Today’s Topics

- Exploration vs. Exploitation
- Generalizing Across State
- SARSA
- TD Backups

- HW3 Due FRIDAY (4pm)
- HW4 out this week (‘easy’ paper-and-pencil HW)
- ‘Unofficial’ class this Friday (the Agent World)
Q’s vs. U’s

- Assume we’re in state S
  Which action do we choose?

- U’s (Model-based)
  - Need to have a ‘next state’ function to generate all possible states (ie, chess)
  - Choose next state with highest $U$ value.

- Q’s (Model-free, though can also do model-based Q learning)
  - Need only know which actions are legal (ie, web)
  - Choose next state with highest $Q$ value.
Exploration vs. Exploitation

In order to learn about better alternatives, we can’t always follow the current policy (‘exploitation’)

Sometimes, need to try random moves (‘exploration’)

© Jude Shavlik 2010  CS 760 – Machine Learning (UW-Madison)  Lecture #25, Slide 3
Exploration vs. Exploitation (cont)

Approaches

1) $p$ percent of the time, make a random move; could let

$$p = \frac{1}{\sqrt{\text{#moves made}}}$$

2) Prob(picking action $A$ in state $S$)

$$\text{Prob}(A) = \frac{\text{const}^Q_{S,A}}{\sum_{i \in \text{actions}} \text{const}^Q_{S,i}}$$

Exponentiating gets rid of negative values.
One-Step Q-Learning Algo

0. $S \leftarrow$ initial state

1. If random $\# \leq P$
   
   then $a = \text{random choice}$

   Else $a = \Pi_1(S)$

2. $S_{\text{new}} \leftarrow W(S, a)$
   
   $R_{\text{immed}} \leftarrow R(S_{\text{new}})$

   \{ Act on world and get reward \}

3. $\text{Error} \leftarrow R_{\text{immed}} + \gamma U(S_{\text{new}}) - Q(S, a)$

4. $Q(S, a) \leftarrow Q(S, a) + \alpha \text{Error}$  \hspace{1cm} // Should also decay $\alpha$

5. $S \leftarrow S_{\text{new}}$

6. Go to 1
Q-Learning: Implementation Details

Remember, conceptually we are filling in a huge table.

<table>
<thead>
<tr>
<th>Actions</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>...</th>
<th>Sn</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td>Q(S2, c)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tables are a very verbose representation of a function.
Representing Q Functions More Compactly

We can use some other function representation (e.g., neural net) to compactly encode this big table.

An encoding of the state (S)

Each input unit encodes a property of the state (e.g., a sensor value).

Second argument is a constant. Or could have one net for each possible action.
Q Tables vs Q Nets

Given: 100 Boolean-valued features

10 possible actions

Size of Q table

\[10 \times 2^{100} \implies \text{# of possible states}\]

Size of Q net (100 HU’s)

\[100 \times 100 + 100 \times 10 = 11,000\]

Weights between inputs and HU’s

Weights between HU’s and outputs
Why Use a Compact Q-Function?

1. Full Q table may not fit in memory for realistic problems.
2. Can generalize across states, thereby speeding up convergence.
   ie, one example ‘fills’ many cells in the Q table.

Notes

1. When generalizing across states, cannot use $\alpha=1$.
2. Convergence proofs only apply to Q tables.
3. Some work on bounding errors caused by using compact representations (eg, Singh & Lee, MLj).
Connectionist Q-Learning (1-step Lookahead)

Algorithm for training the ‘Q-net’

1. Measure sensors \(S_0\)
2. Predict \(Q(S_0, \alpha)\) for each action \(\alpha\)
   
   Let action A be
   
   i. The best action
      
      [ie, \(Q(S_0, A)\) is highest \(Q(S_0, \alpha)\)]
   
   ii. A randomly chosen action
3. Apply action A in the world,
   
   Re-measure sensors \(S_1\)
   
   and immediate reward \(R\)
Connectionist Q-Learning (cont)

4. Find action $\beta$ that maximizes $Q(S_1, \beta)$
   ie compute $u(S_1)$ but *don’t apply* $\beta$

5. Train with
   
   input = $S_0$
   
   output = $Q(S_0, A) + \alpha[R + \gamma Q(S_1, \beta) - Q(S_0, A)]$

   * Only train output corresponding to action $A$

   We create a standard I/O pair!

Typically, discard I/O pair after *one* backprop
Two Forward Props and a BackProp

1. \( S_0 \)
   
   \[
   Q(S_0, A) \quad \Rightarrow \quad Q(S_0, Z)
   \]
   
   Choose action in state \( S_0 \)

2. \( S_1 \)
   
   \[
   Q(S_1, A) \quad \Rightarrow \quad Q(S_1, Z)
   \]
   
   Estimate \( U(S_1) = \max Q(S_1, X) \) where \( X \in \text{actions} \)

3. \( S_0 \)
   
   \[
   Q(S_0, A) = \text{new estimate} \quad \Rightarrow \quad Q(S_0, Z) = \text{‘correct’ as is for other actions}
   \]

   Backprop to reduce error at \( Q(S_0, A) \)

Arguably this is THREE forward props, but could cache result of 1 for use in 3
**k-NN for Q-Learning**

Represent $Q(s, a)$ using $k$-NN

**One Possible Design**

For *each* action, keep set of I/O pairs

Data Point

- **input** = $S_0$
- **output** = $Q(S_0, A) + \alpha [ R + \gamma Q(S_1, B) - Q(S_0, A) ]$

Measure similarity to previous states, average the outputs of neighbors to predict $Q(s, a)$

(Might want to discard old examples since Q’s out of date)

Could also use (relational) regression trees, support-vector regression, etc
SARSA vs. Q-Learning

(1994, 1996)

(1989)

Exploring can be hazardous!
Should we learn to consider its impact?

The Cliff-Walking Task (pg 150 of Sutton + Barto RL Text)

Safe route

Optimal path
-if no exploration!

What would Q-Learning learn?

R = -1 for all of these

R = 0
SARSA = State Action Reward State Action

**SARSA**

\[ Q(s,a) \leftarrow Q(s,a) + \alpha [ R + \gamma Q(s', a') - Q(s,a) ] \]

**Standard Q-Learning**

\[ Q(s,a) \leftarrow Q(s,a) + \alpha [ R + \gamma \max Q(s', a'') - Q(s,a) ] \]

SARSA uses actual next action (still chosen via explore-exploit strategy, e.g. soft-max or “coin-flip”)

- SARSA also converges

Notice that in Q learning, (currently) non-optimal moves do not impact the Q function
Sample Results: Cliff-Walking Task

SARSA learns to avoid ‘pitfalls’ while in exploration phase (always need to explore if in a real-world situation?)

Episodes (ie from Start to Goal)
(prob of random move = 0.1)
'Backups' (from Sutton & Barto)

Key:
- Non-deterministic action
- Possible next states
- 'Terminal state' (end of game)

U(s) ← ?

Backups

Time

Backing Up Info

© Jude Shavlik 2010
CS 760 – Machine Learning (UW-Madison)
Lecture #25, Slide 17
Q-Learning Backup

\[ Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left[ r + \gamma \max_b Q(s',b) \right] \]

Does not need an \textbf{world model} (the ‘next state’ and ‘reward’ functions)
Simple Monte Carlo

\[ U(s) \leftarrow (1-\alpha) U(s) + \alpha \text{Reward(path)} \]

Stochastically run to end of ‘game’ many times

'S in place' averaging

No world model needed

Relatively efficient on very large problems
Stochastic Dynamic Programming

Expected value

\[ U(s) \leftarrow \max_a E [r + \gamma U(s' \text{ given action } a)] \]

Needs a **world model** to compute true expected value

A very informative backup
Complex TD Backups
(TD = Temporal Difference; Sutton, MLj, ’88)

\[ \text{TD}(\lambda) \]

Incrementally computes a weighted mixture of these backups as states are visited

\[ \lambda = 1 \rightarrow \text{Simple Monte Carlo} \]
\[ \lambda = 0 \rightarrow 1\text{-Step Q-Learning} \]
The Sum of the TD(\(\lambda\)) Weighting Terms

\[
\sum_{i=0}^{N-1} (1- \lambda) \times \lambda^i + \lambda^N
\]

\[
= (1- \lambda) \times \left[ \sum_{i=0}^{N-1} \lambda^i \right] + \lambda^N
\]

\[
= (1 - \lambda) \times \left[ \frac{(1 - \lambda^N)}{(1 - \lambda)} \right] + \lambda^N
\]

see [http://www.math.com/tables/expansion/geom.htm](http://www.math.com/tables/expansion/geom.htm)

\[
= (1 - \lambda^N) + \lambda^N
\]

\[
= 1
\]