Reinforcement Learning

- Reinforcement learning:
  - Still have 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

Model-Free Learning

- Temporal difference learning
  - Update each time we experience a transition
  - Frequent outcomes will contribute more updates (over time)

\[
V^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')] \\
\text{sample} = R(s, \pi(s), s') + \gamma V^\pi(s') \\
V^\pi(s) \leftarrow (1 - \alpha) V^\pi(s) + \alpha \text{sample}
\]

Q-Learning

- Learn \( Q^*(s,a) \) values
  - Receive a sample \( (s,a,s',r) \)
  - Consider your old estimate: \( Q(s,a) \)
  - Consider your new sample estimate:
    \[
    Q'(s,a) = \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max_{a'} Q'(s',a') \right] \\
    \text{sample} = R(s,a,s') + \gamma \max_{a'} Q'(s',a')
    \]
  - Incorporate the new estimate into a running average:
    \[
    Q(s,a) \leftarrow (1 - \alpha) Q(s,a) + \alpha [\text{sample}]
    \]

Q-Learning Properties

- Will converge to optimal policy
  - If you explore enough
  - If you make the learning rate small enough
  - But not decrease it too quickly!
  - Basically doesn’t matter how you select actions (!)
- Neat property: learns optimal q-values regardless of action selection noise (some caveats)

Exploration / Exploitation

- Several schemes for forcing exploration
  - Simplest: random actions (\( \epsilon \)-greedy)
    - Every time step, flip a coin
    - With probability \( \epsilon \), act randomly
    - With probability \( 1-\epsilon \), act according to current policy
  - Problems with random actions?
    - You do explore the space, but keep thrashing around once learning is done
    - One solution: lower \( \epsilon \) 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

- **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**

- Q-learning produces tables of q-values:

![Q-values after load expansion](image)

**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
  - 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.
  - 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:
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{[correction]} \]
  \[ f_i(s, a) \rightarrow f_i(s, a) + \alpha \text{[correction]} \]

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

Example: Q-Pacman

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

Linear Regression

Given examples \((x_i, y_i)_{i=1}^n\)
Predict \(y_{n+1}\) given a new point \(x_{n+1}\)

Ordinary Least Squares (OLS)

\[ E(w) = \frac{1}{2} \sum_i \left( \sum_k f_k(x_i) w_k - y_i \right)^2 \]
\[ \frac{\partial E}{\partial w_m} = \sum_i \left( \sum_k f_k(x_i) w_k - y_i \right) f_m(x_i) \]
\[ w_m \leftarrow w_m - \alpha \sum_i \left( \sum_k f_k(x_i) w_k - y_i \right) f_m(x_i) \]

Approximate q update explained:
\[ w_m \leftarrow w_m - \alpha \text{[error]} f_m(s, a) \]
\[ w_m \leftarrow w_m + \alpha \text{[correction]} f_m(s, a) \]
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:
  \[
  \pi_w(s) \propto e^{\sum w_i f_i(s,a)}
  \]
- Turns out you can efficiently approximate the derivative of the returns with respect to the parameters \( w \) (details in the book, optional material)
- 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