Today

- How advanced reinforcement learning works for large problems

- Some previews of fundamental ideas we’ll see throughout the rest of the term

- Next class we’ll start on probabilistic reasoning and reasoning about beliefs
Recap: Q-Learning

- Learn $Q^*(s,a)$ values from samples
  - Receive a sample $(s,a,s',r)$
  - On one hand: old estimate of return: $Q(s,a)$
  - But now we have a new estimate for this sample:
    $$\text{sample} = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$
  - Nudge the old estimate towards the new sample:
    $$Q(s,a) \leftarrow Q(s,a) + \alpha [\text{sample} - Q(s,a)]$$
  - Equivalently, average samples over time:
    $$Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + \alpha [\text{sample}]$$

Q-Learning

- Q-learning produces tables of q-values:
Q-Learning

- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to even 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 states
  - This is a fundamental idea in machine learning, and we’ll see it over and over again

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 its q-states:

- Or even this one!
Feature-Based Representations

- Solution: describe a state using a vector of features
  - 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
    - \( 1 / \text{dist to dot}^2 \)
    - 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 Feature 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 be very different in value!
Function Approximation

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

- Q-learning with linear q-functions:
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha [\text{error}] \]
  \[ w_i \leftarrow w_i + \alpha [\text{error}] f_i(s, a) \]

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

- Formal justification: online least squares (much later)

Example: Q-Pacman

\[ Q(s, a) = 4.0 f_{\text{DOT}}(s, a) - 1.0 f_{\text{GST}}(s, a) \]

- \( f_{\text{DOT}}(s, \text{NORTH}) = 0.5 \)
- \( f_{\text{GST}}(s, \text{NORTH}) = 1.0 \)

- \( Q(s, a) = +1 \)
- \( R(s, a, s') = -500 \)
- \( \text{error} = -499 \)

- \( w_{\text{DOT}} \leftarrow 4.0 + \alpha [-499] 0.5 \)
- \( w_{\text{GST}} \leftarrow 1.0 + \alpha [-499] 1.0 \)

\[ Q(s, a) = 3.0 f_{\text{DOT}}(s, a) - 3.0 f_{\text{GST}}(s, a) \]
Hierarchical Learning

Stratagus: Example of a large RL task, from Bhaskara Marthi’s thesis (w/ Stuart Russell)

- Stratagus is hard for reinforcement learning algorithms
  - > $10^{100}$ states
  - > $10^{30}$ actions at each point
  - Time horizon $\approx 10^4$ steps

- Stratagus is hard for human programmers
  - Typically takes several person-months for game companies to write computer opponent
  - Still, no match for experienced human players
  - Programming involves much trial and error

Hierarchical RL

- Humans supply high-level prior knowledge using partial program
- Learning algorithm fills in the details
Hierarchical RL

- They then define a hierarchical Q-function which learns a linear feature-based mini-Q-function at each choice point.
- Very good at balancing resources and directing rewards to the right region.
- Still not very good at the strategic elements of these kinds of games (i.e. the Markov game aspect).

[DEMO]
Problem: often the feature-based policies that work well aren’t the ones that approximate V / Q best

- E.g. your value functions from 1.3 were probably horrible estimates of future rewards, but they still produce good decisions
- We’ll see this distinction between modeling and prediction again later in the course

Solution: learn the policy that maximizes rewards rather than the value that predicts rewards

This is the idea behind policy search, such as what controlled the upside-down helicopter
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

---

Policy Search*

- **Advanced policy search:**
  - Write a stochastic (soft) policy:

\[
\pi_w(s) \propto e^{\sum_i w_i f_i(s,a)}
\]

  - Turns out you can efficiently approximate the derivative of the returns with respect to the parameters w (details in the book, but you don’t have to know them)

  - Take uphill steps, recalculate derivatives, etc.
Take a Deep Breath…

- We’re done with search and planning!

- Next, we’ll look at how to reason with probabilities
  - Diagnosis
  - Tracking objects
  - Speech recognition
  - Robot mapping
  - … lots more!

- Last part of course: machine learning