Reinforcement Learning
Part 2

Yingyu Liang
yliang@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison

[Based on slides from David Page, Mark Craven]
Goals for the lecture

you should understand the following concepts

• value functions and value iteration (review)
• Q functions and Q learning
• exploration vs. exploitation tradeoff
• compact representations of Q functions
Value function for a policy

• given a policy $\pi : S \rightarrow A$ define

$$V^\pi(s) = \sum_{t=0}^{\infty} \gamma^t E[r_t]$$

assuming action sequence chosen according to $\pi$ starting at state $s$.

• we want the optimal policy $\pi^*$ where

$$*= \arg\max V(s) \text{ for all } s$$

we’ll denote the value function for this optimal policy as $V^*(s)$.
Value iteration for learning $V^*(s)$

initialize $V(s)$ arbitrarily
loop until policy good enough
{
    loop for $s \in S$
    {
        loop for $a \in A$
        {
            $Q(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in S} P(s'| s, a)V(s')$
        }
        $V(s) \leftarrow \max_a Q(s, a)$
    }
}
Q functions

define a new function, closely related to $V^*$

$$Q(s, a) \leftarrow E[r(s, a)] + \gamma E_{s'|s, a}[V^*(s')]$$

if agent knows $Q(s, a)$, it can choose optimal action without knowing $P(s' | s, a)$

$$\pi^*(s) \leftarrow \arg \max_a Q(s, a) \quad V^*(s) \leftarrow \max_a Q(s, a)$$

and it can learn $Q(s, a)$ without knowing $P(s' | s, a)$
$Q$ values

$r(s, a)$ (immediate reward) values

$V^*(s)$ values

$Q(s, a)$ values
Q learning for deterministic worlds

for each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$

observe current state $s$
do forever
  select an action $a$ and execute it
  receive immediate reward $r$
  observe the new state $s'$
  update table entry
    $\hat{Q}(s, a) \leftarrow r + \gamma \max_a \hat{Q}(s', a')$
  $s \leftarrow s'$
Updating $Q$

\[ \hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \]

\[ \leftarrow 0 + 0.9 \max\{63, 81, 100\} \]

\[ \leftarrow 90 \]
Which action do we choose when we’re in a given state?

- **$V$’s (model-based)**
  - need to have a ‘next state’ function to generate all possible states
  - choose next state with highest $V$ value.

- **$Q$’s (model-free)**
  - need only know which actions are legal
  - generally choose next state with highest $Q$ value.
Exploration vs. Exploitation

• in order to learn about better alternatives, we shouldn’t always follow the current policy (exploitation)

• sometimes, we should select random actions (exploration)

• one way to do this: select actions probabilistically according to:

\[ P(a_i \mid s) = \frac{c \hat{Q}(s,a_i)}{\sum_j c \hat{Q}(s,a_j)} \]

where \( c > 0 \) is a constant that determines how strongly selection favors actions with higher \( Q \) values
As described so far, Q learning entails filling in a huge table. A table is a very verbose way to represent a function.
Representing $Q$ functions more compactly

We can use some other function representation (e.g. a neural net) to **compactly** encode a substitute for the big table.

Each input unit encodes a property of the state (e.g., a sensor value) or could have **one net** for each possible action.

```
  encoding of the state (s)

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

```
  or could have one net for each possible action
```
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
   i.e. one instance ‘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  (e.g. Singh & Yee, *Machine Learning* 1994)
Given: 100 Boolean-valued features
10 possible actions

Size of Q table
$10 \times 2^{100}$ entries

Size of Q net (assume 100 hidden units)
$100 \times 100 + 100 \times 10 = 11,000$ weights
weights between inputs and HU’s
weights between HU’s and outputs
Representing $Q$ functions more compactly

- we can use other regression methods to represent $Q$ functions
  - $k$-NN
  - regression trees
  - support vector regression
  - etc.
$Q$ learning with function approximation

1. measure sensors, sense state $s_0$
2. predict $\hat{Q}_n(s_0, a)$ for each action $a$
3. select action $a$ to take (with randomization to ensure exploration)
4. apply action $a$ in the real world
5. sense new state $s_1$ and immediate reward $r$
6. calculate action $a'$ that maximizes $\hat{Q}_n(s_1, a')$
7. train with new instance

$$x = s_0$$

$$y \leftarrow (1 - \alpha)\hat{Q}(s_0, a) + \alpha [r + \gamma \max_{a'} \hat{Q}(s_1, a')]$$

Calculate $Q$-value you would have put into $Q$-table, and use it as the training label