Markov Decision Processes and Reinforcement Learning

An Introduction to Stochastic Planning
Path Planning Assumptions

- Obstacles?
  - Reliable Collision detection (assumes robust perception)

- Transitions
  - Reliable mechanism for moving along path in graph (i.e., a controller)

```
move_block(x1,y1,x2,y2)
```
Two Sources of Error

- **State Estimation**
  - You don’t know exactly where you are
  - Sensors have noise
  - No complete environment information

- **Action Execution**
  - Your actuators don’t do what you tell them
  - Your system responds differently than you expect
  - Friction, gears, air resistance, etc.

**Basic Idea:** Your model of the world is incorrect!
Markov (Decision) Processes: A New Model for Planning

- Handles both forms of uncertainty in a statistically principled way
- Gives us back optimality!
- Of course, I’m talking about (PO)MDPs
  - All this flexibility comes at a cost, as we’ll see...
  - Current research is largely about scalability
• **Problem**: we don’t know where our actions take us

• **Solution**: start thinking about *expected values*

  Weight each outcome by the *probability* of getting there
Formalizing the MDP Model

• **Step 1**: define the core problem representation

• Considerations?

  1. should represent “rewards” somehow
  2. should represent “state” somehow
  3. should represent “actions” somehow

  ➡ next: what if actions aren’t deterministic??
Formalizing the MDP Model

• Step 2: How to handle stochastic action effects (“transitions”)?

• replace transition rule with transition distribution

\[
T(s, s') = P(s'|s) = 
\begin{bmatrix}
P_{11} & P_{12} & \ldots & P_{1n} \\
P_{21} & P_{22} & \ldots & P_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
P_{n1} & P_{n2} & \ldots & P_{nn}
\end{bmatrix}
\]
### Formalizing the MDP Model

- **Overall:**
  \[ MDP = \{ S, A, T, R, \gamma \} \]

- **States** \( S \)
- **Actions** \( A \)
- **Transition Model** \( T \)
- **Rewards** \( R \)

---

**Pacman states**
- \{all positions of pacman, ghosts, food, & pellets\}

**Pacman actions**
- \{N,S,E,W\}

**Pacman model**
- \{move directions, die from ghosts, eat food,...\}

**Pacman rewards**
- -1 per step, +10 food, -500 die, +500 win,...
Markov Processes: Caveman’s World

States: \{H, G, F, D\}

Actions: {} (we’ll get back to this)

Transition Model:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.5</td>
<td>0.4</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>G</td>
<td>0.2</td>
<td>0.1</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>F</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>D</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Rewards:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
</tbody>
</table>

just a CPT
Markov Processes: Caveman’s World

States: \{H, G, F, D\}

Actions: \{
\}

Transition Model:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.5</td>
<td>0.4</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>G</td>
<td>0.2</td>
<td>0.1</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>F</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>D</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Rewards:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
</tbody>
</table>
**Markov Processes: Caveman’s World**

---

### Value

- How good is it to be in a state?
- Sum of discounted expected rewards:

\[
V(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t r_t \right]
\]

- Reward now is better than later. Why??
Value Iteration in Caveman’s World

- Key idea: Bellman Recursion
- Relates value in current state to expected value of next state

\[ V(s) = R(s) + \gamma \sum_{s'} P(s'|s)V(s') \]
Value Iteration in Caveman’s World

- Key idea: Bellman Recursion
- Relates value in current state to expected value of next state

\[ V(s = H) = r + \gamma (P_{HH}(R_H) + P_{HG}(R_G) + P_{HF}(R_F) + P_{HD}(R_D)) \]
\[ = 0 + 0.9(0.5(0) + 0.4(1) + 0.1(10)) \]
Value Iteration in Caveman’s World

\[ V(s = H) = r + \gamma(P_{HH}(R_H) + P_{HG}(R_G) + P_{HF}(R_F) + P_{HD}(R_D)) \]
\[ = 0 + 0.9(0.5(0) + 0.4(1) + 0.1(10)) \]

Value in k-steps

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
<tr>
<td>2</td>
<td>(-0.54)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hungry

0

Got Food

+1

Full

+10

Dead

-10

.2

.1

.4

.6

.5

.1

.1

.9

1

Wednesday, July 10, 13
Value Iteration in Caveman’s World

Value in $k$-steps

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
<tr>
<td>2</td>
<td>-.54</td>
<td>5.69</td>
<td>9.1</td>
<td>-19</td>
</tr>
<tr>
<td>3</td>
<td>.06</td>
<td>4.61</td>
<td>7.85</td>
<td>-27.1</td>
</tr>
<tr>
<td>4</td>
<td>-.75</td>
<td>3.23</td>
<td>7.61</td>
<td>-34.39</td>
</tr>
<tr>
<td>99</td>
<td>-39.08</td>
<td>-34.71</td>
<td>-30.66</td>
<td>-100.0</td>
</tr>
<tr>
<td>100</td>
<td>-39.09</td>
<td>-34.71</td>
<td>-30.66</td>
<td>-100.0</td>
</tr>
</tbody>
</table>

Value Iteration is Guaranteed to Converge
Summary

- Markov Processes represent **uncertainty in state transitions**
- It is possible to determine the **overall value of a state**
- What’s next? Adding actions!
Actions: the value of free-will

- What’d we do so far?
  - Define values of states, and transition probabilities between them

- To add actions, what do we need to look at?
  1. condition on actions: $P(s'|s) \rightarrow P(s'|s,a)$
  2. values of actions: $V(s) = \max_a Q(s,a)$

- Turns out we need only (1), and (2) is RL
• Adding actions back into an MDP:

• How? Make transitions conditional on action

\[ T(s, a, s') = P(s'|s, a) = \begin{bmatrix} P_{11}^a & P_{12}^a & \cdots & P_{1n}^a \\ P_{21}^a & P_{22}^a & \cdots & P_{2n}^a \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1}^a & P_{n2}^a & \cdots & P_{nn}^a \end{bmatrix} \]
Actions: the value of free-will

Value-Iteration needs one more thing:

$$V(s) \leftarrow \max_a \left[ R(s, a) + \gamma \sum_{s'} P(s' | s, a) V(s') \right]$$

*Bellman Equation*

added this max over actions
Actions: the value of free-will

“Free-Will” Values:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
<tr>
<td>2</td>
<td>-.09</td>
<td>8.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Actions: the value of free-will

“Free-Will” Values:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>G</th>
<th>F</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>-10</td>
</tr>
<tr>
<td>2</td>
<td>-.09</td>
<td>8.2</td>
<td>10</td>
<td>-19</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td></td>
<td>3.56</td>
<td>-100</td>
</tr>
</tbody>
</table>

Markov Processes represent uncertainty in state transitions. It is possible to determine the overall future value of a state. Matrix inversion is closed form but expensive. Value iteration is simple and effective.
Value Iteration in Code

initialize $V(s)$ arbitrarily
loop until policy good enough
  loop for $s \in S$
    loop for $a \in A$
      $Q(s, a) := R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s')$
      $V(s) := \max_a Q(s, a)$
    end loop
  end loop
end loop

What’s this “$Q$” function?
⇒ Topic for later, but short answer is to allow action selection without lookahead
Things to really understand about MDPs:

- what a value function is
- why we can converge to $V^*$ with these simple algorithms
- why $V^*$ is overkill sometimes
- why model is so important, and what to do without it
- why these algorithms can be (horribly) inefficient
Value Iteration: Big Questions

- Convergence?
- Efficiency?
- Assumptions?
Value Iteration Convergence

- **Proof Sketch:**

1. Defined in terms of max-norm between any two value functions (in particular $V_i$ and $V^*$)

2. Take advantage of basic property of max:
   \[ |\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)| \]

3. Apply Bellman operator and rearrange
   \[
   |B(V_i) - B(V_j)|_V(s) = \left| \left( R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s') \right) - \left( R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_j(s') \right) \right|
   \]
   \[
   = \gamma (\max_a E_{V_i}[s'] - \max_a E_{V_j}[s'])
   \]
   \[
   \leq \gamma \max_a (E_{V_i}[s'] - E_{V_j}[s'])
   \]
   \[
   \leq \gamma \max_a (V_i(s') - V_j(s'))
   \]

**tl;dr:** max-norm (max difference w.r.t. $V^*$) strictly contracts with each application of Bellman (with factor gamma)
But how important is convergence?

- Why does value matter again? To pick actions
  - IE, we’re interested in $\pi(s)$, not $V(s)$
- Can we optimize the policy directly?
  - Yes! This is “policy iteration”
  - We’ll use the policy form of Bellman:
    \[
    V_{t+1}^\pi(s) \leftarrow R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s))V_t^\pi(s')
    \]
Policy Iteration

- Alternative approach:
  - Step 1: Policy evaluation: calculate value for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using onestep look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges (it does)

- Facts about policy iteration:
  - It’s still optimal!
  - Can converge faster under some conditions. Why??
Implementing Policy Iteration

- Simple change:
  
  1. Evaluate policy somehow
     
     - option 1: solve as linear system
     - option 2: use Bellman for a while

     \[
     V_0^{\pi}(s) \leftarrow 0 \\
     V_{t+1}^{\pi}(s) \leftarrow R(s, \pi_t(s)) + \gamma \sum_{s'} P(s'|s, \pi_t(s)) V_t^{\pi}(s')
     \]

  2. Improve policy using 1-step lookahead

     \[
     \pi_{k+1}^*(s) = \arg \max_a \left[ R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_1^{\pi_k}(s') \right]
     \]
Policy iteration convergence proof sketch:

(1) In every step the policy improves. Means that a given policy can be encountered at most once. This means that after we have iterated as many times as there are different policies (i.e., $|A|^{|S|}$), we must be done and hence have converged.

(2) By definition at convergence we have that $\pi_{k+1}(s) = \pi_k(s)$ $\forall s \in S$. This implies that $V^{\pi_k} = \max_a [R(s, a) + \gamma \sum_{s'} P(s'|s, a)V^{\pi_k}(s')]$ for all states. This satisfies the Bellman equation, which means $V^{\pi_k}$ is equal to the optimal value function $V^*$. 
• What’s the real difference vs. VI?
  • Just puts more effort into policy evaluations in between policy updates

• Why might this be helpful??
  ➡ Early convergence criterion (policy stops changing)
  ➡ When we have lots of actions, so update is expensive
Reinforcement Learning

• Notice: all previous methods required the model

• What if we don’t have it? Can we learn from pure exploration??

• Yes! This is “reinforcement learning”

• Today we’ll derive Q-learning, simplest model-free RL algorithm
Life of an RL Agent

• Agent lives in loop:
  1. receive observation (eg camera image)
  2. select action
  3. receive reward

T: Transition model (dynamics)
I: input (sensor reading)
R: reward (a real number)
B: behavior (an action)

Now “T” is outside the agent
How do we use $V(s)$ for planning?

$$\pi^*(s) = \arg\max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s')\right]$$

(1-step look-ahead)
We assumed a model for $P(s'|s,a)$

What do we do if such a model does not exist?

Model (without walls)

$P(s'|s,a)$ unknown!
We assumed a model for $P(j|i,a)$

What do we do if such a model does not exist?

- Learn one (e.g. Bayesian RL)
- "Model-Free" RL (e.g. Q-Learning)

Want: $\pi^*(s)$ – Optimal policy in the state

Can’t use: $V^*(s)$ – Value of a state

Learn instead: $Q^*(S_i,a)$ – Value of taking an action in a state
Q-function Definition

Key Relationship:

\[ V(s) = \max_a Q(s, a) \]

\[ Q(s,a) = \text{Value of Taking action } a \text{ in state } S \]

How do we use \( Q(s,a) \) for planning?

\[ \pi^*(s) = \arg \max_a Q(s, a) \]
Definition of Q function:

\[
Q(s, a) = R(s, a) + \gamma \max_{a'} \mathbb{E}[Q(s', a')]
\]

\[
= R(s, a) + \gamma \max_{a'} \sum_{s'} P(s' | s, a) Q(s', a')
\]

How to remove dependency on model?
From **Q-function to Q-Learning**

**Key question:** How to remove dependency on model?

\[
Q(s, a) = R(s, a) + \gamma \max_{a'} \sum_{s'} P(s'|s, a) Q(s', a')
\]

\[
\approx R(s, a) + \gamma \max_{a'} Q(s', a'), \quad s' \sim P(s'|s, a)
\]

\[
\approx (1 - \alpha)Q(s, a) + \alpha \left( R(s, a) + \gamma \max_{a'} Q(s', a') \right)
\]

\[
\approx Q(s, a) - \alpha Q(s, a) + \alpha R(s, a) + \alpha \gamma \max_{a'} Q(s', a')
\]

\[
\approx Q(s, a) + \alpha \left( R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)
\]

\[
\approx Q(s, a) + \alpha(\delta_{TD})
\]
Q-Learning Example

\[ \alpha = .7 \]

\[ \begin{array}{cccc}
   \text{s2} & \text{s3} & \text{s1} & \text{s4} \\
   -1 & -1 & -1 & 10 \\
\end{array} \]

\[ \begin{array}{cccc}
   \uparrow & \downarrow & \leftrightarrow & \leftrightarrow \\
   \text{S}_1 & 0 & 0 & 0 & 0 \\
   \text{S}_2 & 0 & 0 & 0 & 0 \\
   \text{S}_3 & 0 & 0 & 0 & 0 \\
   \text{S}_4 & 0 & 0 & 0 & 0 \\
\end{array} \]

Q-Table
### Q-Learning Example

**Q^{est}(S_1, \uparrow) = \frac{7}{10}(-1 + 0.9 \max(0, 0, 0, 0)) + 0.3 \times 0**

<table>
<thead>
<tr>
<th></th>
<th>Up</th>
<th>Down</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>-0.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Q-Learning Example

Q<sup>est</sup>(S<sub>2</sub>, →) =

\[.7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0\]

Q-Table

\[
\begin{array}{c|cccc}
 & \uparrow & \downarrow & \leftrightarrow & \leftrightarrow \\
\hline
S_1 & -.7 & 0 & 0 & 0 \\
S_2 & 0 & 0 & 0 & -.7 \\
S_3 & 0 & 0 & 0 & 0 \\
S_4 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Q-Learning Example

Q^{est}(S_3, \rightarrow) = .7(-1 + .9 \max (0, 0, 0, 0)) + .3 \times 0

<table>
<thead>
<tr>
<th></th>
<th>↑</th>
<th>↓</th>
<th>←</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>-.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-.7</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-.7</td>
</tr>
<tr>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Q-Table
Q-Learning Example

\[ Q_{\text{est}}(S_3, \downarrow) = 0.7(-1 + 0.9 \max(0, 0, 0, 0)) + 0.3 \times 0 \]

<table>
<thead>
<tr>
<th>( S_1 )</th>
<th>( \uparrow )</th>
<th>( \downarrow )</th>
<th>( \leftarrow )</th>
<th>( \rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>0</td>
<td>-0.7</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Q-Table
**Q-Learning Example**

Q-table:

<table>
<thead>
<tr>
<th>State</th>
<th>Up</th>
<th>Down</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>-.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-.7</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>-.7</td>
<td>0</td>
<td>-.7</td>
</tr>
<tr>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

$$Q_{est}(S_4, \leftarrow) = 0.7(10 + 0.9 \max(0, 0, 0, 0)) + 0.3 \times 0$$
Q-Learning Example

Q\textsuperscript{est}(S_4, \uparrow) = 0.7(10 + 0.9 \max (0, -0.7, 0, -0.7)) + 0.3 \times 0

Q-Table

<table>
<thead>
<tr>
<th></th>
<th>\uparrow</th>
<th>\downarrow</th>
<th>\leftarrow</th>
<th>\rightarrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>-0.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S_2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>S_3</td>
<td>0</td>
<td>-0.7</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>S_4</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Q-Learning Example

\[ Q_{\text{est}}(S_3, \downarrow) = 0.7(-1 + 0.9 \max (7,0,7,0)) + 0.3 \times -0.7 \]

Q-Table

<table>
<thead>
<tr>
<th></th>
<th>( \uparrow )</th>
<th>( \downarrow )</th>
<th>( \leftarrow )</th>
<th>( \rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>-0.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>0</td>
<td>3.5</td>
<td>0</td>
<td>-0.7</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
On-Policy Learning: SARSA

- Key idea: perform backups on action actually selected, rather than estimate of optimal action

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s, a) + \gamma Q(s', a') - Q(s, a) \right)$$

- Otherwise same as Q-learning, but “on-policy”

- Less greedy, so addresses problem of locally high-reward/risk states (e.g. cliff task)

The cliff-walking task. Off-policy Q-learning learns the optimal policy, along the safe path, but then keeps falling off because of the cliff. This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the single run, but smoothed.

The backup diagram for Q-learning. The rule (Figure 6.14) is the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Can you guess now what the diagram is? If so, please do make a guess before turning to the answer (on the next page).
TD and eligibility traces

- Problem: Q-values spread slowly
- Solution: Propagate over history
- Mechanism: exponential decay w.p. \( \lambda \)

TD error for last action

\[
\delta_{td} = R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)
\]

TD error at time \( T-t \)

\[
e(s, a) = \gamma^t \lambda^t \delta_{td}
\]
SARSA(λ)

Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$, for all $s, a$

Repeat (for each episode):
  Initialize $s, a$
  Repeat (for each step of episode):
    Take action $a$, observe $r, s'$
    Choose $a'$ from $s'$ using policy derived from $Q$
    (e.g., ε-greedy)
    \[\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)\]
    \[e(s, a) \leftarrow e(s, a) + 1\]
    For all $s, a$:
    \[Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)\]
    \[e(s, a) \leftarrow \gamma \lambda e(s, a)\]
    $s \leftarrow s'$, $a \leftarrow a'$
  until $s$ is terminal

This is the TD-error

Path Taken  action values increased by 1-step SARSA  action values increased by SARSA(λ), $λ=0.9$
Take-home

- Use Q-learning/SARSA when:
  - state space is tiny
  - interested in full policy
  - don’t have access to model
- Use eligibility traces when:
  - always.
Monte-Carlo Reinforcement Learning

- Recall:
  - V.I., P.I., Q-Learning, & SARSA are all direct implementations of bellman recursion, via dynamic programming

- MCRL is direct implementation of reward expectation, via sampling

\[ V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{T} \gamma^t r_t \right] \]

- Returns are simply averaged together
- Variance of the error decreases as \(1/n\)
Unpacking the bellman recursion:

\[ V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right] \]

\[ = \mathbb{E}_\pi \left[ r_0 + \sum_{t=0}^{\infty} \gamma^{t+1} r_{t+1} | s_0 = s \right] \]

\[ = \mathbb{E}_\pi \left[ r_0 + \gamma r_1 + \sum_{t=0}^{\infty} \gamma^{t+2} r_{t+2} | s_0 = s \right] \]

\[ = R(s) + \gamma \sum_a P_\pi(a|s) \sum_{s'} P(s'|s,a) \left[ R(s') + \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^{t+1} r_{t+1} | s_0 = s' \right] \right] \]

\[ = R(s) + \gamma \sum_a P_\pi(a|s) \sum_{s'} P(s'|s,a) V(s') \quad \text{back to bellman} \]

The point: you can approximate bellman using finite sums
Monte-Carlo Reinforcement Learning

How to visualize:

- Action selection according to some exploration policy $\pi$
- Transitions sampled from model or environment
Monte-Carlo Reinforcement Learning

• Key properties:
  • runtime independent of $|S|$ (!)
  • can learn from actual and simulated experience
  • can target parts of the state space we care about (!)

• Problems:
  • Slow
  • When to stop?
    • Variance falls as $1/n$, can we do better?
Sparse-Sampling (MCTS)

- ONLINE MCRL algorithm with provable loss bounds
  - Kearns, Mansour, Ng (ML 2002)
- Key idea: rewards in future matter less than rewards now
- Outputs:
  - \( \varepsilon \)-optimal policy
    \[ |V^A(s) - V^*(s)| \leq \varepsilon \]
Sparse-Sampling (MCTS)

Running Time: \( O((kC)^H) \)

• Hairy Math:

\[
H = \lceil \log_\gamma (\lambda/V_{\text{max}}) \rceil \\
C = \frac{V_{\text{max}}^2}{\lambda^2} \left( 2H \log \frac{kHV_{\text{max}}^2}{\lambda^2} + \log \frac{R_{\text{max}}}{\lambda} \right) \\
\lambda = (\epsilon(1 - \gamma)^2)/4, \quad V_{\text{max}} = R_{\text{max}}/(1 - \gamma)
\]

Planning horizon
Number of rollouts
Useful Constants

• Running time depends only on \( R_{\text{max}}, \epsilon, \) and \( \gamma \)

\[\Rightarrow\] The point: can do MCRL with provable guarantees. But how useful??
Sparse-Sampling (MCTS)

- Problem: $C$ can be HUGE

\[ \varepsilon = 0.1, \ R_{\text{max}} = 1 \]

\[ \gamma = 0.9, \ R_{\text{max}} = 1 \]

\[ \varepsilon = 0.1, \ \gamma = 0.9 \]
Take-home

- Use MCRL/MCTS when:
  - state space is huge
  - interested in subset of $S$ (online planning)
  - planning horizon is small
  - can efficiently sample from model
- Related work:
  - UCT (Kocsis et al 2006)
  - FSSS (Walsh et al 2010)