### 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

### 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: \( Q(s, a) = R(s, a, r_0) + \gamma \max_{a'} Q(s', a') \)
  - Modified Q-Update: \( Q(s, a) = R(s, a, r_0) + \gamma \max_{a'} (Q(s', a') + \beta f(Q(s', a'), n(s', a'))) \)
  - Note: this propagates the "bonus" back to states that lead to unknown states as well!
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. Minimising 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**

- 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 all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalise:
  - Learn about some small number of training states from experience
  - Generalise that experience to new, similar situations
  - This is a fundamental idea in machine learning, and we’ll see it over and over again.

**Generalizing Across States**

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

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

Or even this one?

**Example: Pacman**

Video of Demo Q-Learning Pacman – Tiny – 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
    - Number of ghosts
    - Is Pacman in a tunnel (0/1)
    - …
  - Is it the exact state on this slide?
  - Can also describe a 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:
  \[ V(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s) \]
  \[ Q(s, a) = w_1f_1(s,a) + w_2f_2(s,a) + \ldots + w_nf_n(s,a) \]
  - **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-learning with linear Q-functions:
  - Transition state: \( s \rightarrow s' \)
  - Difference: \[ \text{difference} = Q(s, a) - Q(s, a) \]
  - Linear Q-function: \[ Q(s, a) = w_n + \alpha \text{[difference]} f_n(s, a) \]
  - Intuitive interpretation: adjust weights of active features
    - Example: if something unexpectedly bad happens, blame the features that were on: disprefer all states with those features.
  - Formal justification: online least squares

Example: Q-Pacman

\[ Q(s, a) = 4.0 \text{[Nor]} - 1.0 \text{[Ea]} \]

\[ \text{S'} \rightarrow \text{S'} \quad f_{\text{Nor}}(\text{NORTH}) = 4.1 \quad f_{\text{Ea}}(\text{NORTH}) = 1.0 \]

- \( Q(s, a) = 4.0 \alpha [\text{Nor}] - 1.0 \alpha [\text{Ea}] \)
  - Difference: \(-501\)
  - \[ w_{\text{Nor}} = 4.0 - \alpha [-501] = 4.0 - 0.5 = 3.5 \]
  - \[ w_{\text{Ea}} = 1.0 - \alpha [-501] = 1.0 - 0.5 = 0.5 \]
  - \[ Q(s', a) = 3.0 f_{\text{Nor}}(s', a) - 3.0 f_{\text{Ea}}(s', a) \]
Imagine we had only one point \( x \), with features \( f(x) \), target value \( y \), and weights \( w \):

1. **Error**: 
   \[
   \text{error}(x) = \frac{1}{2} (y - \sum w f(x))^2
   \]

2. **Error Gradient**: 
   \[
   \frac{\partial \text{error}(x)}{\partial w} = -y + \sum w f(x) \cdot f(x)
   \]

3. **Update Rule**: 
   \[
   w \leftarrow w + \alpha \left[ -y + \sum w f(x) \cdot f(x) \right] f(x)
   \]

Approximate Q-update explained:

\[
Q_\alpha \leftarrow Q_\alpha + \alpha \left[ r + \gamma \max_{x'} Q(x', a') - Q(x, a) \right] f(x)
\]

“Target” “prediction"
Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utility) aren't the ones that approximate $V$/$Q$ best
  - e.g. your value functions from project 2 were probably terrible 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

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...

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

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

Andrew Ng

[Andrew Ng]

[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)
- No sensor information about robot’s joint
- Goal: learn to stack tower of 5 blocks from scratch
- Kinect camera for tracking block in end-effector
- Learned policy (AC) of block layout (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!

- 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

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!