MDP Wrap-up + Reinforcement Learning

Sutton and Barto, Chapter 6.1, 6.2, 6.5
AIMA Chapter 21

Slides courtesy of Dan Klein and Pieter Abbeel
University of California, Berkeley
Markov Decision Processes - wrapup

Slides courtesy of Dan Klein and Pieter Abbeel --- University of California, Berkeley

[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.]
The Bellman Equations

How to be optimal:

Step 1: Take correct first action
Step 2: Keep being optimal
The Bellman Equations

- Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

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

\[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

\[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

- These are the Bellman equations, and they characterize optimal values in a way we’ll use over and over
Policy Methods
Policy Evaluation
Fixed Policies

- Expectimax trees max over all actions to compute the optimal values.
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state.
  - ... though the tree’s value would depend on which policy we fixed.
Utilities for a Fixed Policy

- Another basic operation: compute the utility of a state $s$ under a fixed (generally non-optimal) policy.

- Define the utility of a state $s$, under a fixed policy $\pi$:
  \[ V^\pi(s) = \text{expected total discounted rewards starting in } s \text{ and following } \pi \]

- Recursive relation (one-step look-ahead / Bellman equation):
  \[ V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')] \]
Example: Policy Evaluation

Always Go Right

Always Go Forward
Example: Policy Evaluation

Always Go Right

Always Go Forward
Policy Evaluation

- How do we calculate the V’s for a fixed policy \( \pi \)?
  - Idea 1: Turn recursive Bellman equations into updates (like value iteration)
    \[
    V_0^\pi(s) = 0
    \]
    \[
    V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')] \]
  - Efficiency: \( O(S^2) \) per iteration
  - Idea 2: Without the maxes, the Bellman equations are just a linear system
    - Solve with Matlab (or your favorite linear system solver)
Policy Extraction
Let’s imagine we have the optimal values $V^*(s)$

How should we act?
- It’s not obvious!

We need to do a mini-expectimax (one step)

\[
\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]
\]

This is called policy extraction, since it gets the policy implied by the values
Let’s imagine we have the optimal q-values:

How should we act?
  - Completely trivial to decide!

\[ \pi^*(s) = \arg \max_a Q^*(s, a) \]

Important lesson: actions are easier to select from q-values than values!
Policy Iteration
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Problem 1: It’s slow – \( O(S^2A) \) per iteration

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values

[Demo: value iteration (L9D2)]
$k=0$

Noise = 0.2
Discount = 0.9
Living reward = 0
k=1

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
### Values after 2 Iterations

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
<td><strong>0.72</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
<td><strong>-1.00</strong></td>
</tr>
<tr>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
<td><strong>0.00</strong></td>
</tr>
</tbody>
</table>

**Noise** = 0.2
**Discount** = 0.9
**Living reward** = 0
k=3

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=4$

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=5

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=7$

VALUES AFTER 7 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=8$

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=9$

Values after 9 iterations:

- Top left: 0.64
- Top middle: 0.74
- Top right: 0.85
- Middle left: 0.55
- Middle: 0.57
- Middle right: -1.00
- Bottom left: 0.46
- Bottom middle: 0.40
- Bottom right: 0.47
- Right: 0.27

Noise = 0.2
Discount = 0.9
Living reward = 0
$k = 11$

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=12

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=100$

VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

\[
V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]
\]

- Problem 1: It’s slow – \(O(S^2A)\) per iteration

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values
Policy Iteration

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

- **This is policy iteration**
  - It’s still optimal!
  - Can converge (much) faster under some conditions
Policy Iteration

- **Evaluation:** For fixed current policy $\pi$, find values with policy evaluation:
  - Iterate until values converge:
    \[
    V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]
    \]

- **Improvement:** For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
    \[
    \pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]
    \]
Both value iteration and policy iteration compute the same thing (all optimal values)

In value iteration:
- Every iteration updates both the values and (implicitly) the policy
- We don’t track the policy, but taking the max over actions implicitly recomputes it

In policy iteration:
- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
- The new policy will be better (or we’re done)

Both are dynamic programs for solving MDPs
Summary: MDP Algorithms

- So you want to....
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions
Double Bandits
- **Actions**: Blue, Red
- **States**: Win, Lose

The diagram illustrates a Double-Bandit MDP with transitions and rewards.

- **Win (W) to Win (W)**: 0.75 with reward $2
- **Win (W) to Lose (L)**: 0.25 with reward $0
- **Lose (L) to Lose (L)**: 0.75 with reward $2
- **Lose (L) to Win (W)**: 0.25 with reward $0

No discount
100 time steps
Both states have the same value
Solving MDPs is offline planning
- You determine all quantities through computation
- You need to know the details of the MDP
- You do not actually play the game!

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play Red</td>
<td>150</td>
</tr>
<tr>
<td>Play Blue</td>
<td>100</td>
</tr>
</tbody>
</table>

No discount
100 time steps
Both states have the same value
Let’s Play!
Rules changed! Red’s win chance is different.
Let’s Play!
What Just Happened?

- That wasn’t planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn’t solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - **Exploration**: you have to try unknown actions to get information
  - **Exploitation**: eventually, you have to use what you know
  - **Regret**: even if you learn intelligently, you make mistakes
  - **Sampling**: because of chance, you have to try things repeatedly
  - **Difficulty**: learning can be much harder than solving a known MDP
Reinforcement Learning

Read AIMA 21
Also read Sutton and Barton Chapter 6.1, 6.2 and 6.5 (see link on course web site)
Reinforcement Learning

Basic idea:
- Receive feedback in the form of rewards
- Agent’s utility is defined by the reward function
- Must (learn to) act so as to maximize expected rewards
- All learning is based on observed samples of outcomes!
Still assume a Markov decision process (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. we don’t know which states are good or what the actions do
- Must actually try actions and states out to learn
Offline (MDPs) vs. Online (RL)

Offline Solution

Online Learning
Model-Based Learning
Model-Based Learning

- **Model-Based Idea:**
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

- **Step 1: Learn empirical MDP model**
  - Count outcomes $s'$ for each $s, a$
  - Normalize to give an estimate of $\hat{T}(s, a, s')$
  - Discover each $\hat{R}(s, a, s')$ when we experience $(s, a, s')$

- **Step 2: Solve the learned MDP**
  - For example, use value iteration, as before
Example: Model-Based Learning

**Input Policy** $\pi$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
<td><strong>C</strong></td>
<td><strong>D</strong></td>
</tr>
</tbody>
</table>

Assume: $\gamma = 1$

**Observed Episodes (Training)**

- **Episode 1**
  - B, east, C, -1
  - C, east, D, -1
  - D, exit, x, +10

- **Episode 2**
  - B, east, C, -1
  - C, east, D, -1
  - D, exit, x, +10

- **Episode 3**
  - E, north, C, -1
  - C, east, D, -1
  - D, exit, x, +10

- **Episode 4**
  - E, north, C, -1
  - C, east, A, -1
  - A, exit, x, -10

**Learned Model**

$\widehat{T}(s, a, s')$

- $T(B, \text{east, } C) = 1.00$
- $T(C, \text{east, } D) = 0.75$
- $T(C, \text{east, } A) = 0.25$

...$\widehat{R}(s, a, s')$

- $R(B, \text{east, } C) = -1$
- $R(C, \text{east, } D) = -1$
- $R(D, \text{exit, } x) = +10$

...
Example: Expected Age

Goal: Compute expected age of CIS 421/521 students

Known P(A)

\[
E[A] = \sum_a P(a) \cdot a = 0.01 \times 42 + \ldots
\]

Without P(A), instead collect samples \([a_1, a_2, \ldots, a_N]\)

Unknown P(A): “Model Based”

\[
\hat{P}(a) = \frac{\text{num}(a)}{N}
\]

\[
E[A] \approx \sum_a \hat{P}(a) \cdot a
\]

Why does this work? Because eventually you learn the right model.

Unknown P(A): “Model Free”

\[
E[A] \approx \frac{1}{N} \sum_i a_i
\]

Why does this work? Because samples appear with the right frequencies.
Model-Free Learning
Passive Reinforcement Learning
Passive Reinforcement Learning

- **Simplified task: policy evaluation**
  - Input: a fixed policy $\pi(s)$
  - You don’t know the transitions $T(s,a,s')$
  - You don’t know the rewards $R(s,a,s')$
  - Goal: learn the state values

- **In this case:**
  - Learner is “along for the ride”
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - This is NOT offline planning! You actually take actions in the world.
Direct Evaluation

- **Goal:** Compute values for each state under $\pi$

- **Idea:** Average together observed sample values
  - Act according to $\pi$
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples

- **This is called direct evaluation**
Example: Direct Evaluation

**Input Policy** $\pi$

**Observed Episodes (Training)**

**Episode 1**
- B, east, C, -1
- C, east, D, -1
- D, exit, x, +10

**Episode 2**
- B, east, C, -1
- C, east, D, -1
- D, exit, x, +10

**Episode 3**
- E, north, C, -1
- C, east, D, -1
- D, exit, x, +10

**Episode 4**
- E, north, C, -1
- C, east, A, -1
- A, exit, x, -10

**Output Values**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+8</td>
<td>+4</td>
<td>+10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-10</td>
</tr>
</tbody>
</table>

*Assume: $\gamma = 1$*
Problems with Direct Evaluation

- What’s good about direct evaluation?
  - It’s easy to understand
  - It doesn’t require any knowledge of T, R
  - It eventually computes the correct average values, using just sample transitions

- What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

Output Values

If B and E both go to C under this policy, how can their values be different?
Why Not Use Policy Evaluation?

- **Simplified Bellman updates calculate** \( V \) **for a fixed policy:**
  - Each round, replace \( V \) with a one-step-look-ahead layer over \( V \)

\[
V_0^\pi(s) = 0
\]

\[
V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]
\]

- This approach fully exploited the connections between the states
- Unfortunately, we need \( T \) and \( R \) to do it!

- **Key question:** how can we do this update to \( V \) without knowing \( T \) and \( R \)?
  - In other words, how to we take a weighted average without knowing the weights?
Sample-Based Policy Evaluation?

- We want to improve our estimate of $V$ by computing these averages:

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

- Idea: Take samples of outcomes $s'$ (by doing the action!) and average

```
sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^\pi(s'_1)
sample_2 = R(s, \pi(s), s'_2) + \gamma V_k^\pi(s'_2)
... 
sample_n = R(s, \pi(s), s'_n) + \gamma V_k^\pi(s'_n)
```

$$V_{k+1}^\pi(s) \leftarrow \frac{1}{n} \sum_i sample_i$$
Temporal Difference Learning
Temporal Difference Learning

- Big idea: learn from every experience!
  - Update $V(s)$ each time we experience a transition $(s, a, s', r)$
  - Likely outcomes $s'$ will contribute updates more often

- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of $V(s)$:

$$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$$

Update to $V(s)$:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)sample$$

Same update:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$
Exponential Moving Average

- Exponential moving average
  - The running interpolation update: \( \bar{x}_n = (1 - \alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n \)

- Makes recent samples more important:

\[
\bar{x}_n = \frac{x_n + (1 - \alpha) \cdot x_{n-1} + (1 - \alpha)^2 \cdot x_{n-2} + \ldots}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots}
\]

- Forgets about the past (distant past values were wrong anyway)

- Decreasing learning rate (alpha) can give converging averages
Example: Temporal Difference Learning

Assume: $\gamma = 1$, $\alpha = 1/2$

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha \left[ R(s, \pi(s), s') + \gamma V^\pi(s') \right]$$
Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, if we want to turn values into a (new) policy, we’re sunk:

\[
\pi(s) = \arg \max_a Q(s, a)
\]

\[
Q(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V(s') \right]
\]

- Idea: learn Q-values, not values
- Makes action selection model-free too!
Active Reinforcement Learning
Active Reinforcement Learning

- **Full reinforcement learning: optimal policies (like value iteration)**
  - You don’t know the transitions $T(s,a,s')$
  - You don’t know the rewards $R(s,a,s')$
  - You choose the actions now
  - **Goal: learn the optimal policy / values**

- **In this case:**
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens...
Detour: Q-Value Iteration

- Value iteration: find successive (depth-limited) values
  - Start with $V_0(s) = 0$, which we know is right
  - Given $V_k$, calculate the depth $k+1$ values for all states:
    \[
    V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]
    \]

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s, a) = 0$, which we know is right
  - Given $Q_k$, calculate the depth $k+1$ q-values for all q-states:
    \[
    Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right]
    \]
Q-Learning

- **Q-Learning: sample-based Q-value iteration**

\[ Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] \]

- **Learn Q(s,a) values as you go**
  - Receive a sample \((s,a,s',r)\)
  - Consider your old estimate: \(Q(s, a)\)
  - Consider your new sample estimate:
    \[\text{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a')\]
  - Incorporate the new estimate into a running average:
    \[Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha) [\text{sample}]\]
Video of Demo Q-Learning -- Gridworld
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 not decrease it too quickly
- Basically, in the limit, it doesn’t matter how you select actions (!)