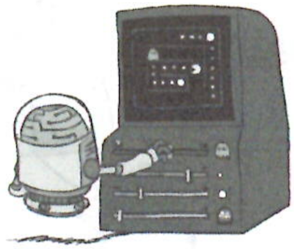
CS 188: Artificial Intelligence
Reinforcement Learning II



Instructors: Dan Klein and Pieter Abbeel --- 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 (optimal) 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 i.e., $\sum_{s'} T(s,a,s')$ ...
  ~~(i.e., compute estimate of T)~~

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

Unknown MDP: Model-Based

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

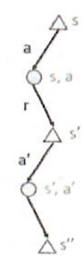
Unknown MDP: Model-Free

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

analogs of VI/PI and PE for MDPs

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

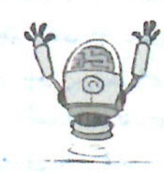


Q-Learning

- 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
    $$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$
    layer of optimality
  - But we want to average over results from (s,a) (Why?) — any 1 sample may be misrepresentative
  - So keep a running average
    $$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

Q-Learning Properties

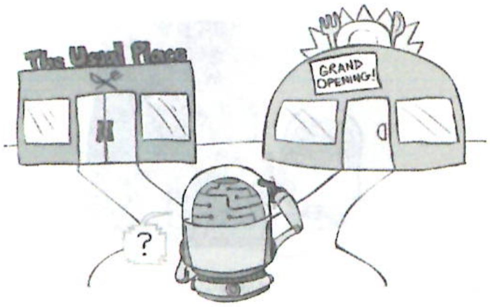
- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - ... but not decrease it too quickly
  - Basically, in the limit, it doesn't matter how you select actions (!)



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

Video of Demo Q-Learning Auto Cliff Grid

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 $1-\varepsilon$, 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 — still ~~not~~ good
    - Another solution: exploration functions

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

- Our first foray into choosing actions

Video of Demo Q-learning – Manual Exploration – Bridge Grid

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

- Even though Crawler learned how to move forward w/ $\varepsilon = 0.8$, this ε is so high that ~~it had~~ the probability of a chain of actions leading to forward movement was very low, so Crawler didnt move forward. Once ε was set to 0, Crawler followed the optimal actions for the policy it learned so far and moved forward although slowly compared to if it had trained for longer.

Exploration Functions

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established, eventually stop exploring

k is a 'bonus'

- 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!

→ trying things that are unknown that lead to unknown states

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

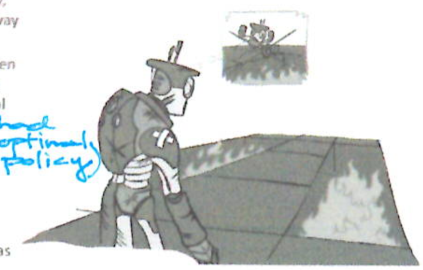
- $\leftarrow_\alpha$ means avg in RHS quantity w/ weight α

Video of Demo Q-learning – Exploration Function – Crawler

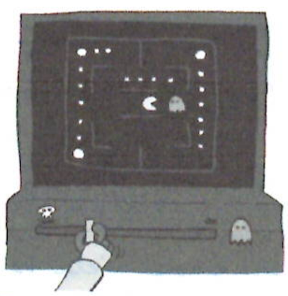
- w/ $\xi = 0.1$ + an exploration func., it ~~quickly learns learns some good behavior~~ more quickly adopts good behavior than before even though this good behavior is far from optimal.
  - w/ exploration func., bot quickly learns some states are bad + stops "consciously" exploring them (still may visit ~~bc~~ though bc $\xi > 0$).

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 (if you had optimal policy)
- Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret



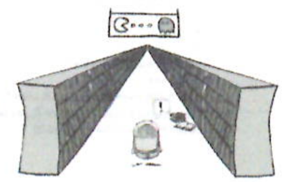
Approximate Q-Learning



- when you learn a ghost is bad, you should transfer that info to other relevant states

Generalizing Across States

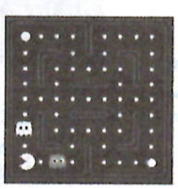
- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training 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
- Faster learning



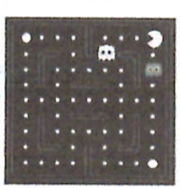
[demo – RL pacman]

Example: Pacman

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

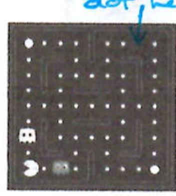


In naïve q-learning, we know nothing about this state:



Or even this one!

missing a dot here



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

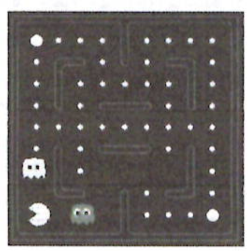
Video of Demo Q-Learning Pacman – Tiny – Watch All

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)



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!

• 2 ghosts on same side ~~are~~ is ~~much less dangerous than~~ better state than 2 ghosts ~~wf~~ on diff sides, like in image to the left.

• Must make features ~~that can~~ ~~so that~~ that discriminate diff. utility ~~states~~ state/action pairs well

Approximate Q-Learning

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

- Q-learning with linear Q-functions:

transition $= (s, a, r, s')$ – same as b4

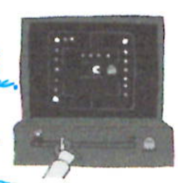
difference $= \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s,a)$ – diff b/t what you get and what you expec.

can't do this know bc we have features

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

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

do this instead



- 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

→ ? generalization

• this is just a gradient descent update

"activation" of the feature

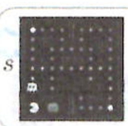
* How is convergence to the optimal policy affected by ~~appr~~ using approximate Q-learning?

Do the features need to explain the underlying process entirely?

Example: Q-Pacman

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

s



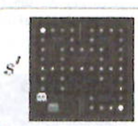
$f_{DOT}(s, \text{NORTH}) = 0.5$

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

$a = \text{NORTH}$

$r = -500$

s'



1.0 > 0.5 → ghost closer than dot ∴

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

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

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

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

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

ghost feature "more active" than dot feature

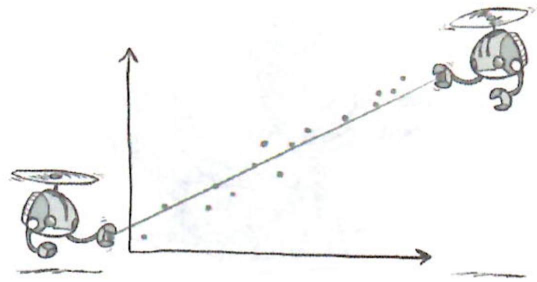
$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

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

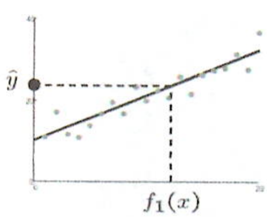
credit assignment

Video of Demo Approximate Q-Learning -- Pacman

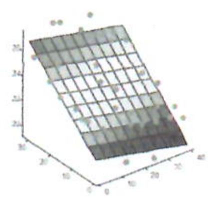
Q-Learning and Least Squares



Linear Approximation: Regression*



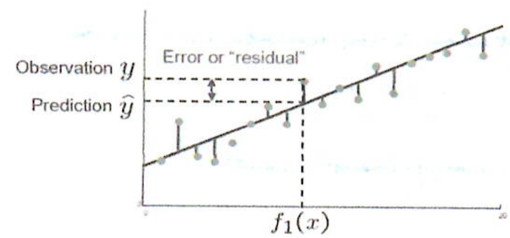
Prediction:
$\hat{y} = w_0 + w_1 f_1(x)$



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

Optimization: Least Squares*

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



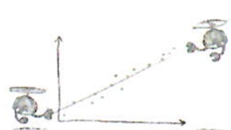
Minimizing Error*

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

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

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

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

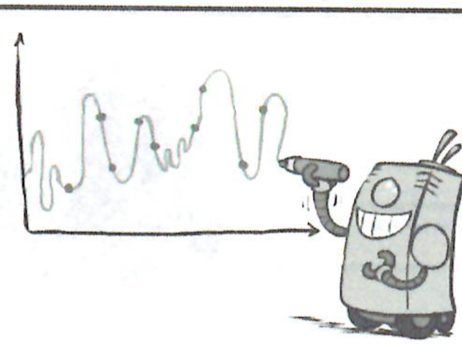


Approximate q update explained:

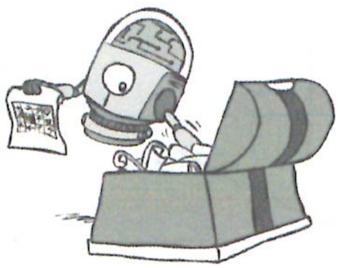
$$w_m \leftarrow w_m + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] f_m(s, a)$$

"target" "prediction"

Overfitting: Why Limiting Capacity Can Help*



Policy Search



- In general, Q-learning ~~only~~ takes you only so far

→ policy search to improve

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) – better modeling → more ~~closer~~ accurate Q-vals.
  - Action selection priority: get ordering of Q-values right (prediction) – better pred. → more ~~better or~~ accurate ordering + decision making
  - We'll see this distinction between modeling and prediction again later in the course
  - tradeoff b/t modeling + prediction
- Solution: learn policies that maximize rewards, not the values that predict them
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

Policy Search

- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - Nudge each feature weight up and down and see if your policy is better than before
  - Don't need Q-val update step here
- 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
  - → usually insufficient
- Better methods exploit lookahead structure, sample wisely, change multiple parameters...

Policy Search

[Andrew Ng]

[Video: HELICOPTER]

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!