CSEP 573: Artificial Intelligence

Markov Decision Processes (MDP)

Ali Farhadi

Many slides over the course adapted from Luke Zettlemoyer, Dan Klein, Pieter Abbeel, Stuart Russell or Andrew Moore
Outline (roughly next two weeks)

- Markov Decision Processes (MDP)
  - MDP formalism
  - Value Iteration
  - Policy Iteration

- Reinforcement Learning (RL)
  - Relationship to MDPs
  - Several learning algorithms
Non-deterministic Search

- Noisy execution of actions
  - Deterministic grid world vs. non-deterministic grid world
Example: Grid World

- A maze-like problem:
  - The agent lives in a grid
  - Walls block the agent’s path
- The agent’s actions do not always go as planned:
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- Agent receives rewards each time step:
  - Small “living” reward each step
  - Big rewards come at the end
- Goal: maximize sum of rewards
Grid World Actions

Deterministic

Stochastic

Actions
Review: Expectimax

- What if we don’t know what the result of an action will be? E.g.,
  - In solitaire, next card is unknown
  - In minesweeper, mine locations
  - In pacman, the ghosts act randomly

- Can do **expectimax search**
  - Chance nodes, like min nodes, except the outcome is uncertain
  - Calculate **expected utilities**
  - Max nodes as in minimax search
  - Chance nodes take average (expectation) of value of children

- Today, we’ll learn how to formalize the underlying problem as a **Markov Decision Process**
An MDP is defined by:

- A set of states \( s \in S \)
- A set of actions \( a \in A \)
- A transition function \( T(s,a,s') \)
  - Prob that \( a \) from \( s \) leads to \( s' \)
  - i.e., \( P(s' | s, a) \)
  - Also called the model
- A reward function \( R(s, a, s') \)
  - Sometimes just \( R(s) \) or \( R(s') \)
- A start state (or distribution)
- Maybe a terminal state

MDPs: non-deterministic search problems
- Reinforcement learning: MDPs where we don’t know the transition or reward functions
What is Markov about MDPs?

- Andrey Markov (1856-1922)
- “Markov” generally means that given the present state, the future and the past are independent
- For Markov decision processes, “Markov” means:

\[
P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots, S_0 = s_0) = P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\]

- This is just like search where the successor function only depends on the current state (not the history)
Solving MDPs

- In deterministic single-agent search problems, want an optimal plan, or sequence of actions, from start to a goal.

- In an MDP, we want an optimal policy \( \pi^* : S \rightarrow A \)
  - A policy \( \pi \) gives an action for each state
  - An optimal policy maximizes expected utility if followed
  - Defines a reflex agent.

- Expectimax didn’t compute the entire policy.
  - It computed the action for a single state only.

Optimal policy when \( R(s, a, s') = -0.03 \) for all non-terminals \( s \)
Example Optimal Policies

R(s) = -0.01
R(s) = -0.03
R(s) = -0.4
R(s) = -2.0
Another Example: Racing Car

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward

![Diagram showing transitions between states and actions with reward values.]

- From Cool to Slow: +1, 0.5
- From Slow to Cool: +1, 0.5
- From Cool to Warm: +2, 0.5
- From Warm to Slow: +1, 0.5
- From Slow to Warm: +1, 0.5
- From Warm to Fast: +2, 0.5
- From Fast to Warm: -10, 1.0
- From Warm to Overheated: -10, 1.0
- From Slow to Overheated: -10, 1.0
- From Overheated to Fast: 1.0
- From Fast to Overheated: -10

The rewards reflect the outcomes of actions and state transitions.
Each MDP state gives an expectimax-like search tree

- (s, a) is a q-state
- T(s, a, s') = P(s' | s, a)
- R(s, a, s')

- s is a state
- (s, a, s') called a transition
Utilities of Sequences

- What preference should an agent have over reward sequences?
- More or less:
  - $[1, 2, 2]$ or $[2, 3, 4]$ 
- Now or later:
  - $[0, 0, 1]$ or $[1, 0, 0]$
Discounting

- It is reasonable to maximize the sum of rewards
- It also makes sense to prefer rewards now to rewards later
- One solution: value of rewards decay exponentially

\[
\text{Worth now} \quad 1 \\
\text{Worth in one step} \quad \gamma \\
\text{Worth in two step} \quad \gamma^2
\]
Discounting

- **How to discount?**
  - Each time we descend, we multiply in the discount once

- **Why discount?**
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge

- **Example: discount of 0.5**
  - $U([1, 2, 3]) = 1*1 + .5*2 + .25*3$
  - $U([1,2,3]) < U([3,2,1])$
Discounting

\[ U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma) \]

- Typically discount rewards by \( \gamma < 1 \) each time step
  - Sooner rewards have higher utility than later rewards
  - Also helps the algorithms converge
Quiz: Discounting

- Given:

<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>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Actions: East, West, and Exit (only available in exit states a, e)
- Transitions: deterministic

- Quiz 1: For $\gamma = 1$, what is the optimal policy?

- Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

- Quiz 3: For which $\gamma$ are West and East equally good when in state d?
Utilities of Sequences

- In order to formalize optimality of a policy, need to understand utilities of sequences of rewards.
- Typically consider stationary preferences:

\[ [r, r_0, r_1, r_2, \ldots] \succ [r, r'_0, r'_1, r'_2, \ldots] \]
\[ \iff \]
\[ [r_0, r_1, r_2, \ldots] \succ [r'_0, r'_1, r'_2, \ldots] \]

- Two ways to define stationary utilities:
  - Additive utility:
    \[ U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \ldots \]
  - Discounted utility:
    \[ U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \ldots \]
Infinite Utilities?! 

- Problem: what if the game lasts forever?
  - Infinite state sequences have infinite rewards
- Solutions:
  - Finite horizon:
    - Terminate episodes after a fixed $T$ steps (e.g. life)
    - Gives nonstationary policies ($\pi$ depends on time left)
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)
  - Discounting: for $0 < \gamma < 1$

\[
U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma)
\]

- Smaller $\gamma$ means smaller “horizon” – shorter term focus
Recap: Defining MDPs

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

- **MDP quantities so far:**
  - Policy = Choice of action for each state
  - Utility (or return) = sum of discounted rewards
Solving MDPs

- We want to find the optimal policy $\pi^*$:
  - Find best action for each state such that it maximizes Utility (or return) = sum of discounted rewards
Optimal Utilities

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

- Define the value of a q-state $(s,a)$:
  \[ Q^*(s,a) = \text{expected utility starting in } s, \text{ taking action } a \text{ and thereafter acting optimally} \]

- Define the optimal policy:
  \[ \pi^*(s) = \text{optimal action from state } s \]
VALUES AFTER 100 ITERATIONS

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.85</td>
<td>0.89</td>
<td>0.93</td>
<td>1.00</td>
</tr>
<tr>
<td>0.81</td>
<td></td>
<td></td>
<td>-1.00</td>
</tr>
<tr>
<td>0.77</td>
<td>0.73</td>
<td>0.70</td>
<td>0.47</td>
</tr>
</tbody>
</table>
Q-VALUES AFTER 100 ITERATIONS
The Bellman Equations

- Definition of “optimal utility” leads to a simple one-step lookahead relationship amongst optimal utility values:
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax does

- Formally:

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

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

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

- Find $V^*(s)$ for all the states in $S$
  - $|S|$ non-linear equations with $|S|$ unknown

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

- Our proposal:
  - Dynamic programming
  - Define $V^*(s)$ as the optimal value of $s$ if game ends in $i$ steps
  - $V^0(s) = 0$ for all the states

$$V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]$$
Racing Car Search Tree

- 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 $\gamma < 1$
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 $s$
Example: $\gamma=0.9$, living reward=0, noise=0.2
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

VALUES AFTER 2 ITERATIONS
Example: Bellman Updates

$$V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right] = \max_a Q_{i+1}(s, a)$$

$$Q_1(\langle 3, 3 \rangle, \text{right}) = \sum_{s'} T(\langle 3, 3 \rangle, \text{right}, s') \left[ R(\langle 3, 3 \rangle, \text{right}, s') + \gamma V_i(s') \right]$$

$$= 0.8 \times [0.0 + 0.9 \times 1.0] + 0.1 \times [0.0 + 0.9 \times 0.0] + 0.1 \times [0.0 + 0.9 \times 0.0]$$
Example: Value Iteration

- Information propagates outward from terminal states and eventually all states have correct value estimates.
VALUES AFTER 3 ITERATIONS
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.51</td>
<td>0.72</td>
<td>0.84</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.27</td>
<td></td>
<td>0.55</td>
<td>-1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.22</td>
<td>0.37</td>
<td>0.13</td>
</tr>
</tbody>
</table>

VALUES AFTER 5 ITERATIONS
VALUES AFTER 6 ITERATIONS
VALUES AFTER 7 ITERATIONS
Recap: Value Iteration

- **Idea:**
  - Start with $V_0^*(s) = 0$, which we know is right (why?)
  - Given $V_i^*$, calculate the values for all states for depth $i+1$:

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

  - This is called a value update or Bellman update
  - Repeat until convergence

- **Theorem:** will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Why Not Search Trees?

- Why not solve with expectimax?

- Problems:
  - This tree is usually infinite (why?)
  - Same states appear over and over (why?)
  - We would search once per state (why?)

- Idea: Value iteration
  - Compute optimal values for all states all at once using successive approximations
  - Will be a bottom-up dynamic program similar in cost to memoization
  - Do all planning offline, no replanning needed!
Computing time limited values

\[ V_4(\square) \quad V_4(\text{truck}) \quad V_4(\text{car}) \]

\[ V_3(\square) \quad V_3(\text{truck}) \quad V_3(\text{car}) \]

\[ V_2(\square) \quad V_2(\text{truck}) \quad V_2(\text{car}) \]

\[ V_1(\square) \quad V_1(\text{truck}) \quad V_1(\text{car}) \]

\[ V_0(\square) \quad V_0(\text{truck}) \quad V_0(\text{car}) \]
Example of Value iteration

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

Assume no discount!
Recap: Value Estimates

- **Calculate estimates** $V_k^*(s)$
  - The optimal value considering only next k time steps (k rewards)
  - As $k \to \infty$, it approaches the optimal value
- **Why:**
  - If discounting, distant rewards become negligible
  - If terminal states reachable from everywhere, fraction of episodes not ending becomes negligible
  - Otherwise, can get infinite expected utility and then this approach actually won’t work
Convergence

- How do we know the $V_k$ vectors are going to 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 difference is that on the bottom layer, $V_{k+1}$ has actual rewards while $V_k$ has zeros
  - That last layer is at best all $R_{\text{MAX}}$
  - It is at worst $R_{\text{MIN}}$
  - But everything is discounted by $\gamma^k$ that far out
  - So $V_k$ and $V_{k+1}$ are at most $\gamma^k \max |R|$ different
  - So as $k$ increases, the values converge
Value Iteration Complexity

- **Problem size:**
  - $|A|$ actions and $|S|$ states

- **Each Iteration**
  - Computation: $O(|A| \cdot |S|^2)$
  - Space: $O(|S|)$

- **Num of iterations**
  - Can be exponential in the discount factor $\gamma$
Practice: Computing Actions

- Which action should we chose from state $s$:
  - Given optimal values $Q$?
    \[ \arg \max_a Q^*(s, a) \]
  - Given optimal values $V$?
    \[ \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \]
- Lesson: actions are easier to select from $Q$’s!
Aside: Q-Value Iteration

- **Value iteration**: find successive approx optimal values
  - Start with $V_0^*(s) = 0$
  - Given $V_i^*$, calculate the values for all states for depth $i+1$:
    \[
    V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]
    \]

- But Q-values are more useful!
  - Start with $Q_0^*(s,a) = 0$
  - Given $Q_i^*$, calculate the q-values for all q-states for depth $i+1$:
    \[
    Q_{i+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_i(s', a') \right]
    \]
Example: Value Iteration
Utilities for Fixed Policies

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

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

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

- How do we calculate the V’s for a fixed policy?
- **Idea one:** modify Bellman updates

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

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

- **Idea two:** it’s just a linear system, solve with Matlab (or whatever)
Policy Iteration

- Problem with value iteration:
  - Considering all actions each iteration is slow: takes $|A|$ times longer than policy evaluation
  - But policy doesn’t change each iteration, time wasted

- Alternative to value iteration:
  - **Step 1: Policy evaluation:** calculate utilities for a fixed policy (not optimal utilities!) until convergence (fast)
  - **Step 2: Policy improvement:** update policy using one-step lookahead with resulting converged (but not optimal!) utilities (slow but infrequent)
  - Repeat steps until policy converges
Policy Iteration

- **Policy evaluation**: with fixed current policy $\pi$, find values with simplified Bellman updates
  - Iterate until values converge

  \[ V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} T(s, \pi_k(s), s') \left[ R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s') \right] \]

  - Note: could also solve value equations with other techniques

- **Policy improvement**: with fixed utilities, find the best action according to one-step look-ahead

  \[ \pi_{k+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_k}(s') \right] \]
Policy Iteration Complexity

- **Problem size:**
  - $|A|$ actions and $|S|$ states

- **Each Iteration**
  - Computation: $O(|S|^3 + |A| \cdot |S|^2)$
  - Space: $O(|S|)$

- **Num of iterations**
  - Unknown, but can be faster in practice
  - Convergence is guaranteed
Comparison

- In value iteration:
  - Every pass (or “backup”) updates both utilities (explicitly, based on current utilities) and policy (possibly implicitly, based on current policy)

- In policy iteration:
  - Several passes to update utilities with frozen policy
  - Occasional passes to update policies

- Hybrid approaches (asynchronous policy iteration):
  - Any sequences of partial updates to either policy entries or utilities will converge if every state is visited infinitely often
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**