Reinforcement Learning

- Reinforcement learning:
  - 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 policy \( \pi(s) \)
  - New twist: don’t know \( T \) or \( R \)
    - I.e., don’t know which states are good or what the actions do
    - Must actually try actions and states out to learn

The Story So Far: MDPs and RL

Things we know how to do:

- If we know the MDP
  - Compute \( V^*, Q^*, \pi^* \) exactly
- If we don’t know the MDP
  - We can estimate the MDP then solve
  - We can estimate \( V \) for a fixed policy \( \pi \)
  - We can estimate \( Q^*(s, a) \) for the optimal policy while executing an exploration policy

Techniques:

- Model-based DPs
  - Value and policy iteration
  - Policy evaluation
- Model-based RL
- Model-free RL:
  - Value learning
  - Q-learning

Model-Free Learning

- Model-free (temporal difference) learning
  - Experience world through episodes \((s, a, r, s', r', s'', r', s''', r', s'''', ...)\)
  - Update estimates each transition \((s, a, r, s')\)
  - Over time, updates will mimic Bellman updates
- Q-Value iteration (model-based, requires known MDP)
  \[ Q_{t+1}(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_t(s', a') \right] \]
- Q-Learning (model-free, requires only experienced transitions)
  \[ Q(s, a) = (1 - \alpha) Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') \right] \]

Q-Learning

- We’d like to do Q-value updates to each Q-state:
  \[ Q_{t+1}(s, a) \rightarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_t(s', a') \right] \]
- 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
    \[ Q(s, a) \approx r + \gamma \max_{a'} Q(s', a') \]
    (Why?)
  - So keep a running average
    \[ Q(s, a) = (1 - \alpha) Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') \right] \]

Q-Learning Properties

- Will converge to optimal policy
  - If you explore enough (i.e., visit each q-state many times)
  - If you make the learning rate small enough
    - Basically doesn’t matter how you select actions (!)
- Off-policy learning: learns optimal q-values, not the values of the policy you are following
Q-Learning

- Q-learning produces tables of q-values:

![Q-values table]

Exploration / Exploitation

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With probability ε, act randomly
    - With probability 1-ε, act according to current policy
  - Regret: expected gap between rewards during learning and rewards from optimal action
    - Q-learning with random actions will converge to optimal values, but possibly very slowly, and will get low rewards on the way
    - Results will be optimal but regret will be large
    - How to make regret small?

Exploration Functions

- When to explore
  - Random actions: explore a fixed amount
  - Better ideas: explore areas whose badness is not (yet) established, explore less over time
  - One way: exploration function
    - Takes a value estimate and a count, and returns an optimistic utility, e.g. \( f(u, n) = u + k/n \) (exact form not important)

\[
Q_{t+1}(s, a) = R(s, a, s') + \gamma \max_{a'} Q_t(s', a') \\
Q_{t+1}(s, a) = R(s, a, s') + \gamma \max_{a'} f(Q_t(s', a'), N(s', a'))
\]

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 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 (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)?
    - 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 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 actually be very different in value!

### Function Approximation

- Q-learning with linear q-functions:
  \[ Q(s, a) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s, a) \]

- Q-learning with linear q-functions:
  \[ Q(s, a) = w_1 f_1(s) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- Exact Q’s

- Approximate Q’s

### Example: Q-Pacman

- Q(s, a) = 4.0f_{DOT}(s, a) − 1.0f_{GHOST}(s, a)
- f_{DOT}(s, NORTH) = 0.5
- f_{GHOST}(s, NORTH) = 1.0
- Q(s, a) = +1
- R(s, a, s') = −500
- difference \(\approx -501\)
- \(w_{DOT} \leftarrow 4.0 + \alpha [-501] 0.5\)
- \(w_{GHOST} \leftarrow -1.0 + \alpha [-501] 1.0\)
- Q(s, a) = 3.0f_{DOT}(s, a) − 3.0f_{GHOST}(s, a)

### Linear Regression

- Prediction
  \[ \hat{y} = w_0 + w_1 f_1(x) \]

- Minimizing Error

- Imagine we had only one point \(x\) with features \(f(x)\):
  \[ \text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2 \]
  \[ \frac{\partial \text{error}(w)}{\partial w_m} = -\left( y - \sum_k w_k f_k(x) \right) f_m(x) \]
  \[ w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a) - Q(s, a) \right] f_m(x) \]
Problem: often the feature-based policies that work well 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
- 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

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

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 \( w \) (optional material)
  - Take uphill steps, recalculate derivatives, etc.

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