Reinforcement Learning

Chris Amato
Northeastern University

Some images and slides are used from: Rob Platt, CS188 UC Berkeley, AIMA

Reinforcement Learning (RL)

Previous session discussed sequential decision making problems where the transition model and reward function were known

In many problems, the model and reward are *not known* in advance

Agent must learn how to act through *experience* with the world

This session discusses *reinforcement learning* (*RL*) where an agent receives a reinforcement signal

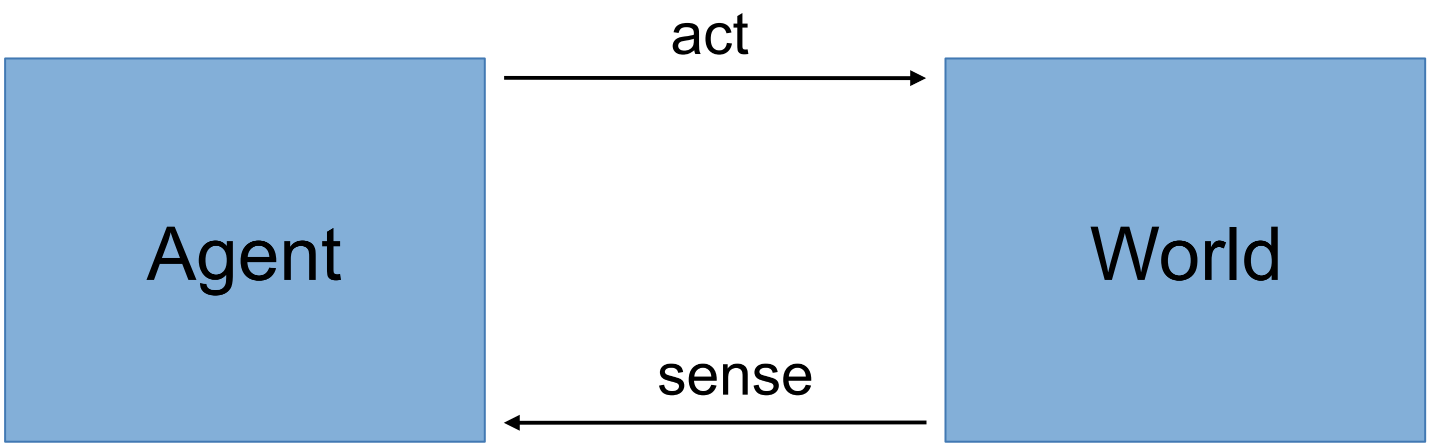
Challenges in RL

Exploration of the world must be balanced with *exploitation* of knowledge gained through experience

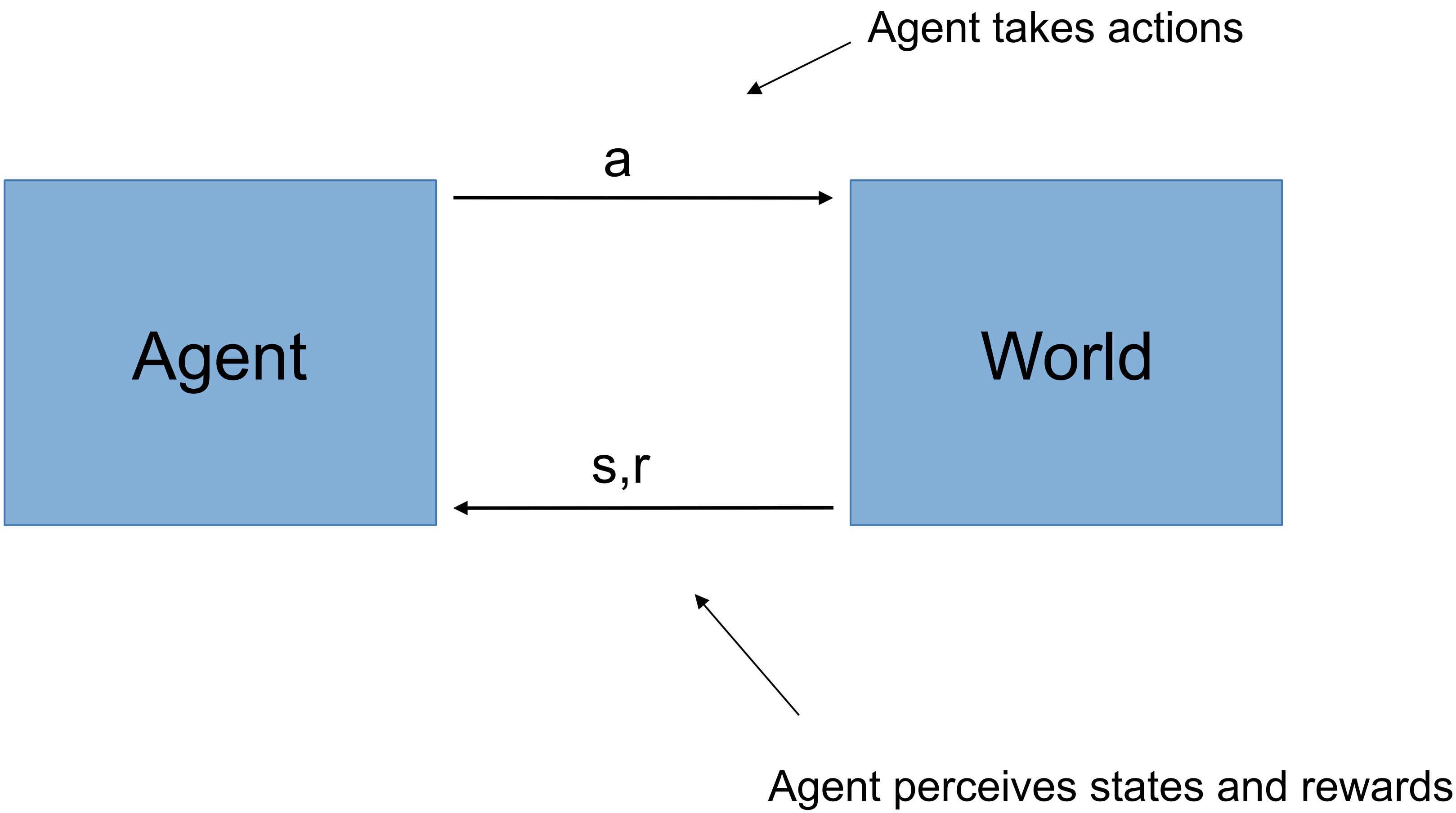
Reward may be received long after the important choices have been made, so *credit* must be assigned to earlier decisions

Must *generalize* from limited experience

Conception of agent



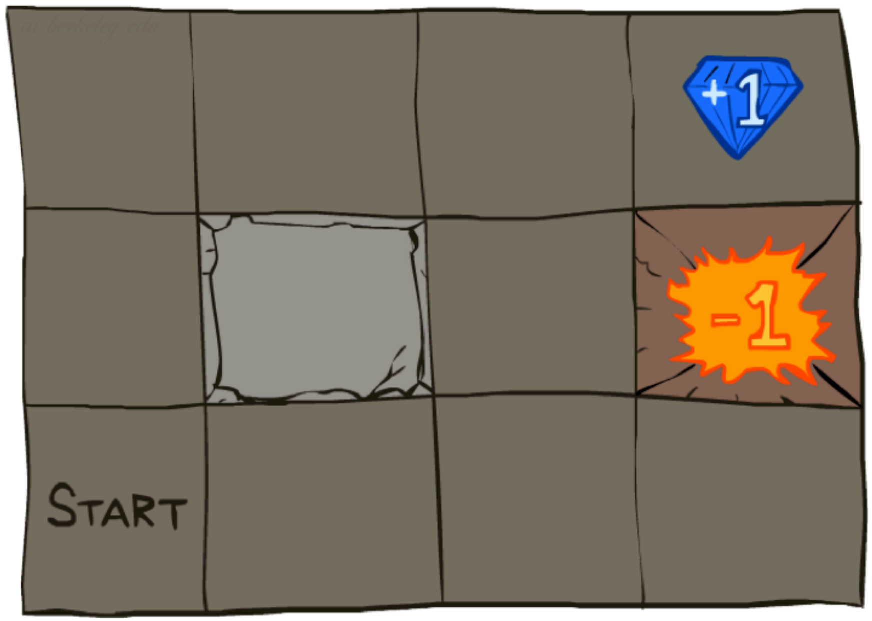
RL conception of agent



Transition model and reward function are initially unknown to the agent!
– value iteration assumed knowledge of these two things...

Value iteration

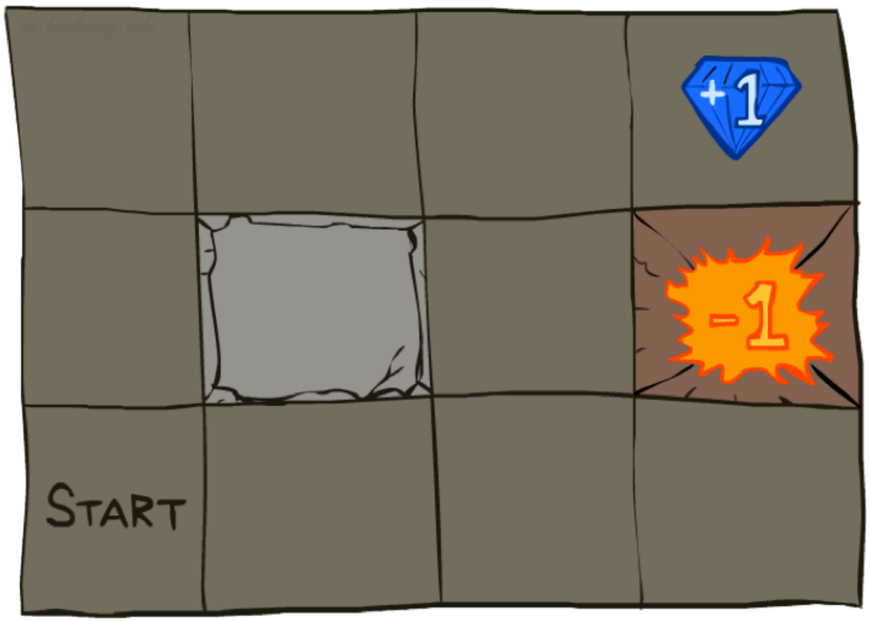
We know the reward function



We know the probabilities of moving in each direction when an action is executed

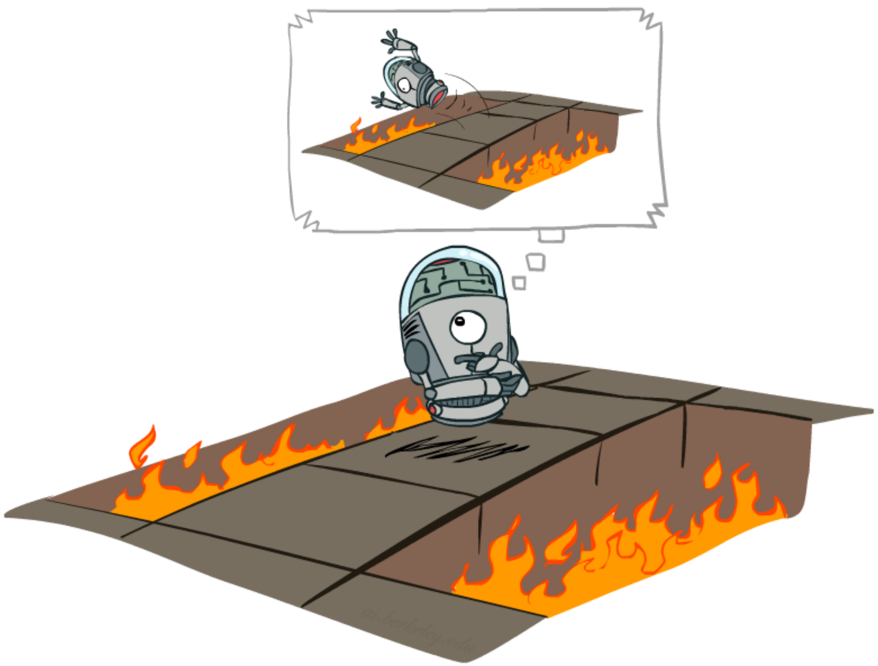
Reinforcement Learning

~~We know the reward function~~



~~We know the probabilities of moving in each direction when an action is executed~~

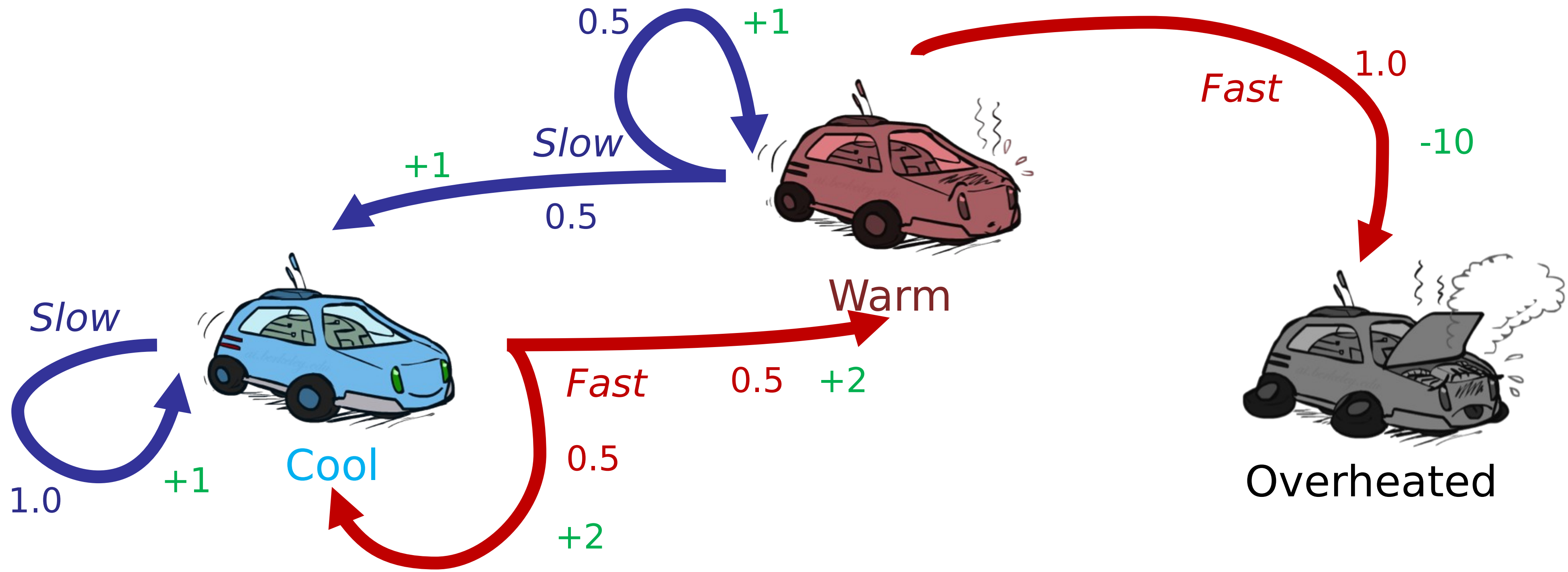
The different between RL and value iteration



Offline Solution
(value iteration)

Online Learning
(RL)

Value iteration vs RL



RL still assumes that we have an MDP

Value iteration vs RL

RL still assumes that we have an MDP

– but, we assume we don't know *T* or *R*

Reinforcement Learning

Still assume a Markov decision process (MDP):

- A set of states s ∈ 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 π(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

Example: Learning to Walk


Initial


A Learning Trial


After Learning [1K Trials]

[Kohl and Stone, ICRA 2004]

Example: Learning to Walk

Initial

[Kohl and Stone, ICRA 2004]

Example: Learning to Walk

Training

[Kohl and Stone, ICRA 2004]

Example: Learning to Walk

Finished

[Kohl and Stone, ICRA 2004]

Video of Demo Crawler Bot

Model-based RL

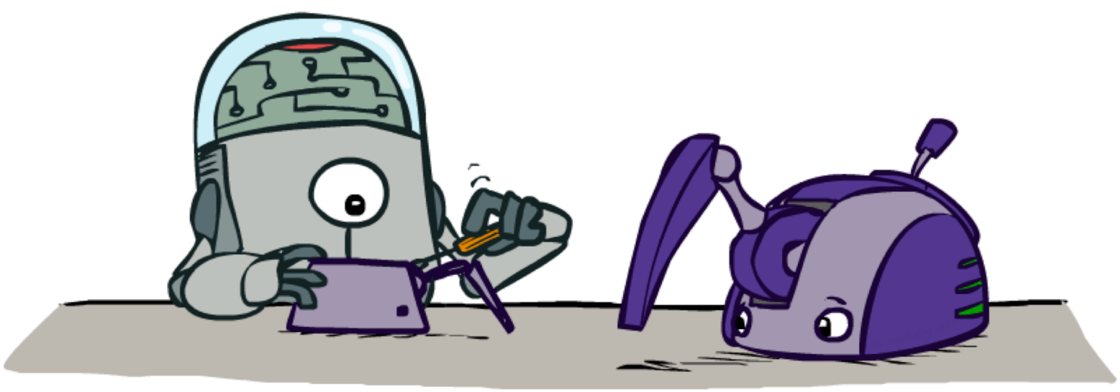
1. estimate T, R by averaging experiences

2. solve for policy in MDP (e.g., value iteration)

a. choose an exploration policy
   – policy that enables agent to explore all relevant states

b. follow policy for a while

c. estimate T and R



Model-based RL

1. estimate T, R by averaging experiences

2. solve for policy in MDP (e.g., value iteration)

a. choose an exploration policy
   – policy that enables agent to explore all relevant states

b. follow policy for a while

c. estimate T and R

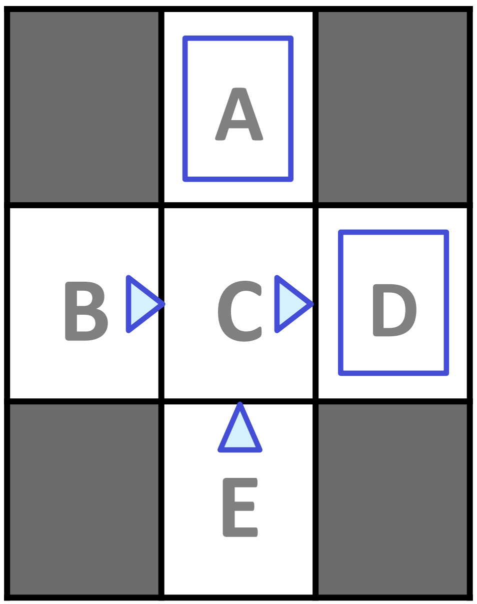
$$N_{s,a,s'} \equiv$$ Number of times agent reached *s'* by taking *a* from *s*

$$R_{s,a,s'} \equiv$$ Set of rewards obtained when reaching *s'* by taking *a* from *s*

$$T(s,a,s') \approx \frac{N_{s,a,s'}}{\sum_{s'} N_{s,a,s'}} \qquad R(s,a,s') \approx \frac{1}{N_{s,a,s'}} R_{s,a,s'}$$

Example: Model-based RL

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

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

T(B, 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
…

Prioritized sweeping

Prioritized sweeping uses a priority queue of states to update (instead of random states)

Key point: set priority based on (weighted) change in value

Pick the highest priority state s to update

Remember current utility $U_{old} = U(s)$

Update the utility: $U(s) \leftarrow \max_a [R(s,a) + \gamma \sum_{s'} T(s' | s,a) U(s')]$

Set priority of s to 0

Increase priority of predecessors s':

increase priority of s' to $T(s | s', a') | U_{old} - U(s) |$

Bayesian RL

Bayesian approach involves specifying a prior over T and R

Update posterior over T and R based on observed transitions and rewards

Problem can be transformed into a *belief state MDP* , with b a probability distribution over T and R

- States consist of pairs (s,b)
- Transition function $T(s',b'|s,b,a)$
- Reward function $R(s',b',a)$

High-dimensional continuous states of belief-state MDP makes them difficult to solve

Model-based RL

1. estimate T, R by averaging experiences

a. choose an exploration policy
– policy that enables agent to explore all relevant states

a while

2. solve for po
(e.g., value ite

R

What is a downside of this approach?

$N_{s,a,s'} \equiv$

s

$R_{s,a,s'} \equiv$

a from *s*

$$T(s,a,s') \approx \frac{N_{s,a,s'}}{\sum_{s'} N_{s,a,s'}} \qquad R(s,a,s') \approx \frac{1}{N_{s,a,s'}} R_{s,a,s'}$$

Model-based vs Model-free learning

Goal: Compute expected age of students in this class

Known P(A)

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

Without P(A), instead collect samples [a_1, a_2, … a_N]

Unknown P(A): "Model Based"

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

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

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

Unknown P(A): "Model Free"

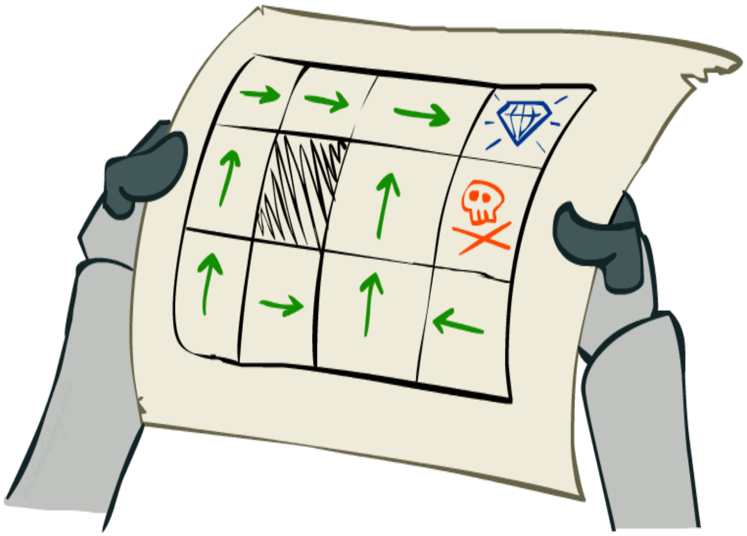
$$E[A] \approx \frac{1}{N} \sum_i a_i$$

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

Policy evaluation

Simplified task: policy evaluation

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



In this case:

- Learner is "along for the ride"
- No choice about what actions to take
- Just execute the policy and learn from experience
- This is NOT offline planning! You actually take actions in the world.

Direct evaluation

Goal: Compute values for each state under π

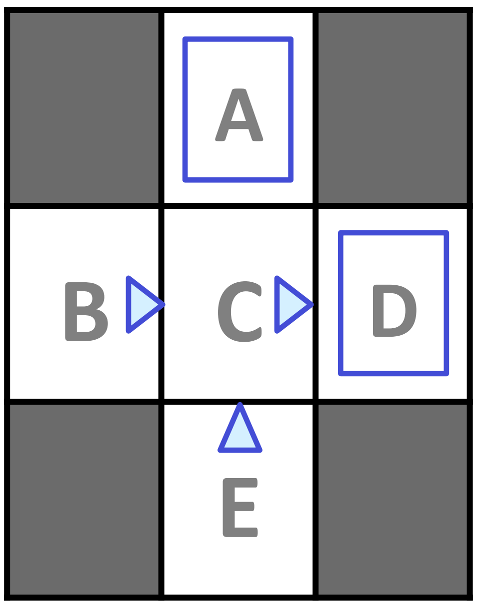
Idea: Average together observed sample values

- Act according to π
- Every time you visit a state, write down what the sum of discounted rewards turned out to be
- Average those samples

This is called direct evaluation

Example: Direct evaluation

Input Policy
π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

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

Episode 2

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

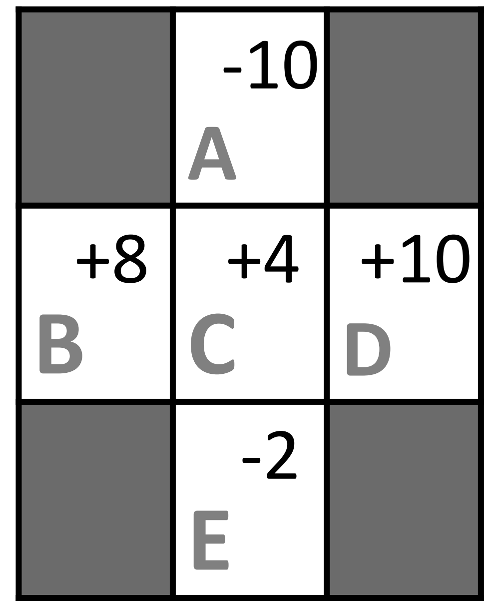
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



Problems with direct evaluation

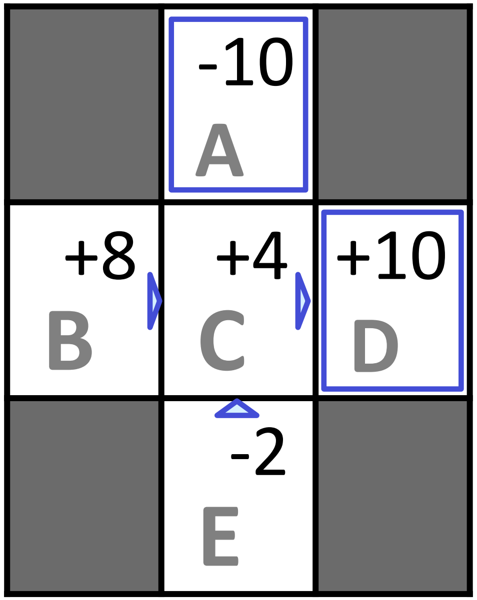
What's good about direct evaluation?

- 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?

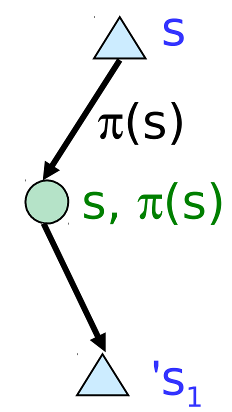
Sample-Based Policy Evaluation

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

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

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

$$sample_1 = R(s, \pi(s), s_1') + \gamma V_k^{\pi}(s_1')$$
$$\ldots$$



Sample-Based Policy Evaluation

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

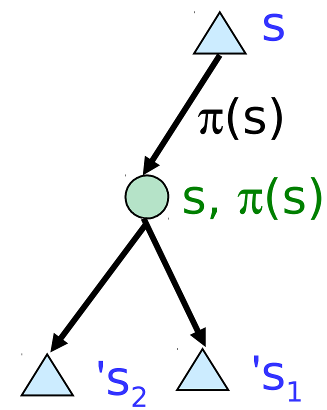
$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')]$$

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

$$sample_1 = R(s,\pi(s),s_1') + \gamma V_k^{\pi}(s_1')$$

$$\ldots$$

$$sample_2 = R(s,\pi(s),s_2') + \gamma V_k^{\pi}(s_2')$$



Sample-Based Policy Evaluation

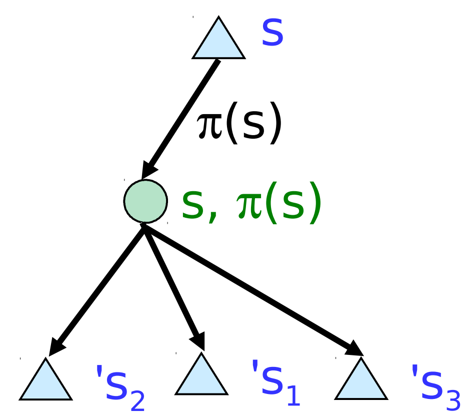
- We want to improve our estimate of V by computing these averages:

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

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

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1)$$
$$\ldots$$
$$sample_2 = R(s, \pi(s), s'_2) + \gamma V_k^{\pi}(s'_2)$$

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



Sample-Based Policy Evaluation

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

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

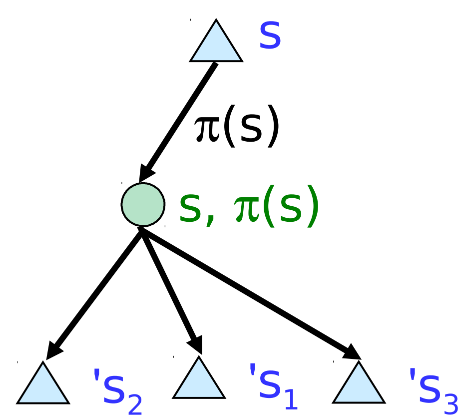
- Idea: Take samples of outcomes s' (by doing the action!) and average

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

$$\ldots$$

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

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



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

Sidebar: incremental estimation of mean

Suppose we have a random variable X and we want to estimate the mean from samples $x_1,\ldots,x_k$

After k samples $\hat{x}_k = \frac{1}{k}\sum_{i=1}^{k} x_i$

Can show that $\hat{x}_k = \hat{x}_{k-1} + \frac{1}{k}(x_k - \hat{x}_{k-1})$

Can be written $\hat{x}_k = \hat{x}_{k-1} + \alpha(k)(x_k - \hat{x}_{k-1})$

Learning rate $\alpha(k)$ can be functions other than 1, loose k conditions on learning rate to ensure convergence to mean

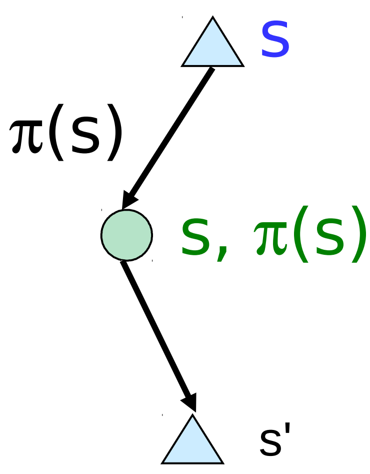
If learning rate is constant, weight of older samples decay exponentially at the rate $(1 - \alpha)$

Forgets about the past (distant past values were wrong anyway)

Update rule $\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x})$

TD Value 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 (incremental mean)

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

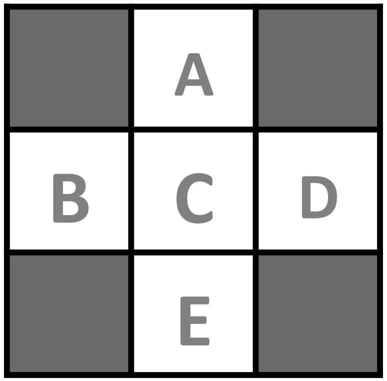
Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha) sample$

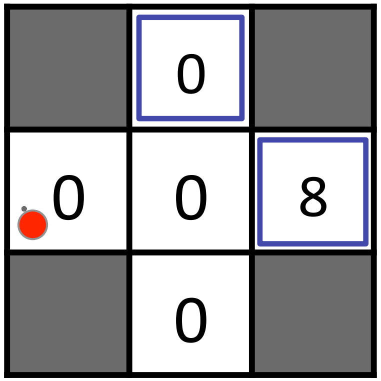
Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

TD Value Learning: example

Observed Transitions

States





Assume: γ = 1,
α = 1/2

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

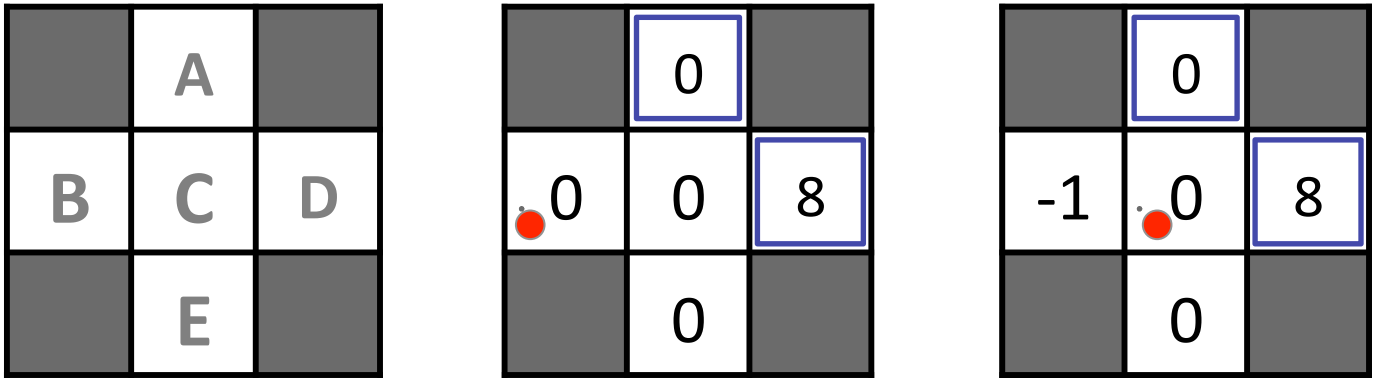
TD Value Learning: example

Observed reward

States

Observed Transitions

B, east, C, -2



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

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

TD Value Learning: example

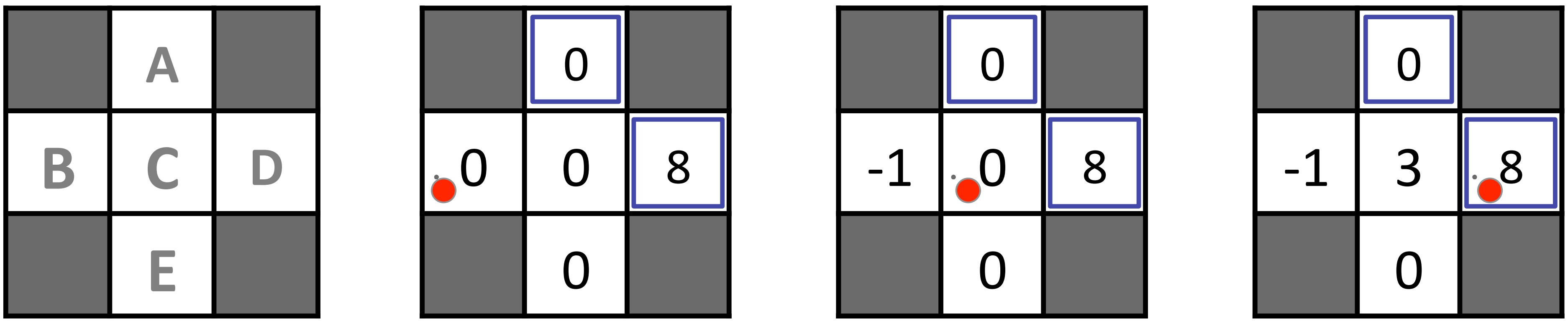
Observed reward

Observed Transitions

States

B, east, C, -2

C, east, D, -2



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

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

What's the problem w/ TD Value Learning?

What's the problem w/ TD Value Learning?

Can't turn the estimated value function into a policy!

This is how we did it when we were using value iteration:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Why can't we do this now?

What's the problem w/ TD Value Learning?

Can't turn the estimated value function into a policy!

This is how we did it when we were using value iteration:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Why can't we do this now?

<u>Solution: Use TD value learning to estimate Q*, not V*</u>

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:

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

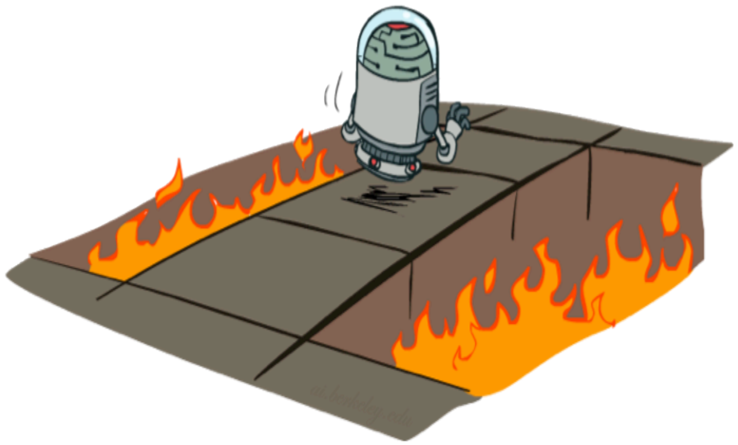
- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$, which we know is right
  - Given Q_k, calculate the depth k+1 q-values for all q-states:

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

Active Reinforcement Learning

Full reinforcement learning: generate 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 RL

Model-free (temporal difference) learning

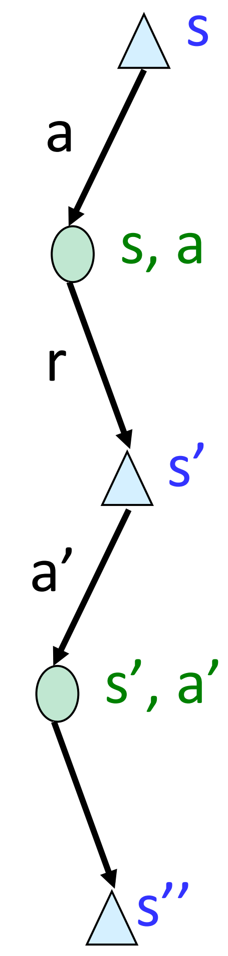
Experience world through episodes

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

Update estimates each transition

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

Over time, updates will mimic Bellman updates



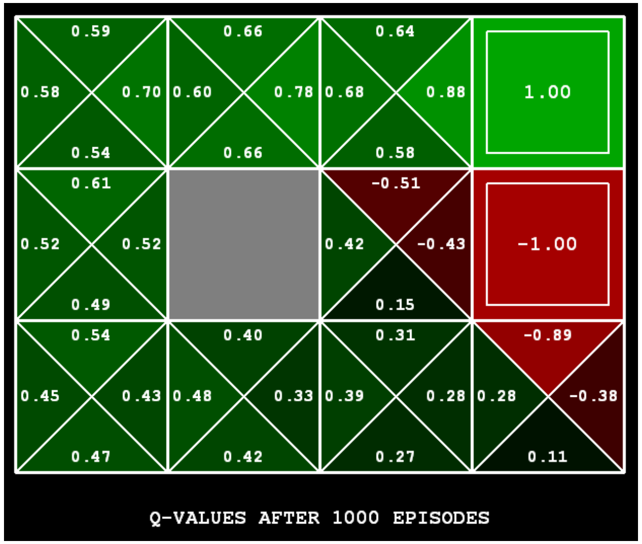
Q-Learning

- 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:

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



  - Incorporate the new estimate into a running average:

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

Q-Learning video -- Crawler

Q-Learning: properties

Q-learning converges to optimal Q-values if:

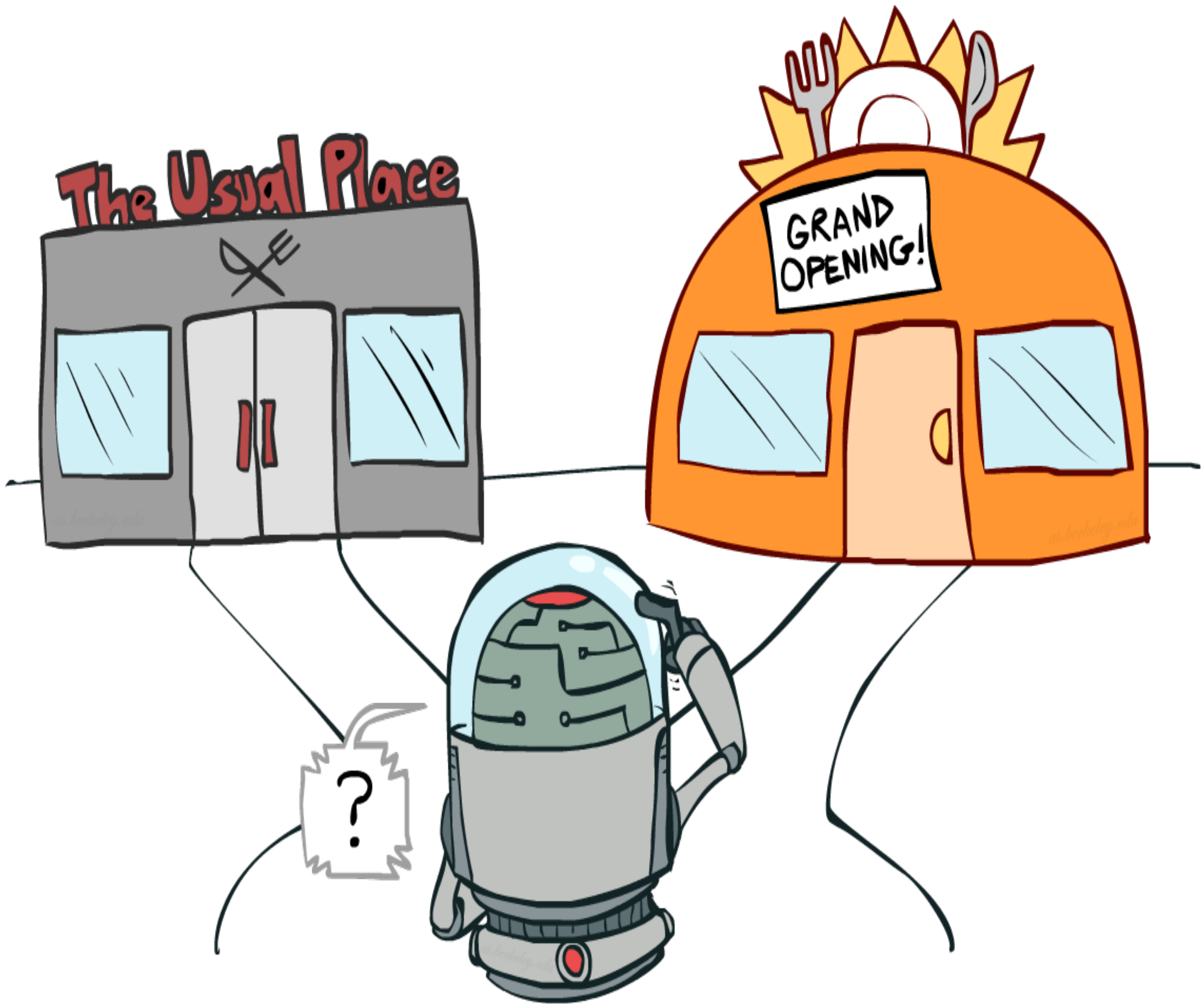
1. it explores every s, a, s' transition sufficiently often

2. the learning rate approaches zero (eventually)

Key insight: Q-value estimates converge even if experience is obtained using a suboptimal policy.

This is called off-policy learning

Exploration vs. exploitation



How to explore?

Several schemes for forcing exploration

Simplest: random actions (Ɛ-greedy)

- Every time step, flip a coin
- With (small) probability Ɛ, act randomly
- With (large) probability 1-Ɛ, 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 Ɛ over time
- Another solution: exploration functions

Q-Learning video – Crawler with epsilon-greedy

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!

Q-Learning video – Crawler with exploration function

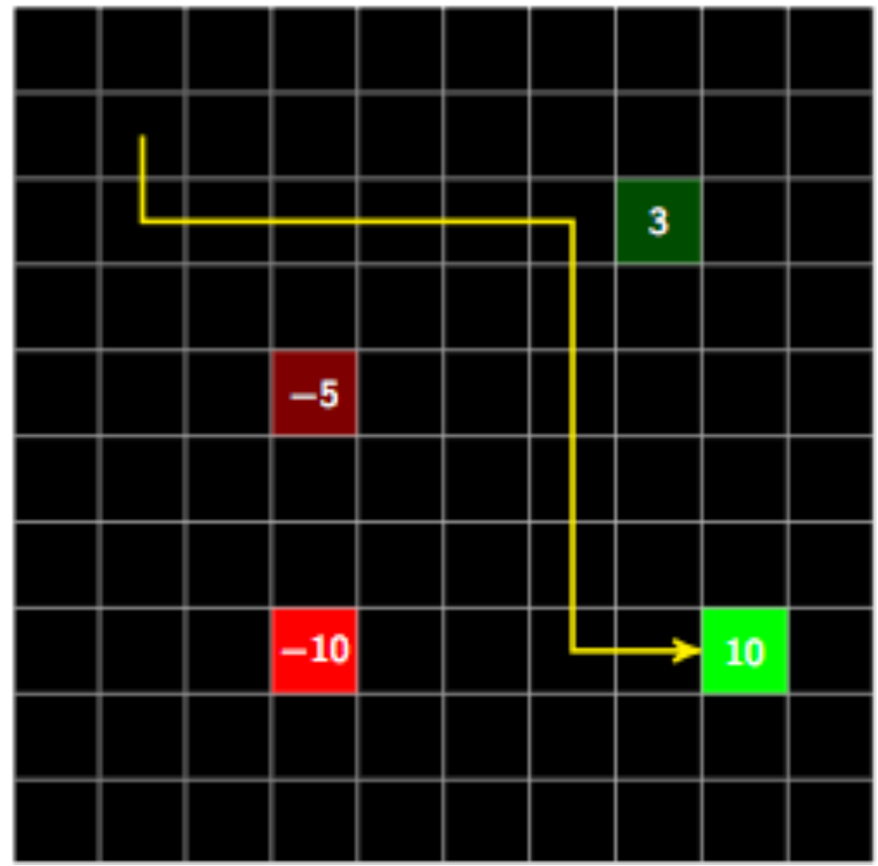
Q-Learning

Q-learning will converge to the optimal policy

However, Q-learning typically requires a *lot of experience*

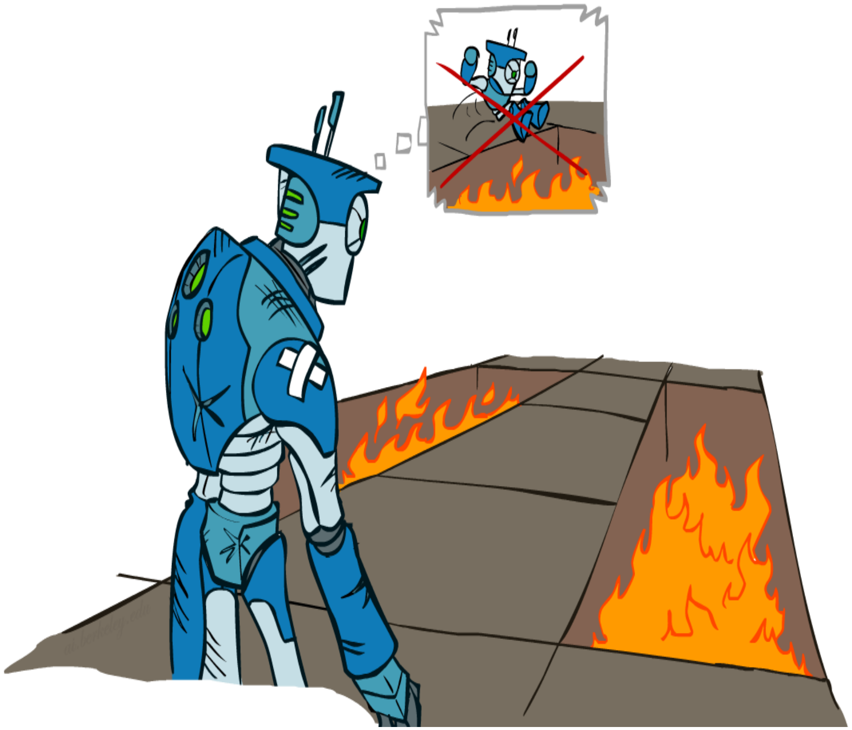
Utility is updated one step at a time

Eligibility traces allow states along a path to be updated



Regret

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



Generalizing across states

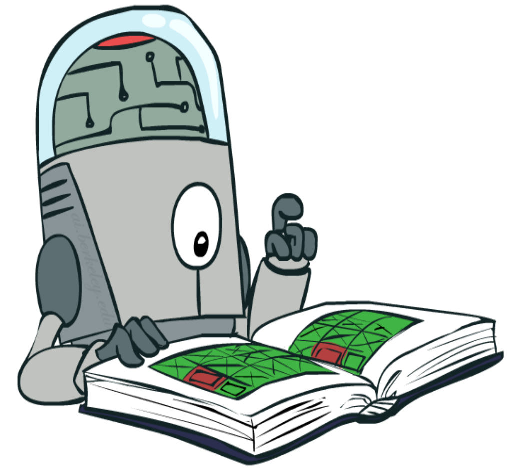
Basic Q-Learning keeps a table of all q-values

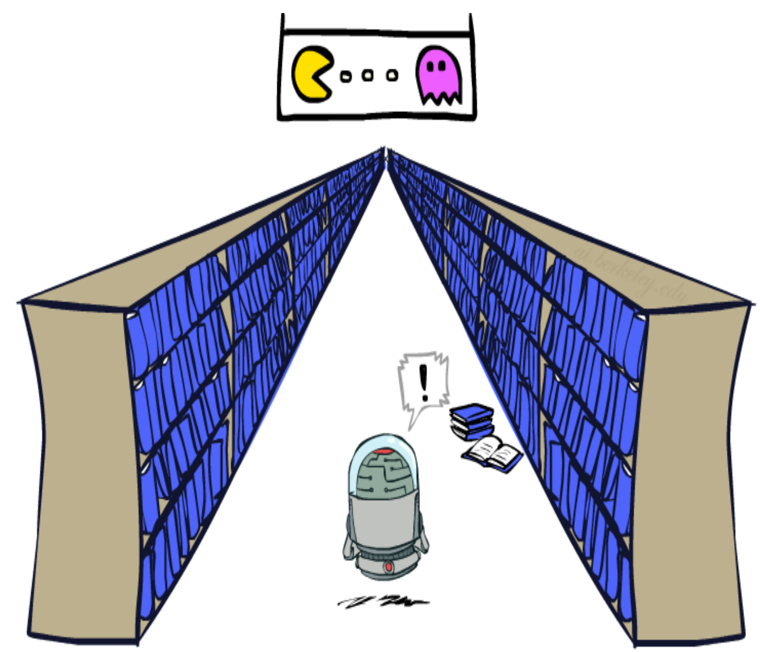
In realistic situations, we cannot possibly learn about every single state!

- Too many states to visit them all in training
- Too many states to hold the q-tables in memory

Instead, we want to generalize:

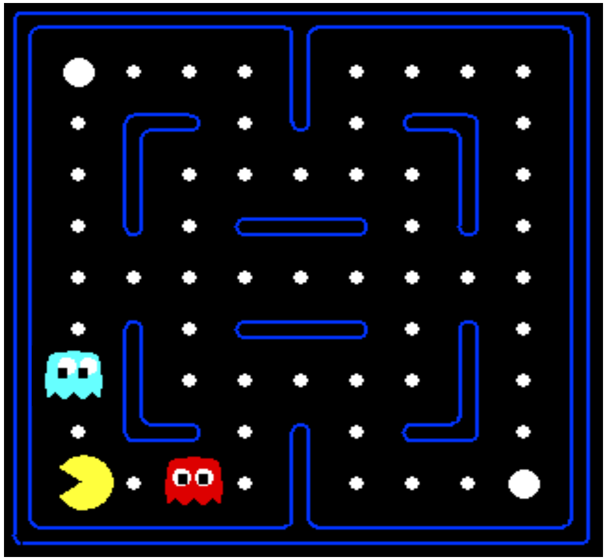
- Learn about some small number of training states from experience
- Generalize that experience to new, similar situations
- This is a fundamental idea in machine learning, and we'll see it over and over again



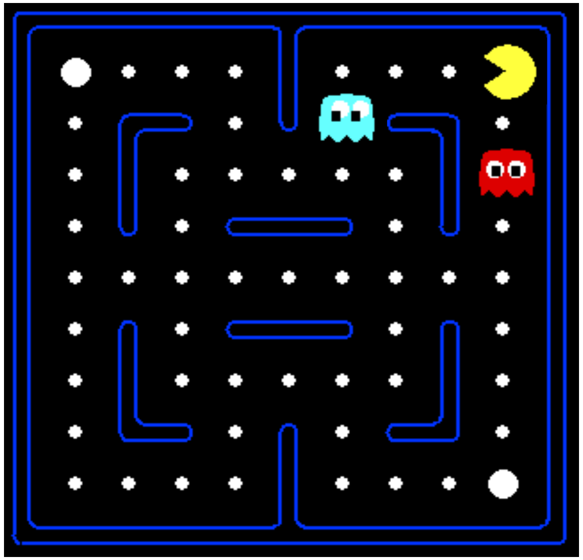


Example: Pac-man

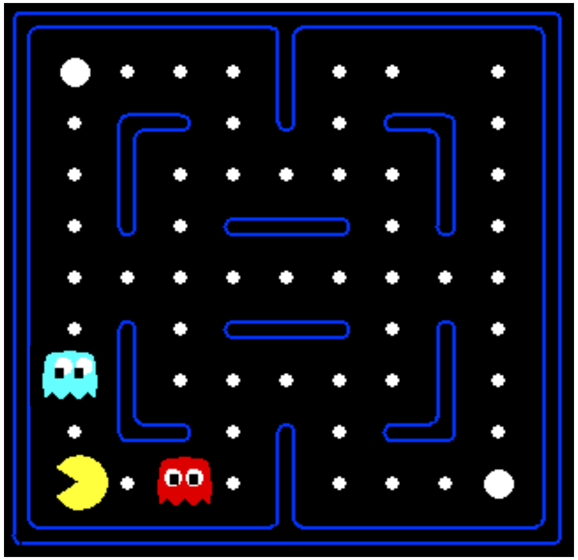
We discover through experience that this state is bad:



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



Or even this one!

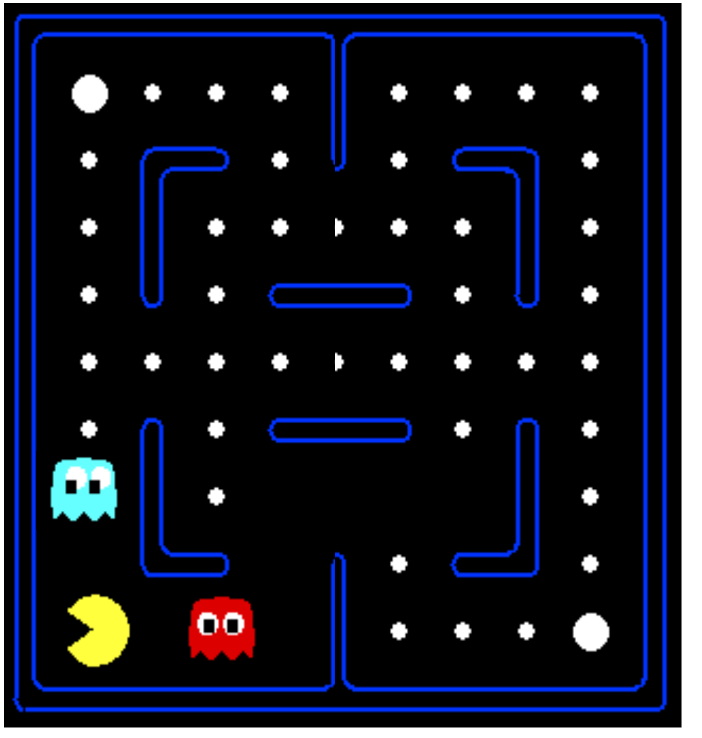


Q-Learning video – Pacman Tiny

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.
- 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!

Approximate Q-learning

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

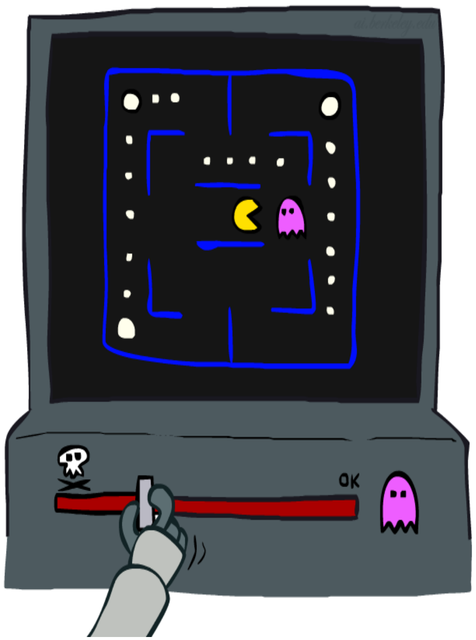
Q-learning with linear Q-functions:

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

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

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

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



Intuitive interpretation:

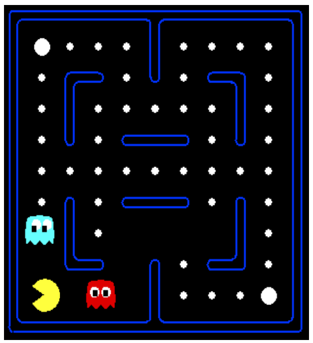
- Adjust weights of active features
- E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

Formal justification: online least squares

Example: Q-Pacman

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

s



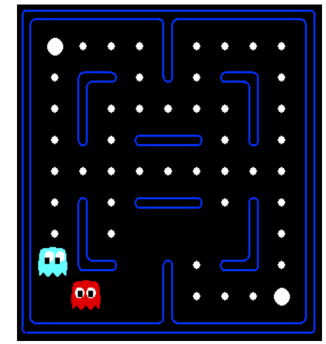
$f_{DOT}(s, \text{NORTH}) = 0.5$

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

$a = \text{NORTH}$

$r = -500$

s'



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

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

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

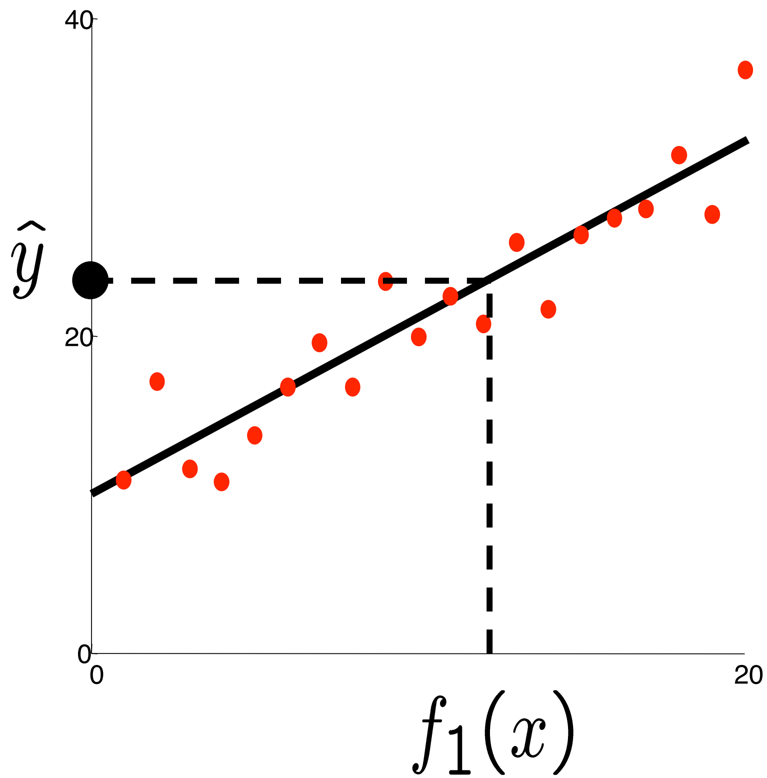
difference $= -501$

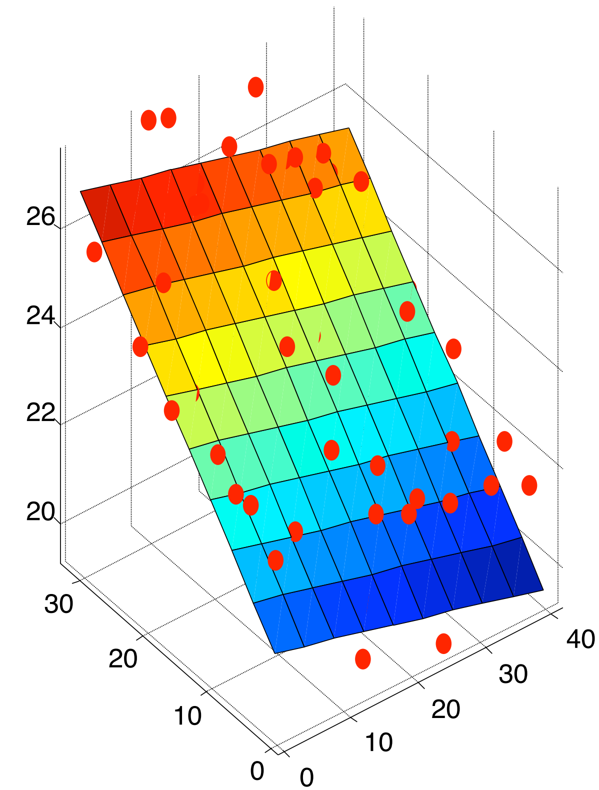
$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$

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

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

Linear Approximation: Regression





Prediction:

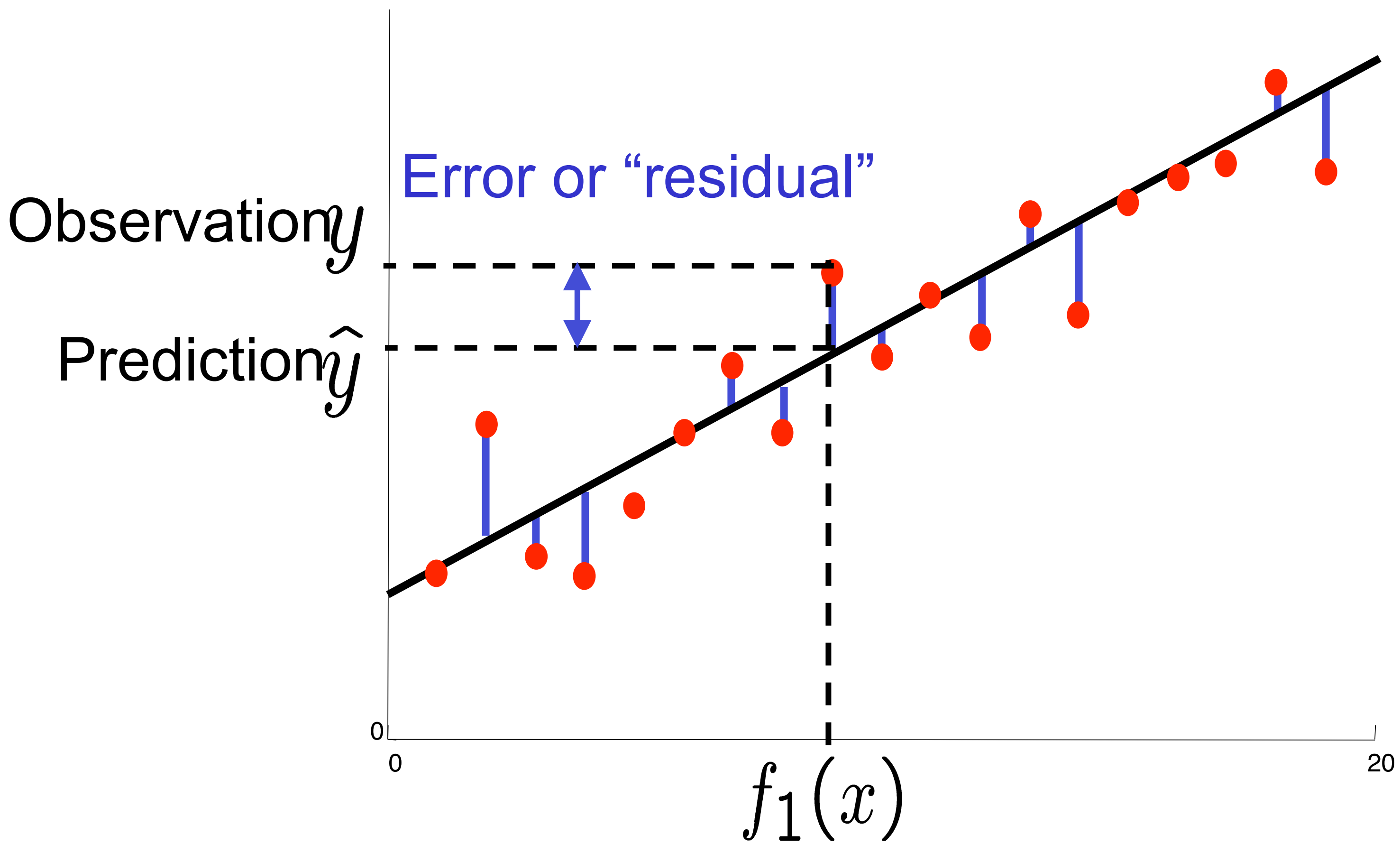
$$\hat{y} = w_0 + w_1 f_1(x)$$

Prediction:

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

Optimization: Least Squares

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



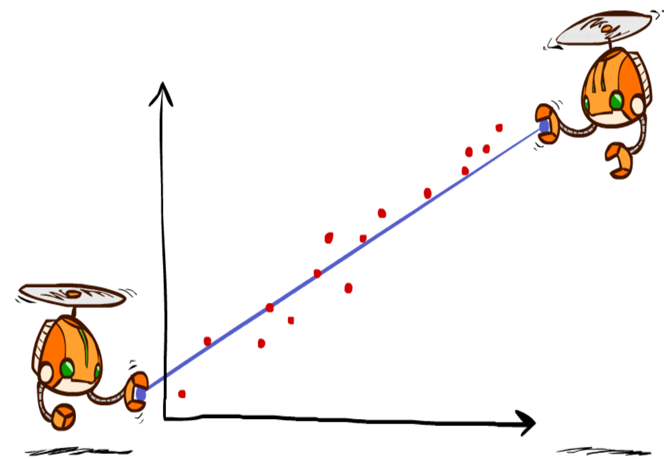
Minimizing error

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

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

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

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

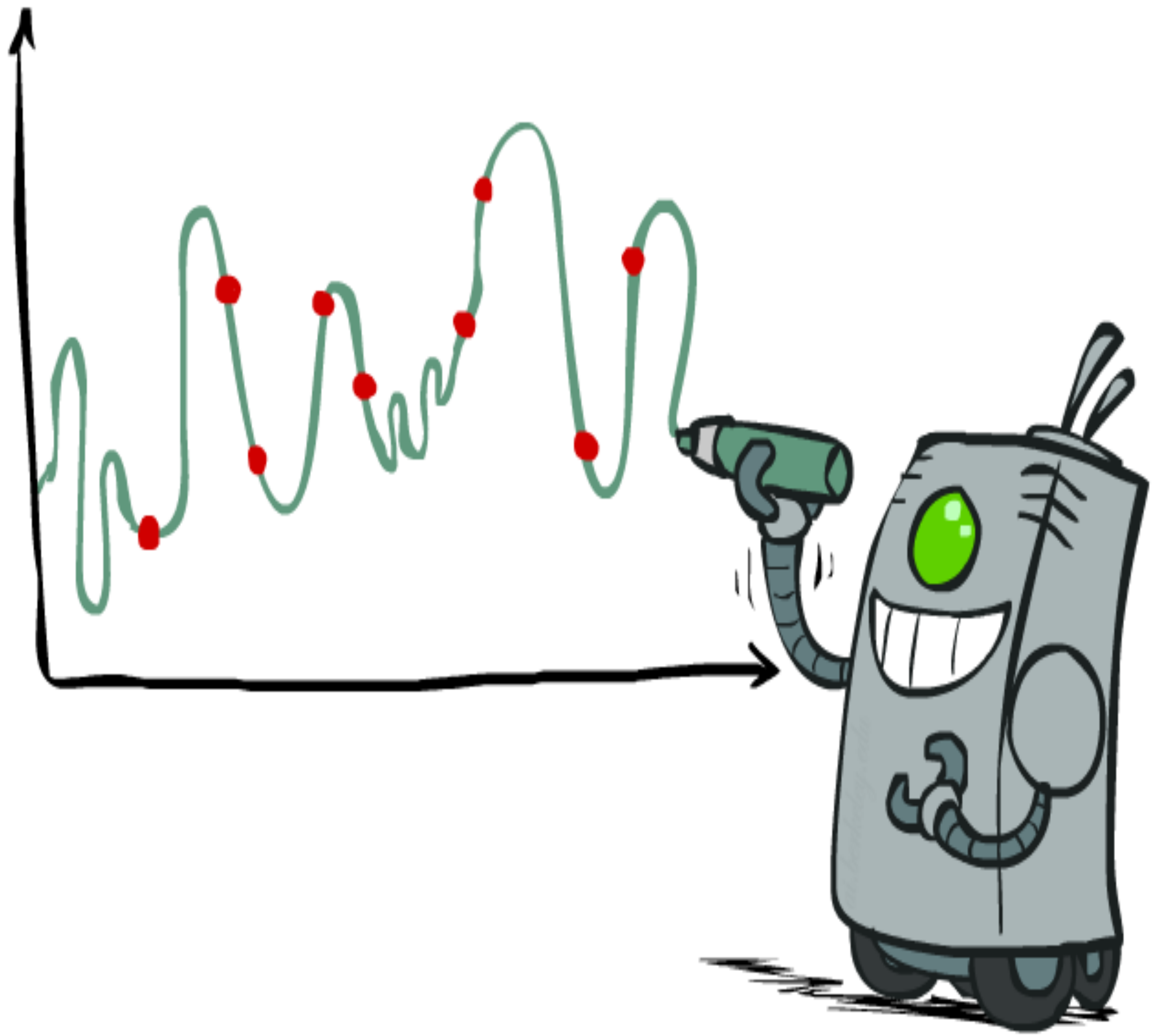


Approximate q update explained:

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

“target” “prediction”

Overfitting: Why limiting capacity can help



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)
  - We'll see this distinction between modeling and prediction again later in the course

- Solution: learn policies that maximize rewards, not the values that predict them

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

Policy search

Simplest policy search:

- Start with an initial linear value function or Q-function
- Nudge each feature weight up and down and see if your policy is better than before

Problems:

- How do we tell the policy got better?
- Need to run many sample episodes!
- If there are a lot of features, this can be impractical

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

Policy search: autonomous helicopter

[Andrew Ng]

Summary

Reinforcement learning is a computational approach to learning intelligent behavior from experience

Exploration must be carefully balanced with exploitation

Credit must be assigned to earlier decisions

Must generalize from limited experience

Next session will start looking at graphical models for representing uncertainty

Overview: MDPs and RL

Known MDP: Offline Solution

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

Unknown MDP: Model-Based

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

Unknown MDP: Model-Free

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