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:
    \[
    Q(s,a) \leftarrow Q(s,a) + \alpha \left[ \text{sample} - Q(s,a) \right]
    \]
  - 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:

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 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/(dist to dot)²
  - Is Pacman in a tunnel? (0/1)
- 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

\[ Q(s, a) = 4.0 f_{DOR}(s, a) - 1.0 f_{GSTR}(s, a) \]

- \( f_{DOR}(s, \text{NORTH}) = 0.5 \)
- \( f_{GSTR}(s, \text{NORTH}) = 1.0 \)
- \( Q(s, a) = 1 \)
- \( R(s, a, s') = -50 \)
- \( \text{error} = -495 \)
- \( w_{DOR} \leftarrow 4.0 + \alpha [-495] 0.5 \)
- \( w_{GSTR} \leftarrow 1.0 + \alpha [-495] 1.0 \)

\[ Q(s, a) = 3.0 f_{DOR}(s, a) - 3.0 f_{GSTR}(s, a) \]

Hierarchical Learning

Hierarchical RL

- Stratagus: Example of a large RL task, from Bhaskara Marthi’s thesis (w/ Stuart Russell)
- Stratagus is hard for reinforcement learning algorithms
  - \( > 10^{10} \) states
  - \( > 10^{10} \) actions at each point
  - Time horizon \( = 10^4 \) 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
Partial "Alisp" Program

(defun top ()
  (loop
    (choose
      (gather-wood)
      (gather-gold))))

(defun gather-wood ()
  (with-choice
    (dest *forest-list*)
    (nav dest)
    (action 'get-wood)
    (nav *base-loc*)
    (action 'dropoff)))

(defun gather-gold ()
  (with-choice
    (dest *goldmine-list*)
    (nav dest)
    (action 'get-gold)
    (nav *base-loc*)
    (action 'dropoff)))

(defun nav (dest)
  (until (= (pos (get-state))
    dest)
    (with-choice
      (move '(N S E W NOOP))
      (action move))))

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]

Policy Search

- 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:
    \[ \pi_w(s) \propto e^{\sum \omega_i f_i(s,a)} \]
  - Turns out you can efficiently approximate the derivative of the returns with respect to the parameters \(\omega\) (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