Reinforcement Learning II

Steve Tanimoto

[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

Reinforcement Learning

- We still assume an MDP:
  - A set of states \( S \)
  - 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 \( R \), so must try out actions
- Big idea: Compute all averages over \( T \) using sample outcomes

The Story So Far: MDPs and RL

Known MDP: Offline Solution

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute ( V^<em>, Q^</em>, \pi^* )</td>
<td>Value / policy iteration</td>
</tr>
<tr>
<td>Evaluate a fixed policy ( \pi )</td>
<td>Policy evaluation</td>
</tr>
</tbody>
</table>

Unknown MDP: Model-Based

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute ( V^<em>, Q^</em>, \pi^* )</td>
<td>VI/PI on approx. MDP</td>
</tr>
<tr>
<td>Evaluate a fixed policy ( \pi )</td>
<td>PE on approx. MDP</td>
</tr>
</tbody>
</table>

Unknown MDP: Model-Free

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute ( V^<em>, Q^</em>, \pi^* )</td>
<td>Q-learning</td>
</tr>
<tr>
<td>Evaluate a fixed policy ( \pi )</td>
<td>Value Learning</td>
</tr>
</tbody>
</table>

Q-Learning

- We’d like to do Q-value updates to each Q-state:
  \[
  Q_{t+1}(s,a) = \sum \limits_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max \limits_{a'} Q_t(s',a') \right]
  \]
  - 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
      \[
      Q(s,a) \leftarrow r + \gamma \max \limits_{a'} Q(s',a')
      \]
    - But we want to average over results from \( (s,a) \) [Why?] 
    - So keep a running average
      \[
      Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + \alpha \left[ r + \gamma \max \limits_{a'} Q(s',a') \right]
      \]

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy – even if you’re acting suboptimally!
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - ... but do not decrease it too quickly
  - Basically, in the limit, it doesn’t matter how you select actions [1]
How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions ($\epsilon$-greedy)
    - Every time step, flip a coin
    - With (small) probability $\epsilon$, act randomly
    - With (large) probability $1-\epsilon$, 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 $\epsilon$ 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, 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: $Q(s, a) \leftarrow Q(s, a) + \gamma \max_a' Q(s', a')$
  - Modified Q-Update: $Q(s, a) \leftarrow Q(s, a) + \gamma \max_a' f(Q(s', a'), N(s', a'))$
  - 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.

Minimizing 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 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 situations

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 even this one!

Video of Demo Q-learning Pacman – Tiny – Watch All
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)^2
  - 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 Value Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:
  \[ V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_N f_N(s) \]
  \[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_N f_N(s, a) \]
- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but actually be very different in value!

Approximate Q-Learning

- Q-learning with linear Q-functions:
  \[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_N f_N(s, a) \]
- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features
- Formal justification: online least squares

Example: Q-Pacman

- Exact Q's
- Approximate Q's

\[ Q(s, a) = 4.0 f_{DOF}(s, a) - 1.0 f_{GOST}(s, a) \]
Video of Demo Approximate Q-Learning -- Pacman

Q-Learning and Least Squares

Linear Approximation: Regression*

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

Optimization: Least Squares*

\[
\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i (y_i - \sum_k w_k f_k(x_i))^2
\]

Minimizing Error*

Imagine we had only one point \( x \), with features \( f(x) \), target value \( y \), and weights \( w \):

\[
\text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2
\]

\[
\frac{\partial \text{error}(w)}{\partial w_k} = -\left( y - \sum_k w_k f_k(x) \right) f_k(x)
\]

\[
w_k = w_k + \alpha \left( y - \sum_k w_k f_k(x) \right) f_k(x)
\]

Approximate q update explained:

\[
w_{sa} \leftarrow w_{sa} + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] f_{s, a}
\]

Overfitting: Why Limiting Capacity Can Help*

"target"  "prediction"
Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utilities) 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
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
  - We'll see this distinction between modeling and prediction again later in the course
- 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...

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!