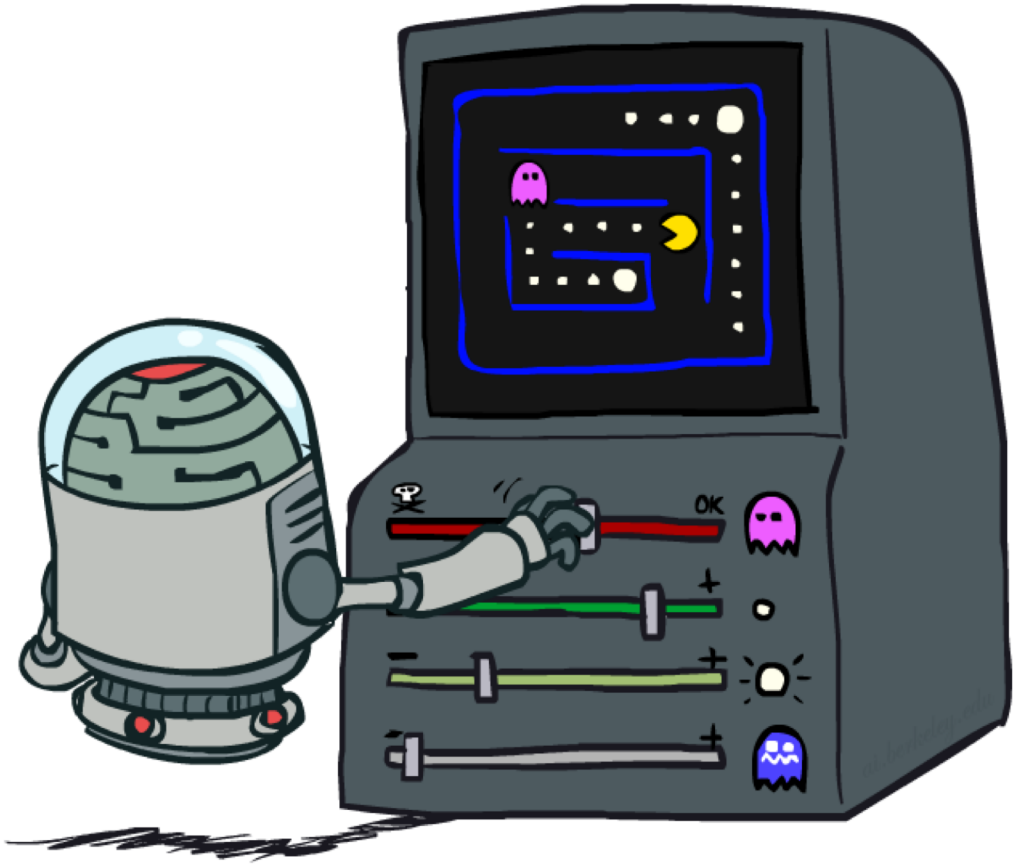
Announcements

- Project 2 Mini-Contest (Optional)
  - Ends Sunday 9/30
- Homework 5
  - Released, due Monday 10/1 at 11:59pm.
- Project 3: RL
  - Released, due Friday 10/5 at 4:00pm.

CS 188: Artificial Intelligence

Reinforcement Learning II



Instructors: Pieter Abbeel & Dan Klein --- University of California, Berkeley

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 $\pi(s)$

  we want to find the policy that maximizes the expected discounted future rewards

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

you could do this in two ways:
1) just do value iteration, but instead of having to take the max over actions, you just take the action prescribed by the policy
2) without the max, it's actually a linear set of equations, so you could just solve it with a linear-solver (i.e. Matlab)

Unknown MDP: Model-Based

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

- let's collect data and from that data, estimate the model (i.e. T & R). Then we can use the techniques we already know for known MDPs

Unknown MDP: Model-Free

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

even better because it tells you which actions to take in any given state

without learning the model, directly learn V*, Q*, pi

Two approaches:
1) Direct Eval: calculate avg future rewards after being in a state
2) TD/Indirect eval: Use sampled version of Bellman's equation to update our estimated V(s) of the state we were just in

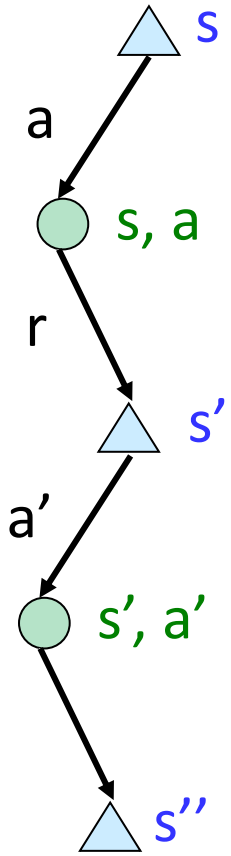
Model-Free Learning

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

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

  - Update estimates each transition (s, a, r, s')
  - Over time, updates will mimic Bellman updates

using a sample-based way instead of having to know the model



Q-Learning

start with Qk = 0

- 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

this gives us one term in the sum (in the above equation). In the above equation, we consider all possible transitions (s, a, r, s'). This is because the transition function suggests that there are many possible s'. But here in the sample, we only visit one of the many possible s'.

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

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

we'll slightly adjust the original estimate with the result of the sample

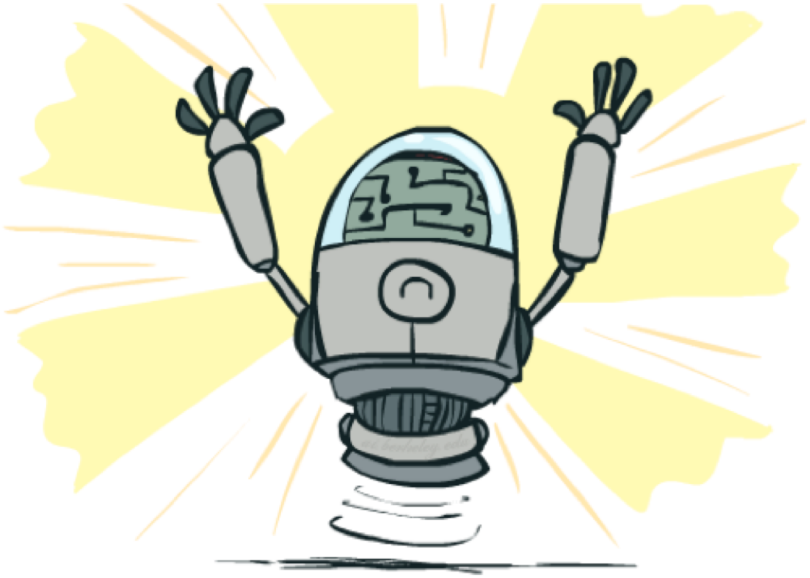
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha) \left[r + \gamma \max_{a'} Q(s',a') \right]$$

Q-Learning Properties

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

this is due to the "max" in the equation. The update happens based on the max of the q-values, not the action that you actually took.

- This is called off-policy learning



- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough

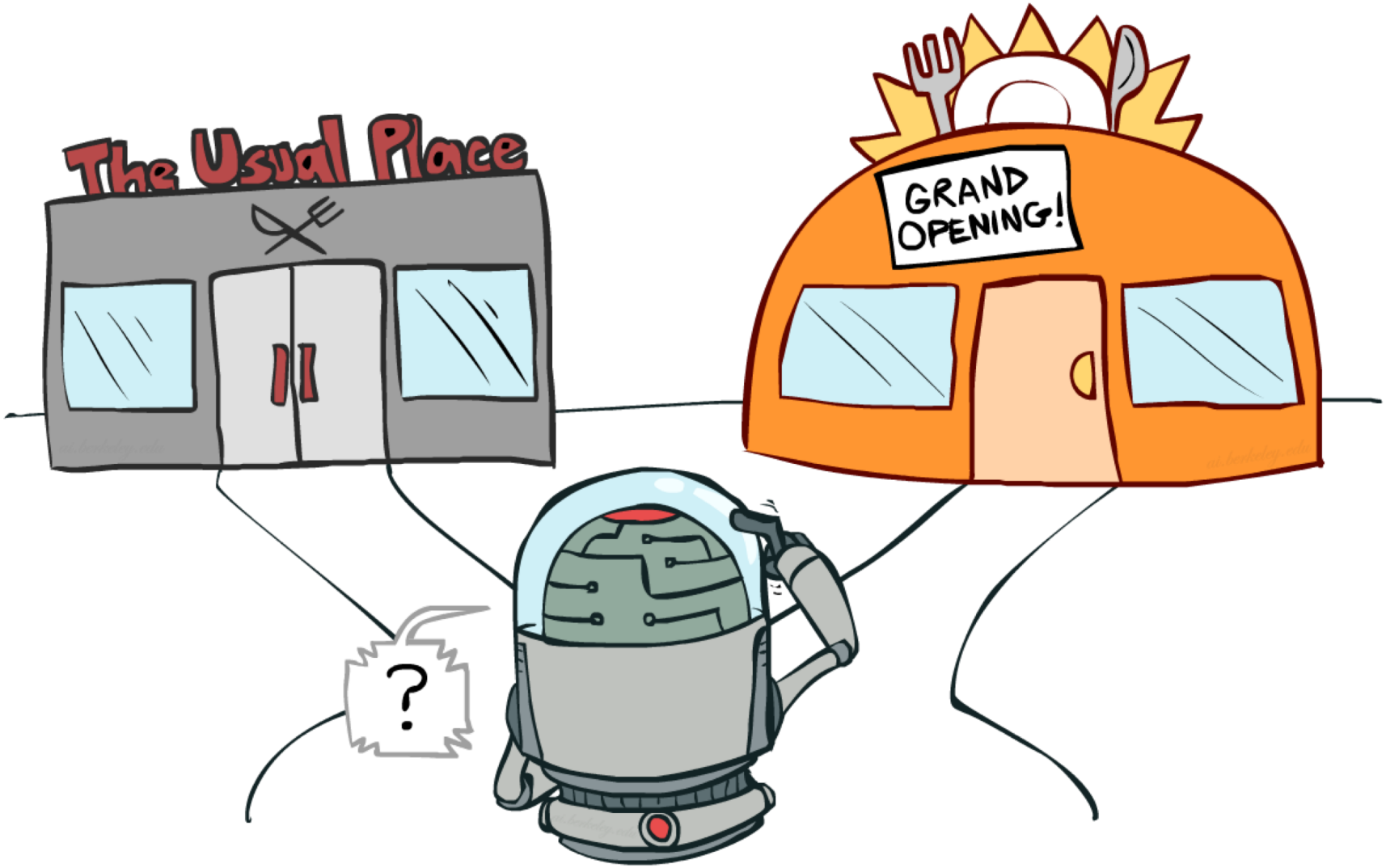
need to gradually decay your alpha, but not too quickly

  - … 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

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 – 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

explore areas were we haven't explored much yet

- Exploration function
  - Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g. $f(u, n) = u + k/n^{+1}$

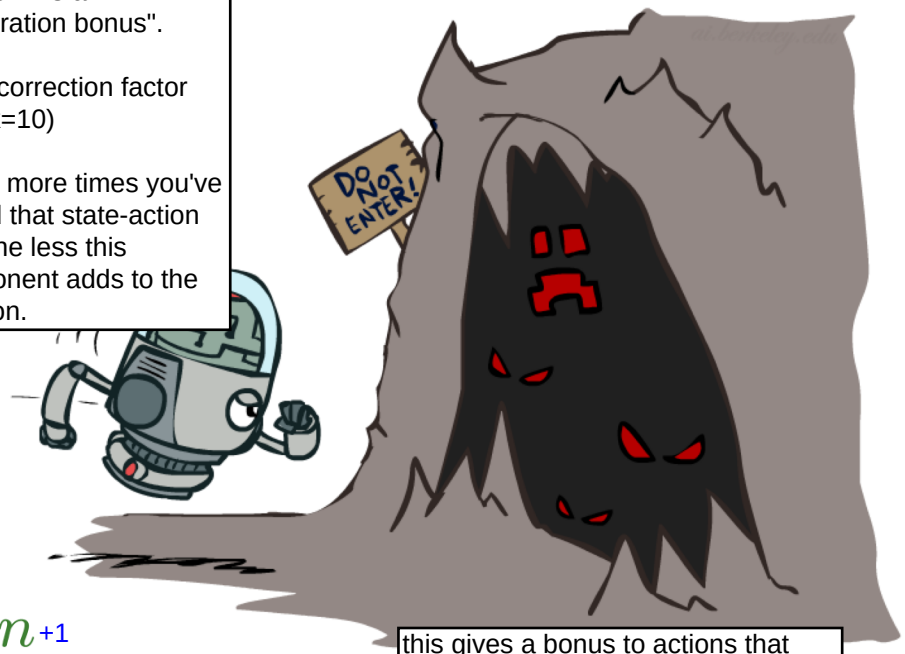
takes in an estimate of the utility and the count

so for each state-action pair, we keep of its utility and count

This term is an "exploration bonus".

k is a correction factor (e.g. k=10)

so the more times you've visited that state-action pair, the less this component adds to the function.



this gives a bonus to actions that haven't been taken often in that state. So we favor these actions b/c they get artificially higher values. So it makes these actions more likely to occur.

this just means it's a weighted update. It's short hand for the (1-alpha) thing we did on previous slides

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)]

Video of Demo Q-learning – Exploration Function – Crawler

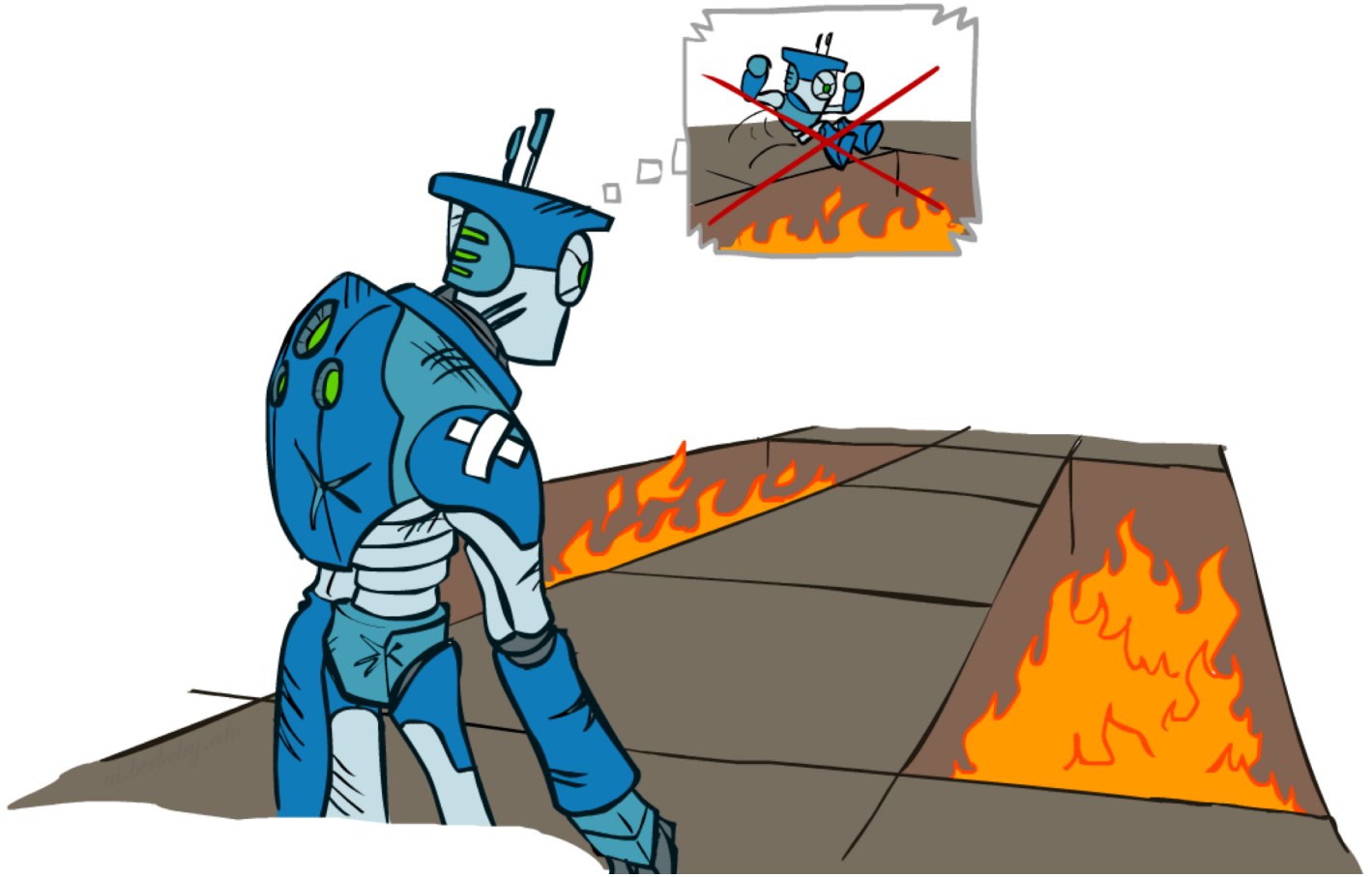
you can never have 0 regret because you always have to explore at the start!

Regret

we can compare algorithms based on if they get to an optimal policy. But we can also compare them on how fast they get to an optimal policy! We know epsilon-greedy q-learning and exploration-function q-learning get to an optimal policy. But the exploration-function q-learning does it quicker. You could measure this by summing the rewards over the entire learning process, instead of just seeing the rewards of the final policy.

- 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

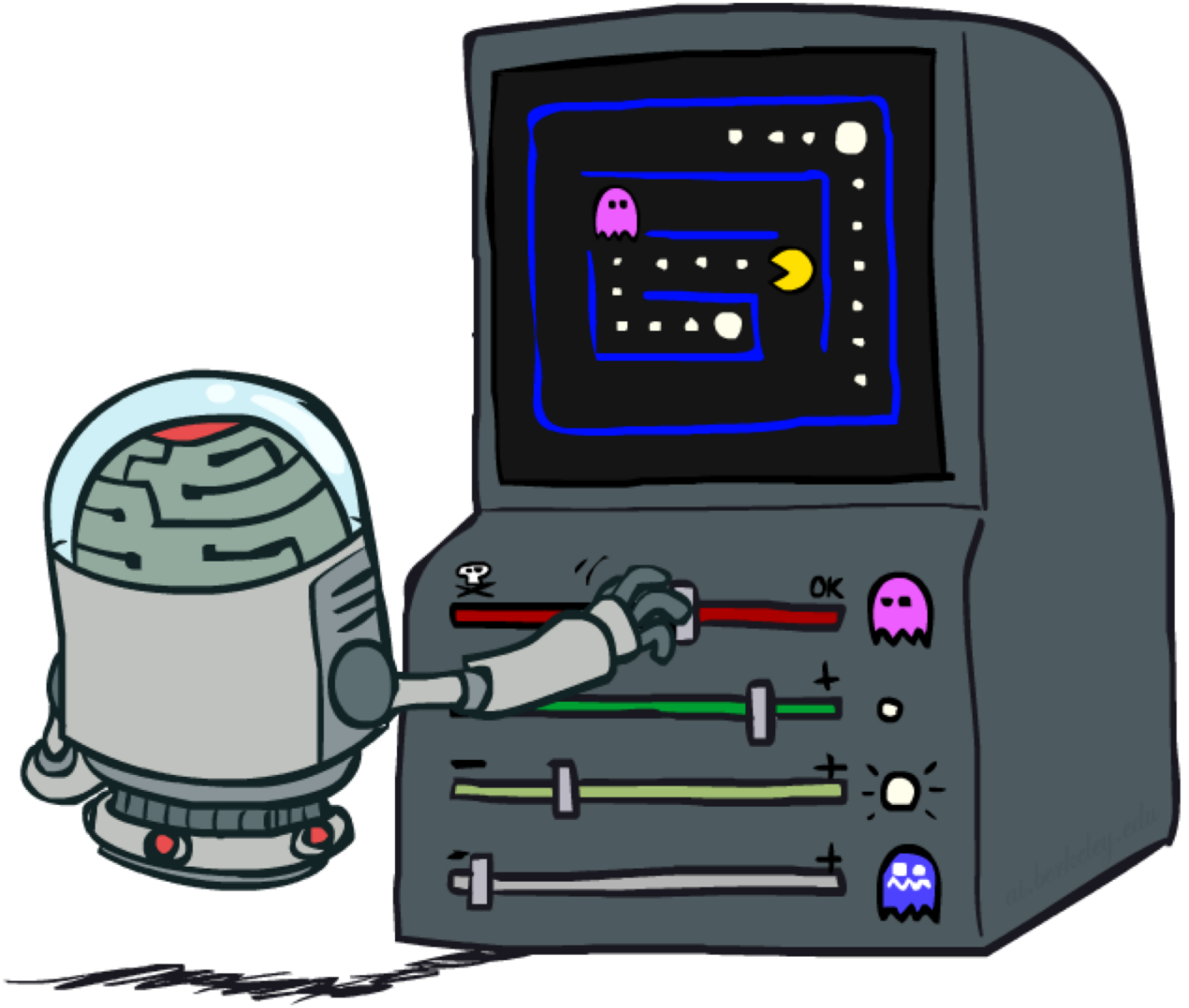
the total accumulated rewards during the training process in random exploration is lower than the total accumulated rewards during the training process using an exploration function



Approximate Q-Learning

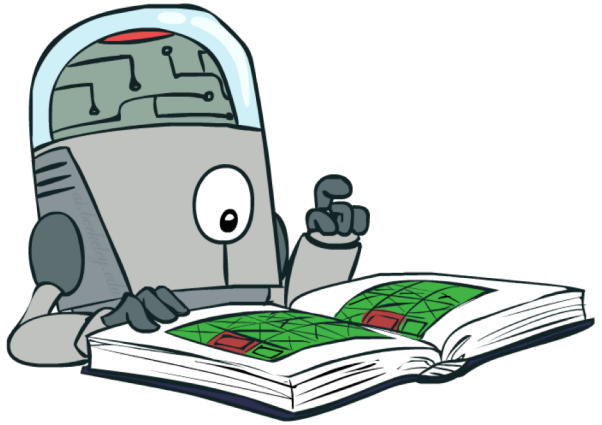
So far we've looked at exact methods. If you run them long enough, you'll get the exact answer.

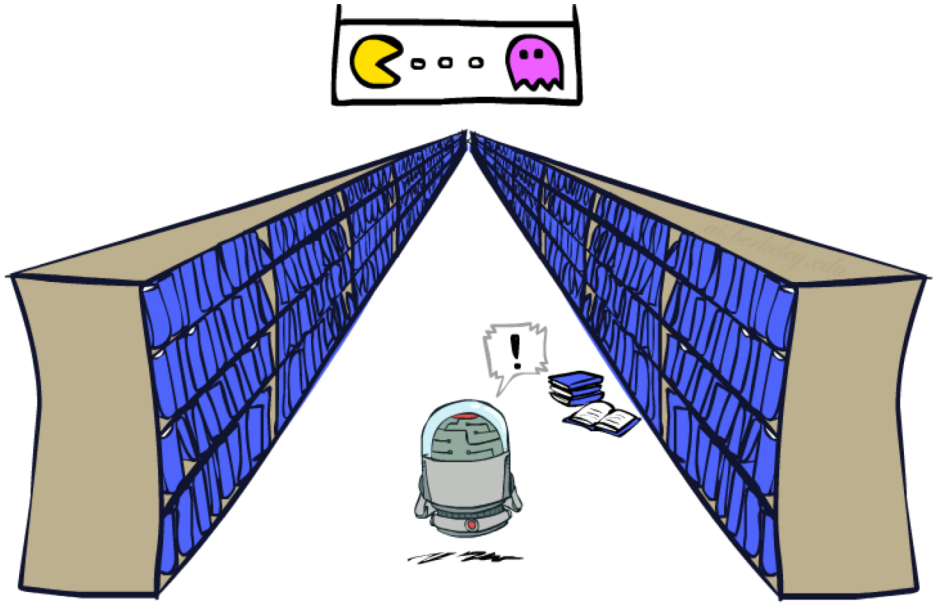
Now we'll look at approximating this solution.



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

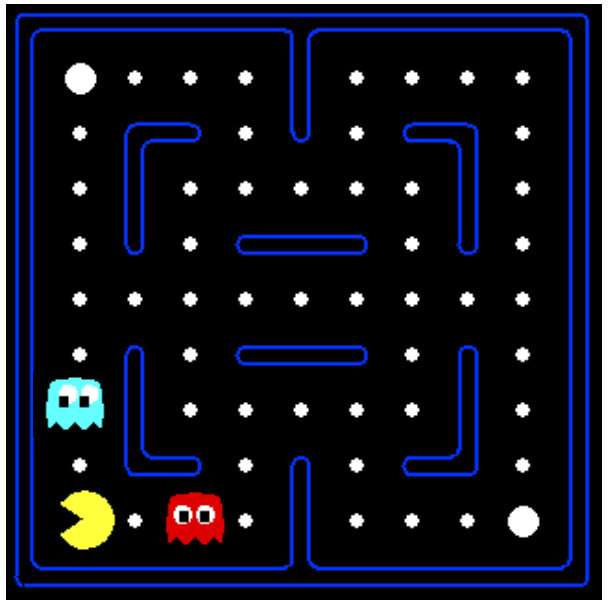




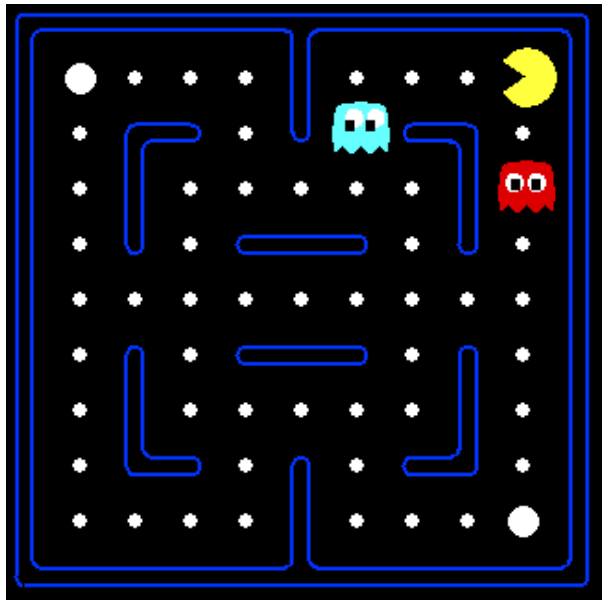
[demo – RL pacman]

Example: Pacman

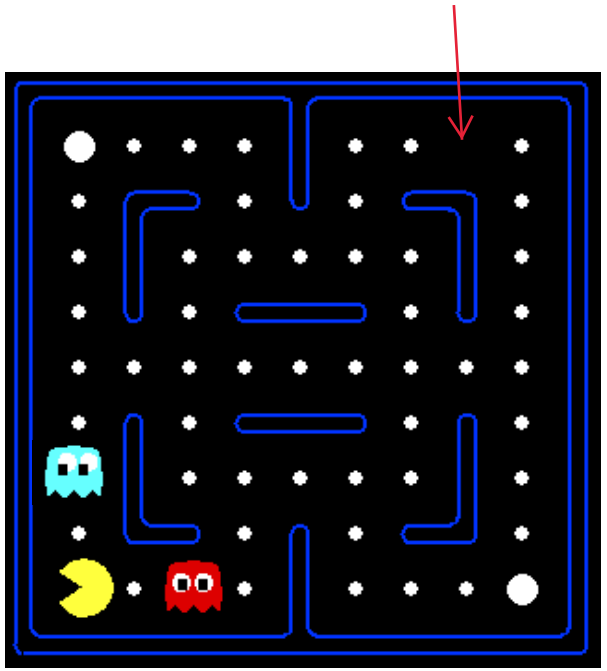
Let's say we discover through experience that this state is bad:



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)]

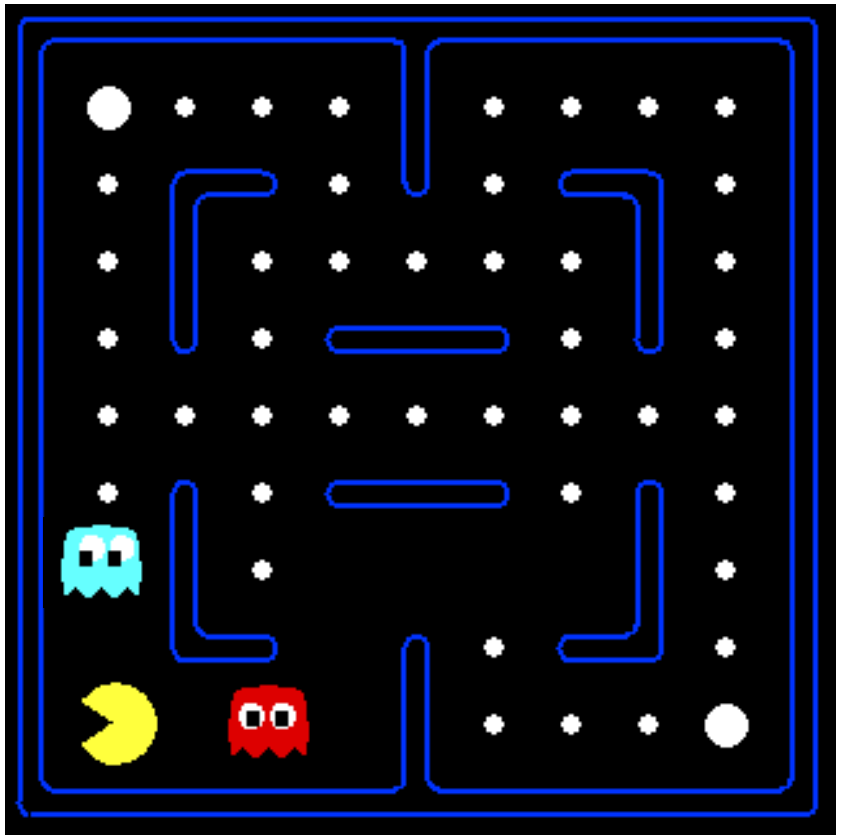
Video of Demo Q-Learning Pacman – Tiny – Watch All

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 / (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)



we could also use non-linear function approximators...but in the course, they hadn't gotten to neural networks or ML yet

Linear Value Functions

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

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

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

learn these weights

- 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

the weight for feature i is being updated. You update it by taking the original value of w_i and adding the product of 1) a learning rate 2) the Q-value difference between our current sample and our approximation so far, 3) the feature value

the correction is scaled by how big/ active the feature was. Large features dominate how we think about that state. Small, near zero features don't really impact our thinking about that state.

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

- Q-learning with linear Q-functions:

current sample

approximation so far

nudge the current Q-value based on the sample we just saw

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

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

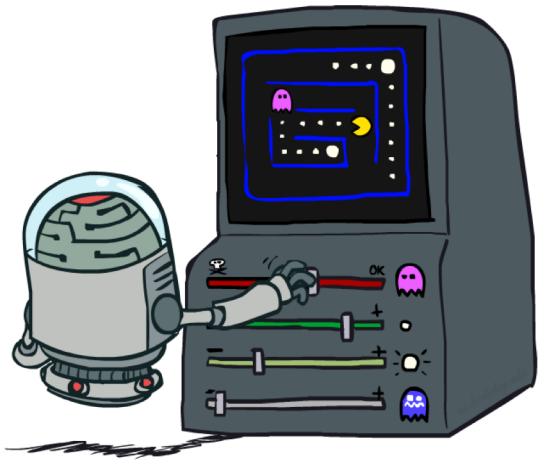
$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}]$$

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

Exact Q's

Approximate Q's

now we want to update the weights, not the table of q-values!



- 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

Let's say that feature i can be either 1 or 0. If it's 0, that means the feature wasn't present in the sample and its weight doesn't change. Let's assume that feature i is 1. Let's also assume the difference is positive, meaning that the q-value we got from our current sample is higher than our approximation thus far. Using the formula, this means that we would increase the value of weight i, which would increase the q-value of s,a.
If the difference is lower, then the weight will be nudged down, meaning that the q-value will now be lower.

Let's instead assume feature i is continuous (let's say it can go from 0 to 10). If the difference is positive and f_i = 10 then the weight will increase a lot. If the difference is positive and f_i = 1, then the weight will increase by a little. This makes sense. When the weight is super active (i.e. 10) and our sample had a higher q-value than what we expected, it makes sense to increase the weight of that feature a lot.

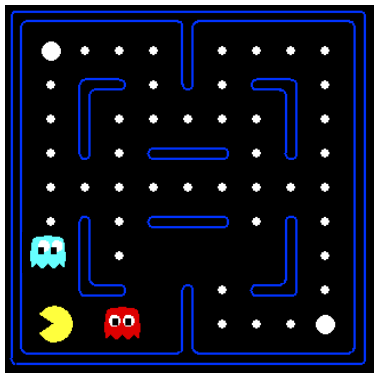
- Formal justification: online least squares

Two features: 1) distance to closest dot 2) distance to closest ghost

Example: Q-Pacman

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

s



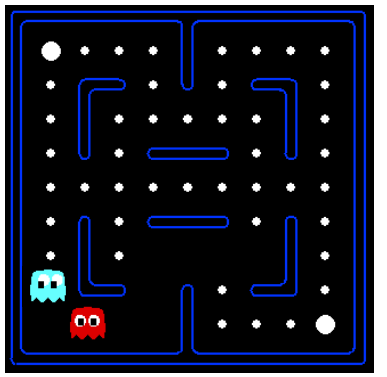
$$f_{DOT}(s, \text{NORTH}) = 0.5$$

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

bad action, we got eaten by the ghost!

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

s'



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

Original Q-value estimate: (4 * 0.5) - (1 * 1) = +1

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

game over, no reward

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

Our sample estimate

difference: -500 - 1 = -501

feature value we had at that state

original w_DOT

$$\text{difference} = -501$$

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

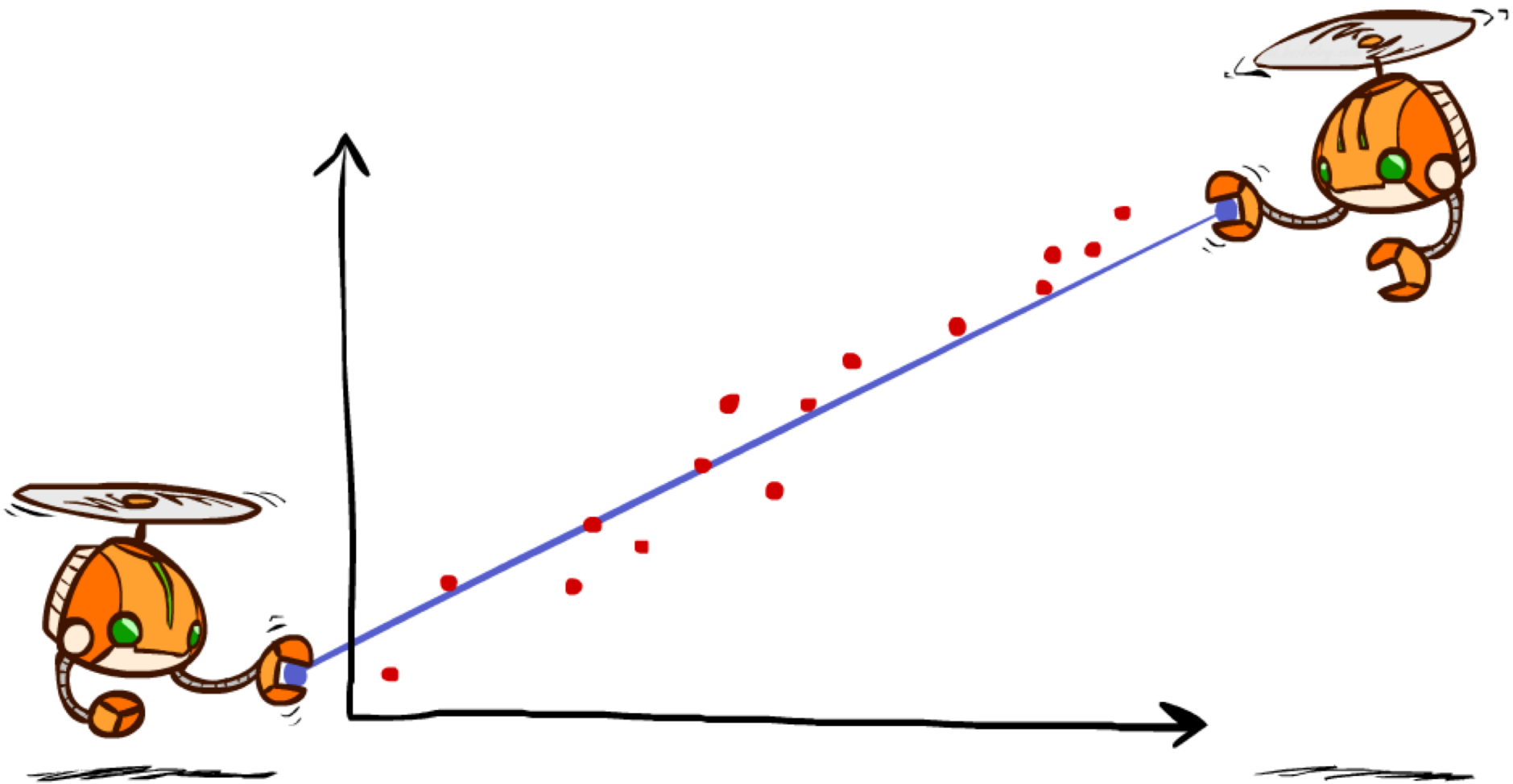
After the update, we get this

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

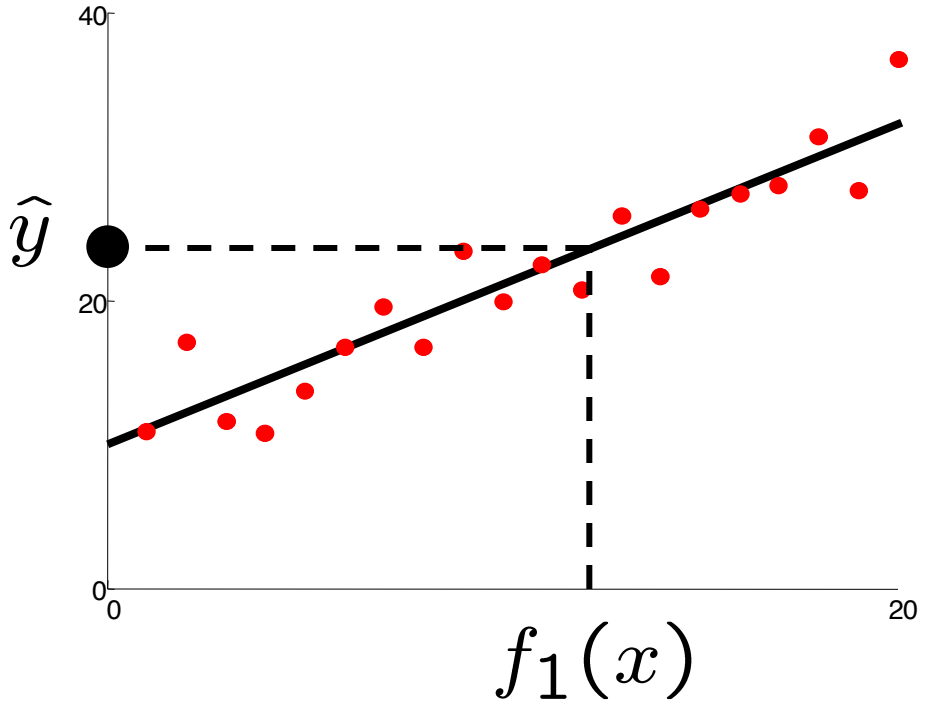
[Demo: approximate Q-learning pacman (L11D10)]

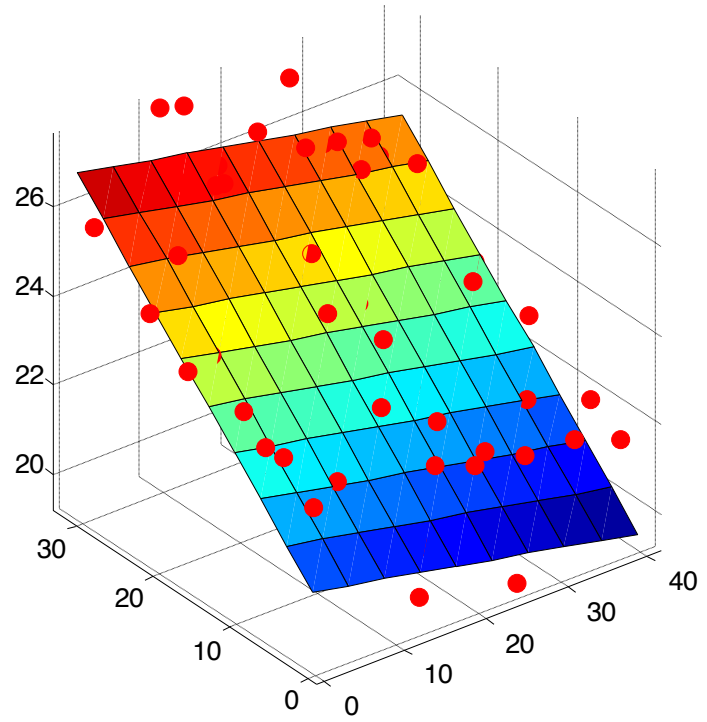
Video of Demo Approximate Q-Learning -- Pacman

Q-Learning and Least Squares



Linear Approximation: Regression*





Prediction:

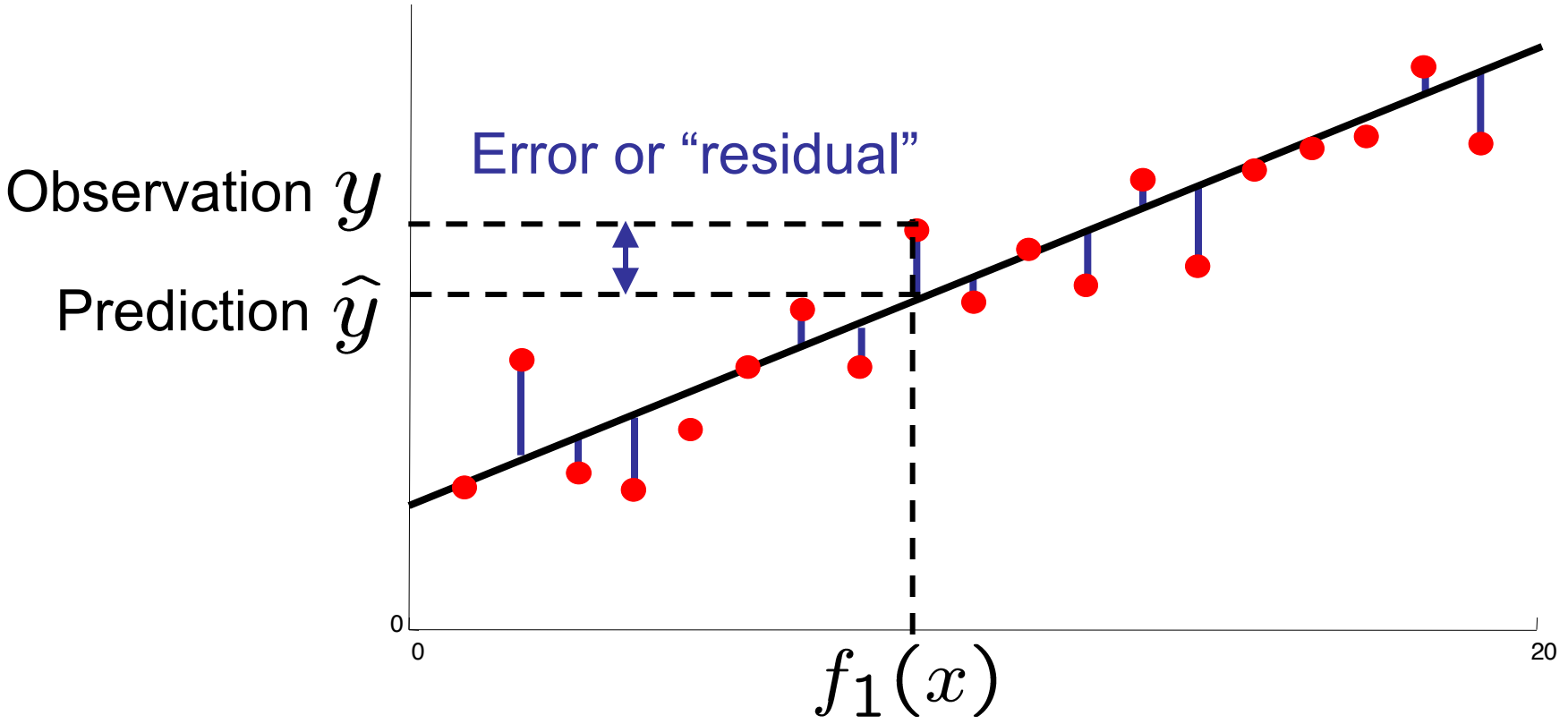
$$\widehat{y} = w_0 + w_1 f_1(x)$$

Prediction:

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

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:

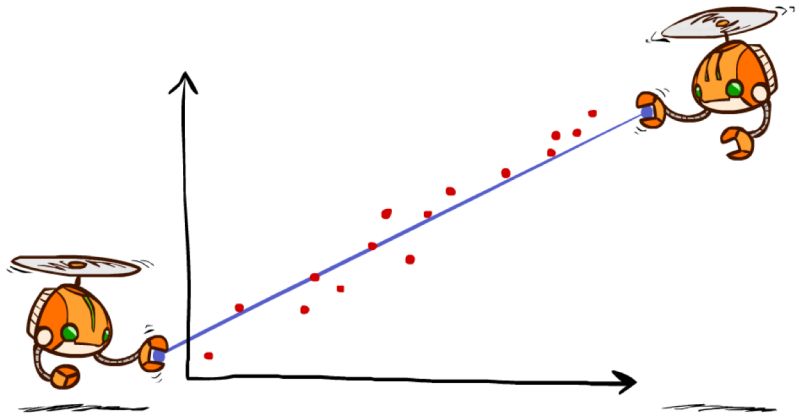
Gradient descent.

we compute the derivative and move in the negative direction

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

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

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



when we take the derivative, let's assume that we get a negative.

this equation says, if we increase w_m by one unit, how much will our error go up.

so in this case, the error actually went down, which is good. Therefore, we want to move w_m in the opposite direction of this, meaning we want it to go up. Thus in the next formulas, we drop the negative.

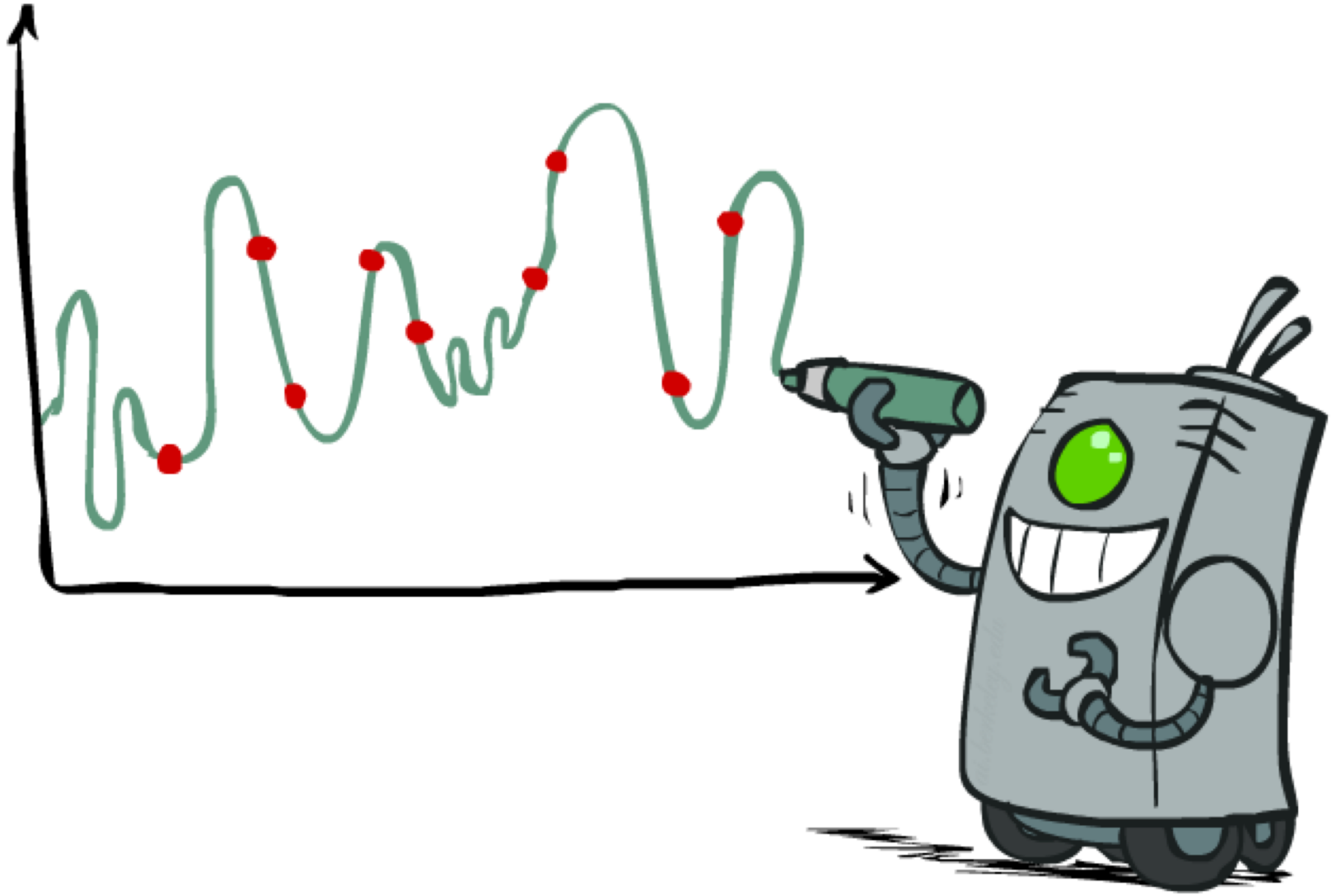
nate q update explained:

the above corresponds to the below

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

"target" "prediction"

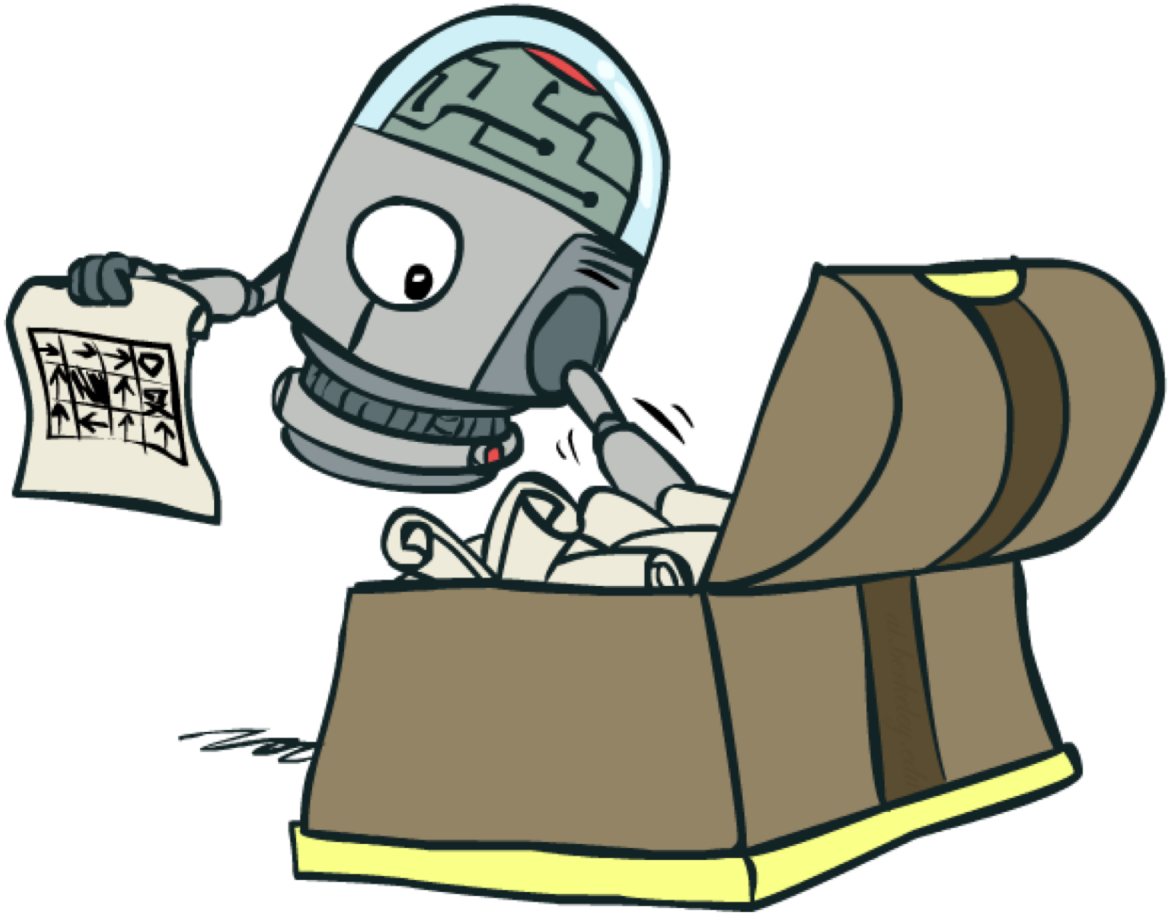
Overfitting: Why Limiting Capacity Can Help*



Policy Search

The goal in MDP and RL has ultimately been to find the best policy.

There are tons of different policies to choose from. In Policy Search, we just say, "forget about learning Q-values or V-values, let's just try about bunch of policies and use whichever is the best.



In approximate Q-values we trying to reduce the loss in the Bellman equation. While this is related to good performance, it's not directly tied to good performance. It's approximating something related to good performance. So it's not exactly clear what you are loosing when you're not able to get an exact approximation of the Bellman equation.

Policy Search

So can we do something else?

- 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) it's not making sure that actions which are the best are favored over actions which are not the best
  - 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

you do this by executing the policy and see how well it does

- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes! policy is less sample efficient than q-learning
  - If there are a lot of features, this can be impractical

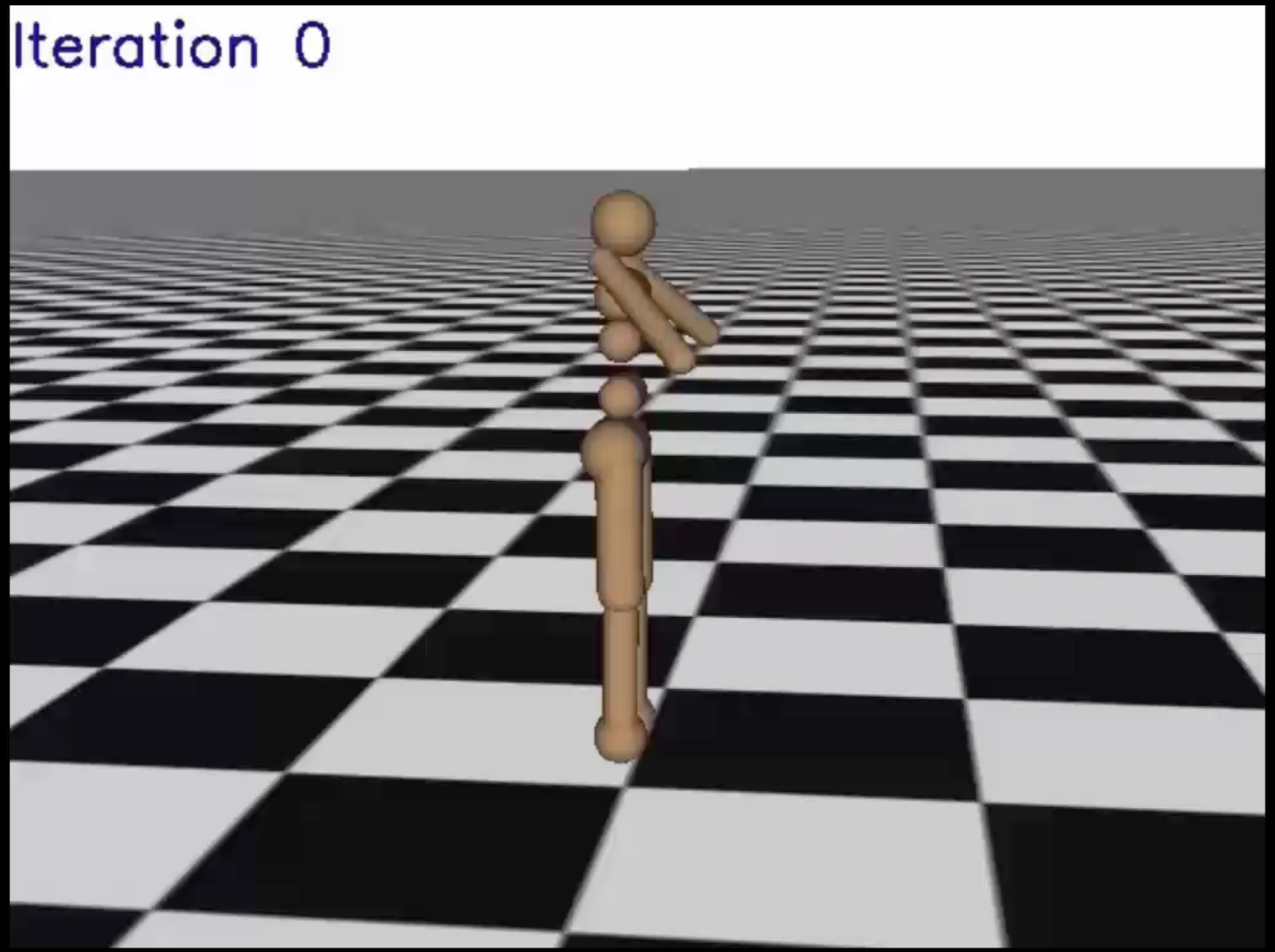
- Better methods exploit lookahead structure, sample wisely, change multiple parameters…

RL: Helicopter Flight

[Andrew Ng]

[Video: HELICOPTER]

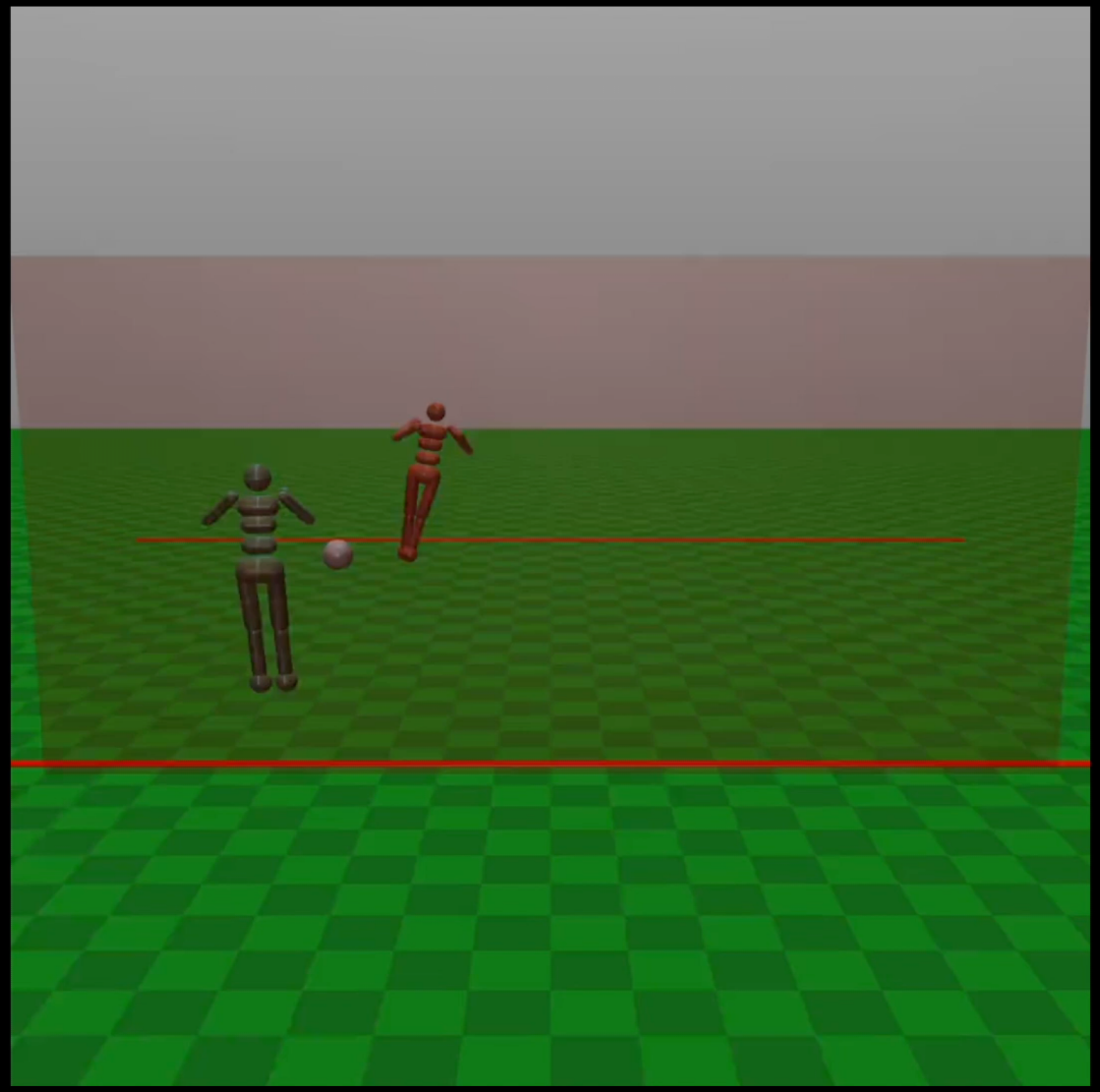
RL: Learning Locomotion



[Schulman, Moritz, Levine, Jordan, Abbeel, ICLR 2016]

[Video: GAE]

RL: Learning Soccer



[Bansal et al, 2017]

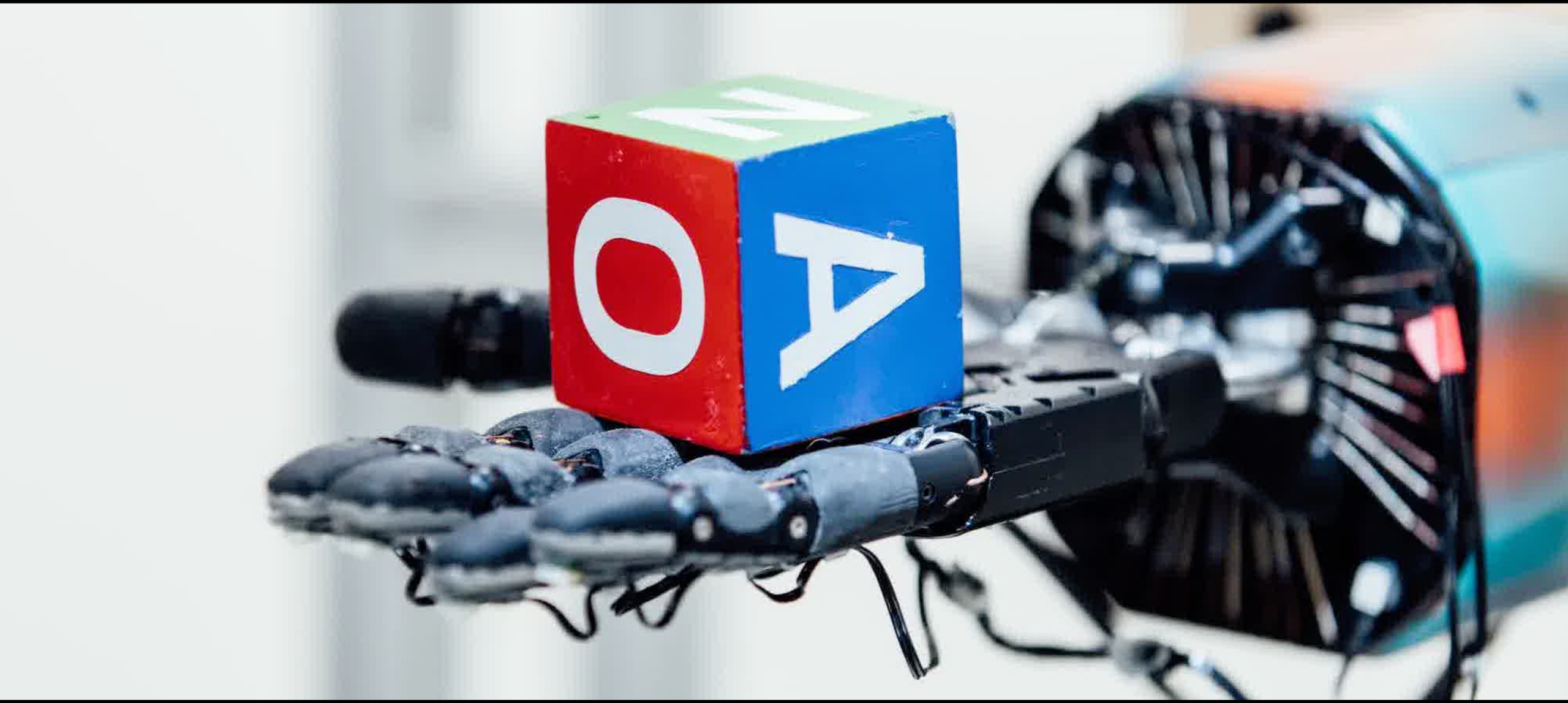
RL: Learning Manipulation

[Levine*, Finn*, Darrell, Abbeel, JMLR 2016]

RL: NASA SUPERball

[Geng*, Zhang*, Bruce*, Caluwaerts, Vespignani, Sunspiral, Abbeel, Levine, ICRA 2017]

RL: In-Hand Manipulation



[OpenAI]

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 up: Part II: Uncertainty and Learning!