Stochastic Planning: MDPs

Recap: MDPs

- Markov decision processes:
  - States $S$
  - Actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s'|s,a)$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)
  - Start state $s_0$

- Quantities:
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)

Solving MDPs

- Value Iteration
- Real-Time Dynamic Programming
- Policy Iteration
- Reinforcement Learning

Optimal Quantities

- The value (utility) of a state $s$:
  \[ V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \]

- The value (utility) of a q-state $(s,a)$:
  \[ Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- The optimal policy:
  \[ \pi^*(s) = \text{optimal action from state } s \]

Gridworld Values $V^*$

<table>
<thead>
<tr>
<th>Values after 100 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.64</td>
</tr>
<tr>
<td>0.57</td>
</tr>
<tr>
<td>0.49</td>
</tr>
</tbody>
</table>

Demo: gridworldvclasses.34011
Gridworld: Q*

Q-VALUES AFTER 100 ITERATIONS

The Bellman Equations

1. Definition of "optimal utility" via expectimax recurrence
   gives a simple one-step lookahead relationship amongst optimal utility values

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

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

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

2. These are the Bellman equations, and they characterize optimal values in a way we'll use over and over

The Bellman Equations

- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: Deep parts of the tree eventually don't matter if \( \gamma < 1 \)

Racing Search Tree

The Bellman Equations

- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: Deep parts of the tree eventually don't matter if \( \gamma < 1 \)

Time-Limited Values

- Key idea: time-limited values
- Define \( V_k(s) \) to be the optimal value of \( s \) if the game ends in \( k \) more time steps
  - Equivalently, it’s what a depth-\( k \) expectimax would give from \( s \)

Time-Limited Values:

Avoiding Redundant Computation
**Value Iteration**

- Start with $V_k(s) = 0$: no time steps left means expected reward sum of zero.
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:
  \[ V_{k+1}(s) = \max_a \sum_s P(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
- Repeat until convergence
  (trust me, it does)

**Example:** Bellman Backup

\[
V_0 = 0 \\
Q_1(s, a_1) = 2 + \gamma 0 \sim 2 \\
Q_1(s, a_2) = 5 + \gamma 0.9 \sim 1 + \gamma 0.1 \sim 2 \sim 6.1 \\
Q_1(s, a_3) = 4.5 + \gamma 2 \sim 6.5
\]

If agent is in 4,3, it only has one legal action: get jewel. It gets a reward and the game is over.
If agent is in the pit, it has only one legal action: die. It gets a penalty and the game is over.
Agent does NOT get a reward for moving INTO 4,3.

<table>
<thead>
<tr>
<th>$k=0$</th>
<th>$k=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Values after 0 iterations" /></td>
<td><img src="image" alt="Values after 1 iterations" /></td>
</tr>
</tbody>
</table>

Noise = 0.2
Discount = 0.9
Living reward = 0
k=2

VALUES AFTER 2 ITERATIONS

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=6

VALUES AFTER 6 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

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

VALUES AFTER 9 ITERATIONS

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$

$\text{Noise} = 0.2$
$\text{Discount} = 0.9$
$\text{Living reward} = 0$
**Value Iteration**

- Start with $V_0(s) = 0$.
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:
  \[ V_{k+1}(s) = \max_a \sum_j T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \]
- Repeat until convergence
- Complexity of each iteration: $O(S^2 A)$
- Number of iterations: poly($|S|, |A|, 1/(1-g)$)
- Theorem: will converge to unique optimal values

**Bellman equations** characterize the optimal values:
\[ V^*(s) = \max_a \sum_j T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \]

Value iteration computes them:
\[ V_{k+1}(s) = \max_a \sum_j T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \]

Value iteration is just a fixed point solution method
- ... though the $V_k$ vectors are also interpretable as time-limited values

**Convergence**

- How do we know the $V_k$ vectors will converge?
- Case 1: If the tree has maximum depth $M$, then $V_M$ holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state $V_k$ and $V_{k+1}$ can be viewed as depth $k+1$ expectimax results in nearly identical search trees
  - The max difference happens if big reward at $k+1$ level
  - That last layer is at best all $R_{max}$
  - But everything is discounted by $\gamma^k$ that far out
  - So $V_k$ and $V_{k+1}$ are at most $\gamma^k |R|$ different
  - As $k$ increases, the values converge

**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)
  \[ \pi^*(s) = \arg \max_a \sum_j T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \]
- This is called policy extraction, since it gets the policy implied by the values

**Computing Actions from 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!
Problems with Value Iteration

- Value iteration repeats the Bellman updates:
  \[ V_{k+1}(s) = \max_{a} \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Problem 1: It’s slow – \(O(SA)\) per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values

VI \(\rightarrow\) Asynchronous VI

- Is it essential to back up all states in each iteration? No!
- States may be backed up
  - many times or not at all
  - in any order
- As long as no state gets starved...
  - convergence properties still hold!!

Prioritization of Bellman Backups

- Are all backups equally important?
- Can we avoid some backups?
- Can we schedule the backups more appropriately?
Asynch VI: Prioritized Sweeping

- Why backup a state if values of successors same?
- Prefer backing a state
  - whose successors had most change
- Priority Queue of (state, expected change in value)
- Backup in the order of priority
- After backing a state update priority queue
  - for all predecessors

Solving MDPs

- Value Iteration
- Policy Iteration
- Reinforcement Learning

Policy Methods

Policy Evaluation

Fixed Policies

Utilities for a Fixed Policy

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

- 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 π:
  \[ V^π(s) = \text{expected total discounted rewards starting in } s \text{ and following } π \]
- Recursive relation (one-step look-ahead / Bellman equation):
  \[ V^π(s) = \sum_{s'} T(s, π(s), s') [R(s, π(s), s') + \gamma V^π(s')] \]
### Example: Policy Evaluation

<table>
<thead>
<tr>
<th>Always Go Right</th>
<th>Always Go Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
</tbody>
</table>

### Example: Policy Evaluation

<table>
<thead>
<tr>
<th>Always Go Right</th>
<th>Always Go Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
</tbody>
</table>

---

### 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)
  
  
  $V_0^\pi(s) = 0$
  $V_{k+1}^\pi(s) = \sum_s P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$

- 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-expectmax (one step)
    
    $\pi^*(s) = \arg\max_a \sum_s P(s, a, s')[R(s, a, s') + \gamma V^*(s')]$

- This is called policy extraction, since it gets the policy implied by the values

---

### Computing Actions from Values

- Let’s imagine we have the optimal values $V^*(s)$
  
  - How should we act?
    - Completely trivial to decide!

### Computing Actions from Q-Values

- Let’s imagine we have the optimal q-values:
  
  $\pi^*(s) = \arg\max_a Q^*(s, a)$

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

<table>
<thead>
<tr>
<th>Evaluation: For fixed current policy ( \pi ), find values with policy evaluation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate until values converge:</td>
</tr>
<tr>
<td>( V_{\pi+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_{\pi}^\pi(s')] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Improvement: For fixed values, get a better policy using policy extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-step look-ahead:</td>
</tr>
<tr>
<td>( \pi_{\pi+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_{\pi}^\pi(s')] )</td>
</tr>
</tbody>
</table>

### Compare

<table>
<thead>
<tr>
<th>Both value iteration and policy iteration compute the same thing (all optimal values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In value iteration:</td>
</tr>
<tr>
<td>Every iteration updates both the values and (implicitly) the policy</td>
</tr>
<tr>
<td>We don’t track the policy, but taking the max over actions implicitly recomputes it</td>
</tr>
</tbody>
</table>

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

| 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-Bandit MDP

- Actions: Blue, Red
- States: Win, Lose

No discount 100 time steps
Both states have the same value

<table>
<thead>
<tr>
<th>State</th>
<th>Blue</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win</td>
<td>$2</td>
<td>$2</td>
</tr>
<tr>
<td>Lose</td>
<td>$0</td>
<td>$2</td>
</tr>
</tbody>
</table>

Value

- Play Red: 150
- Play Blue: 100

Offline Planning

- 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!

Online Planning

- Rules changed! Red’s win chance is different.

Let’s Play!

- Play Red
- Play Blue

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
Next Time: Reinforcement Learning!