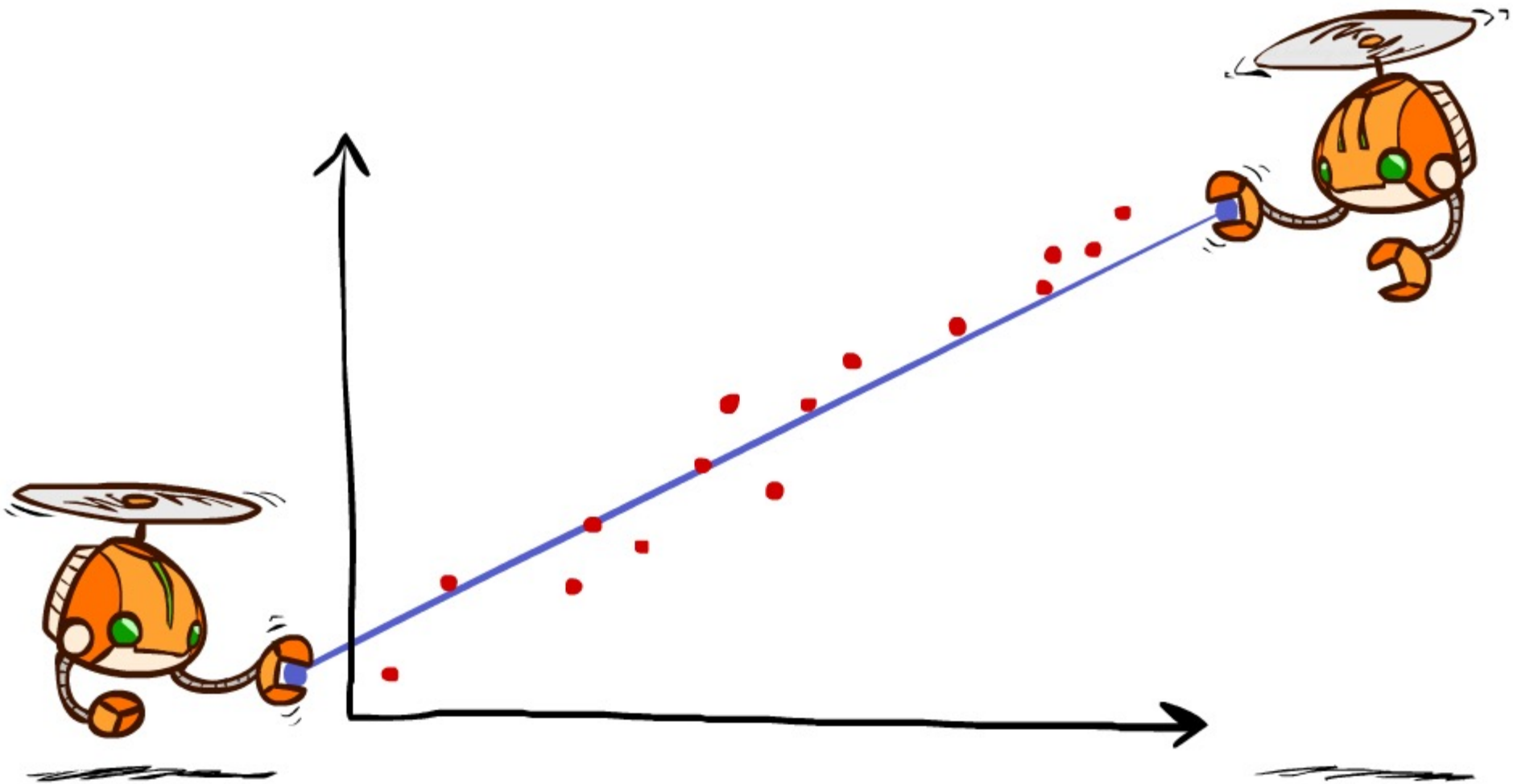
$w_{DOT} \leftarrow 4.0 + \alpha[-501]0.5$

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

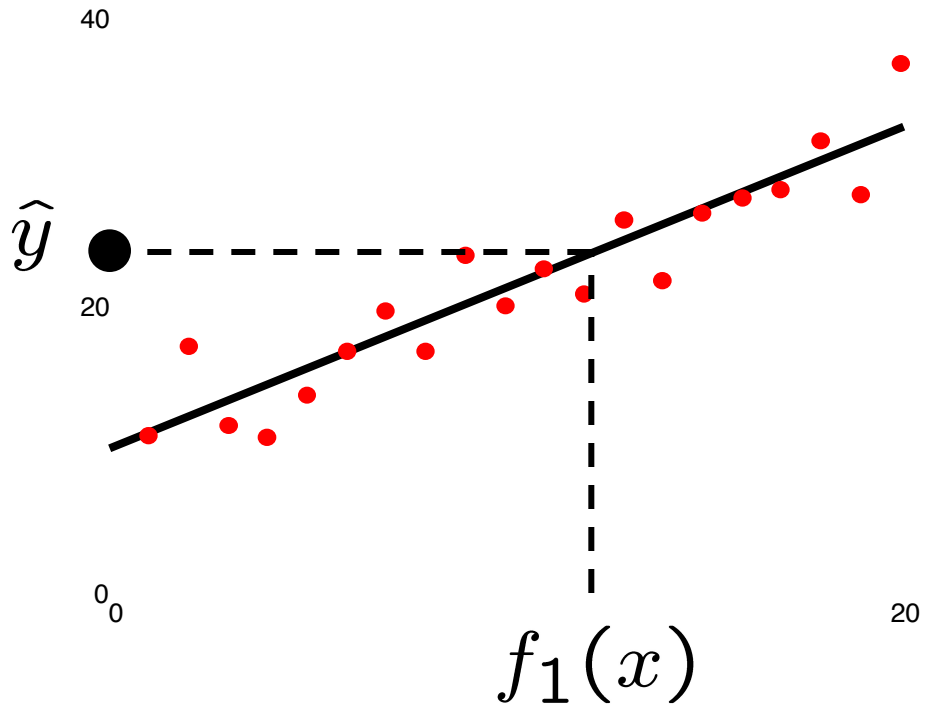
$$Q(s,a) = 3.0\, f_{DOT}(s,a) - 3.0\, f_{GST}(s,a)$$

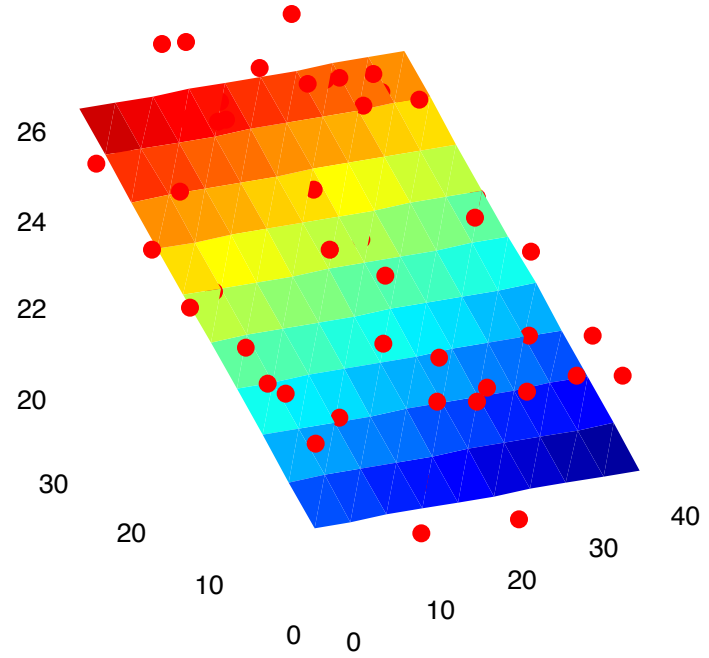
Demo Approximate Q-Learning -- Pacman

Q-Learning and Least Squares



Linear Regression*





Model:

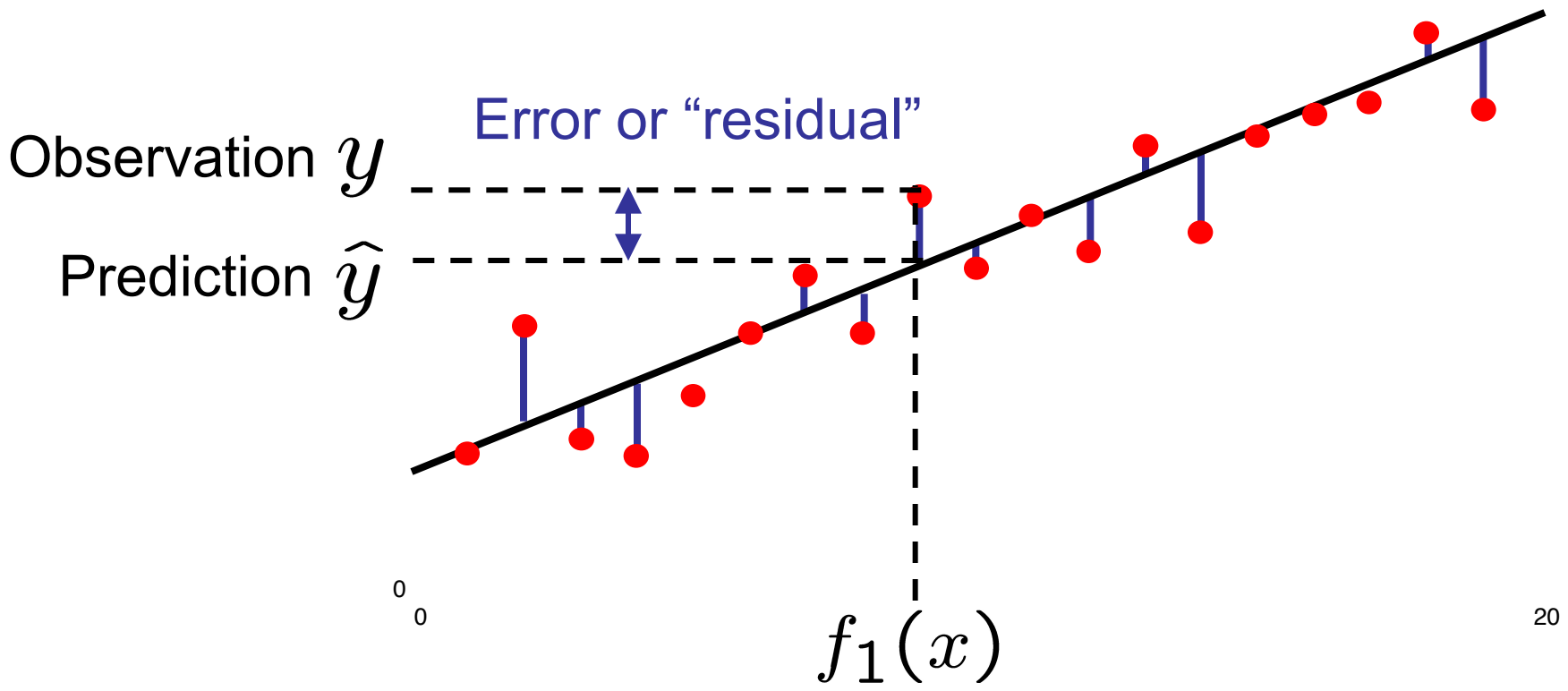
$$\hat{y} = w_0 + w_1 f_1(x)$$

Model:

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

Optimization: Least Squares*

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



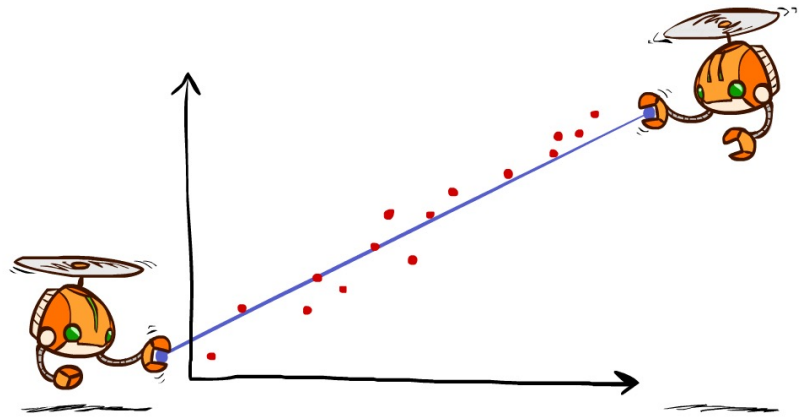
Minimizing Error*

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 is very similar:

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

"target" "prediction"

Convergence*

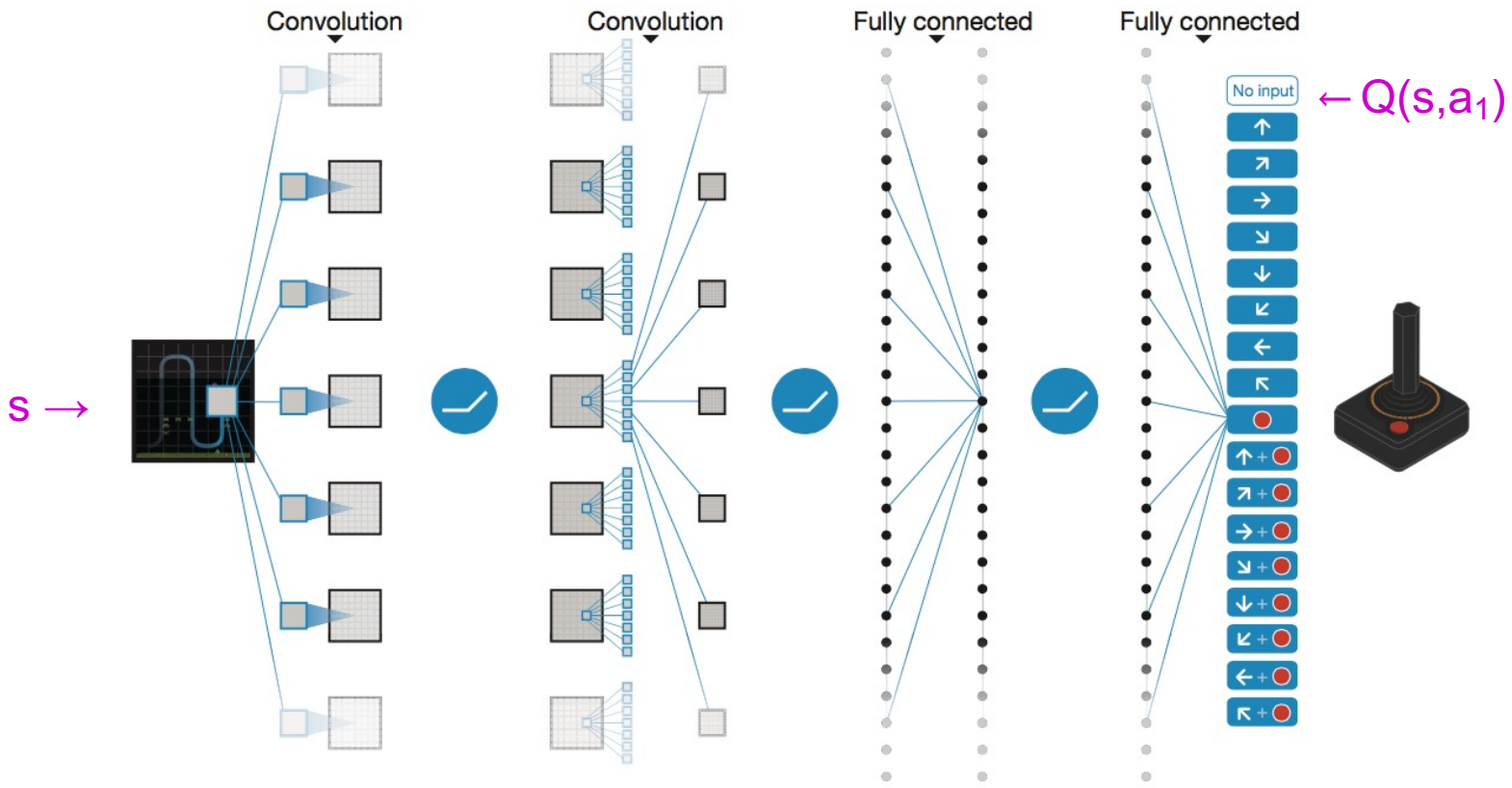
- Let V^L be the closest linear approximation to V^*.
- TD learning with a linear function approximator converges to some V that is pretty close to V^L
- Q-learning with a linear function approximator may diverge
- With more sophisticated update rules, stronger convergence results can be achieved
  - – even for *nonlinear function approximators*

Nonlinear function approximators

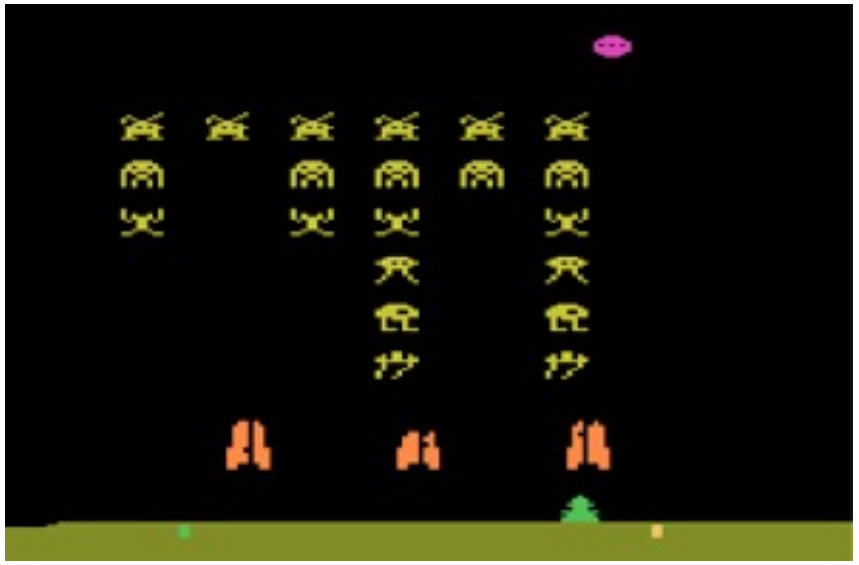
- The gradient-based update can be applied to ***any*** Q_w:
  - $w_i \leftarrow w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\ ] \partial Q_w(s,a)/\partial w_i$
- Neural networks?
  - Back-propagation computes the gradient!
- **Hypothesis**: maybe we can get much better V or Q approximators using deep neural nets instead of linear functions

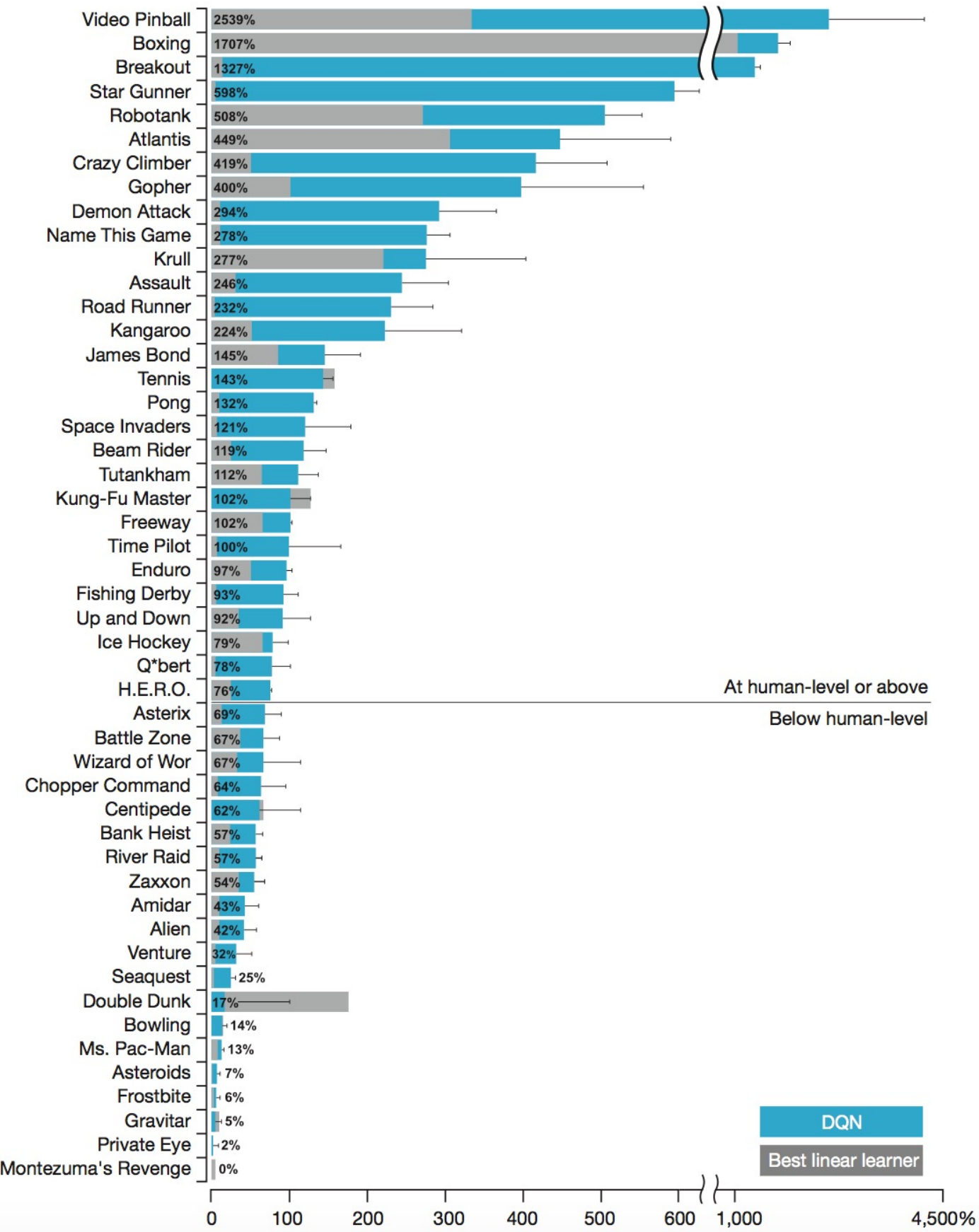
DeepMind DQN

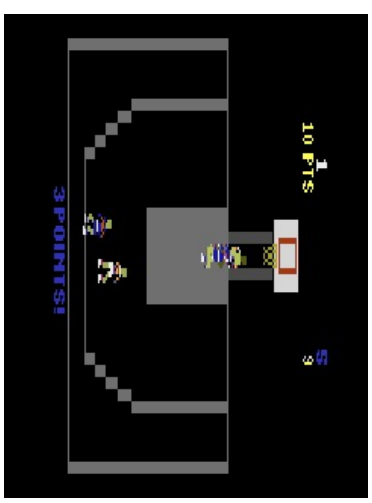
- Used a deep neural network to represent Q:
  - Input: last 4 screen images (84x84 pixel values) + score
  - Output: Q values



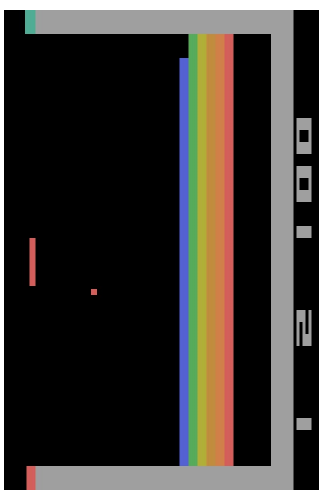
Play 49 Atari games with DQN







Double dunk



Video Pinball
Boxing
Breakout

Summary

- Exploration vs. exploitation
  - Exploration guided by unfamiliarity and potential
  - Appropriately designed bonuses tend to minimize regret
- Generalization allows RL to scale up to real problems
  - Represent V or Q with parameterized functions
  - Adjust parameters to reduce prediction error of samples