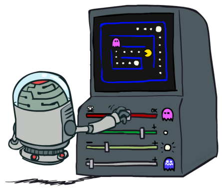
CS 473: Artificial Intelligence

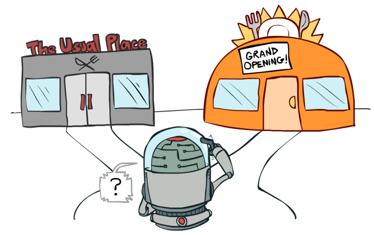
Reinforcement Learning II



Dieter Fox / University of Washington

[Most slides were taken from Dan Klein and Pieter Abbeel / CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

Exploration vs. Exploitation



How to Explore?

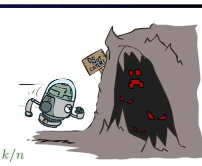
- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on ***current policy***
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

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

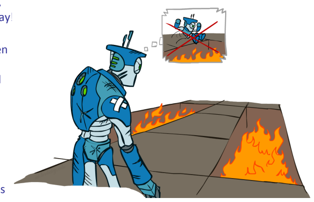
Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

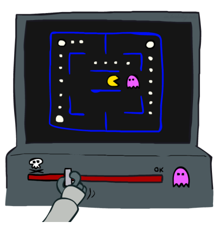
Video of Demo Q-learning – Exploration Function – Crawler

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

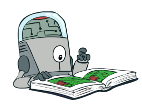


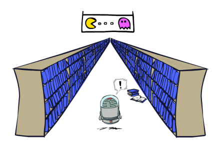
Approximate Q-Learning



Generalizing Across States

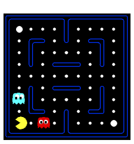
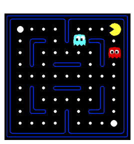
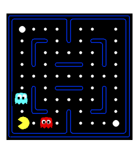
- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training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

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)]

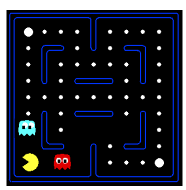
Video of Demo Q-Learning Pacman – Tiny – Watch All

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** (aka "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!

Approximate Q-Learning

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

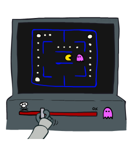
- Q-learning with linear Q-functions:

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

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

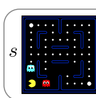


- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features
- Formal justification: online least squares

Example: Q-Pacman

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

s

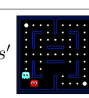


$f_{DOT}(s, \text{NORTH}) = 0.5$

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

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

s'



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

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

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

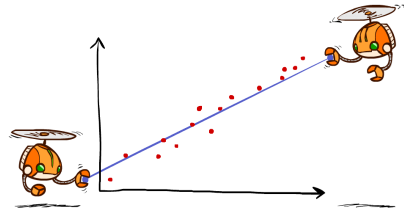
difference $= -501$ $\Rightarrow$ $w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$
$w_{GST} \leftarrow -1.0 + \alpha\,[-501]\,1.0$

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

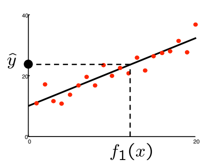
[Demo: approximate Q-learning pacman (L11D10)]

Video of Demo Approximate Q-Learning -- Pacman

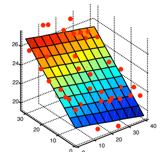
Q-Learning and Least Squares



Linear Approximation: Regression*



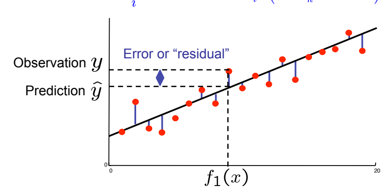
Prediction:
$\hat{y} = w_0 + w_1 f_1(x)$



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

Optimization: Least Squares*

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



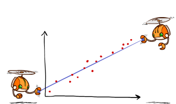
Minimizing Error*

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

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

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

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

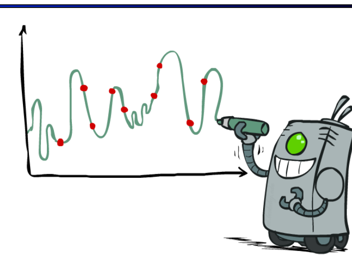


Approximate q update explained:

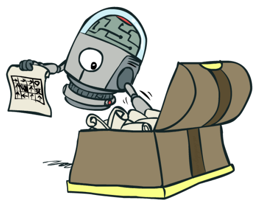
$$w_m \leftarrow w_m + \alpha \left[r + \gamma \max_a Q(s', a') - Q(s, a)\right] f_m(s, a)$$

"target" "prediction"

Overfitting: Why Limiting Capacity Can Help*



Policy Search



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)
- 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

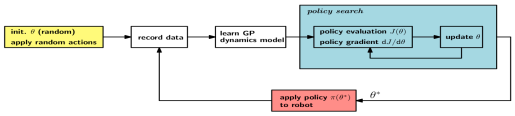
Policy Search

- 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

[Andrew Ng] [Video: HELICOPTER]

PILCO (Probabilistic Inference for Learning Control)



- Model-based policy search to minimize given cost function
- Policy: mapping from state to control
- Rollout: plan using current policy and GP dynamics model
- Policy parameter update via CG/BFGS
- Highly data efficient

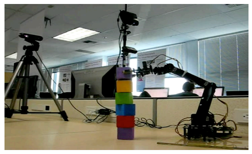
[Deisenroth-etal, ICML-11, RSS-11, ICRA-14, PAMI-14]

Demo: Standard Benchmark Problem

- Swing pendulum up and balance in inverted position
- Learn nonlinear control from scratch
- 4D state space, 300 controller parameters
- **7 trials/17.5 sec experience**
- Control freq.: 10 Hz

Controlling a Low-Cost Robotic Manipulator

- **Low-cost** system ($500 for robot arm and Kinect)
- **Very noisy**
- No sensor information about robot's joint configuration used
- **Goal:** Learn to stack tower of 5 blocks from scratch
- Kinect camera for tracking block in end-effector
- State: coordinates (3D) of block center (from Kinect camera)
- 4 controlled DoF
- 20 learning trials for stacking 5 blocks (5 seconds long each)
- Account for **system noise**, e.g.,
  - Robot arm
  - Image processing



That's all for Reinforcement Learning!

Data (experiences with environment) → Reinforcement Learning Agent → Policy (how to act in the future)

- Very tough problem: How to perform any task well in an unknown, noisy environment!
- Traditionally used mostly for robotics, but becoming more widely used
- Lots of open research areas:
  - How to best balance exploration and exploitation?
  - How to deal with cases where we don't know a good state/feature representation?

Midterm Topics

- Agency: types of agents, types of environments
- Search
  - Formulating a problem in terms of search
  - Algorithms: DFS, BFS, IDS, best-first, uniform-cost, A*, local
  - Heuristics: admissibility, consistency, creation
  - Constraints: formulation, search, forward checking, arc-consistency, structure
  - Adversarial: min/max, alpha-beta, expectimax
- MDPs
  - Formulation, Bellman eqns, V*, Q*, backups, value iteration, policy iteration

33

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!