Lecture 22: Markov Decision Processes

Jacob Steinhardt

November 9, 2021
Previous lectures have explored several themes:

- Decision-making (e.g. FDR, bandits)
- Time dynamics and statefulness (e.g. Markov models)
- Value of information (e.g. multi-armed bandits)

We will combine all of these with

1. *Markov decision processes* (stateful decision-making), and
2. *reinforcement learning* (stateful decision-making with uncertainty).
Roadmap

- Review: dynamic programming
- Markov decision processes
  - Bellman equations
  - Solution via dynamic programming
- Reinforcement learning (next lecture)
Dynamic programming warm-up: Fibonacci

Fibonacci sequence: \( F_n = F_{n-1} + F_{n-2} \) \((F_0 = 0, F_1 = 1)\)

Recursive function:

```python
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

What happens if we call \texttt{fib(50)}?
Exponential blow-up
Solution 1: Memoization

Remember answers in a dict:

```python
memo_dict = dict()
def fib(n):
    if n in memo_dict.keys():
        return memo_dict[n]
    elif n <= 1:
        ans = n
    else:
        ans = fib(n-1) + fib(n-2)
    memo_dict[n] = ans
    return ans
```

Can use decorators for slick code

Slow (dict lookup each time)

J. Steinhardt
Solution 1: Memoization

Remember answers in a `dict`:

```python
memo_dict = dict()

def fib(n):
    if n in memo_dict.keys():
        return memo_dict[n]
    elif n <= 1:
        ans = n
    else:
        ans = fib(n-1) + fib(n-2)
    memo_dict[n] = ans
    return ans
```

- Can use decorators for slick code
- Slow (dict lookup each time)
Solution 2: Dynamic Programming

Can replace with for loop if we do things in right order:

```python
import numpy as np
n_max = 50
fibs = np.array(n_max)
fibs[0], fibs[1] = 0, 1
for n in range(2, n_max):
    fibs[n] = fibs[n-1] + fibs[n-2]
```
Solution 2: Dynamic Programming

Can replace with for loop if we do things in right order:

```python
import numpy as np
n_max = 50
fibs = np.array(n_max)
fibs[0], fibs[1] = 0, 1
for n in range(2, n_max):
    fibs[n] = fibs[n-1] + fibs[n-2]
```

- **Pro**: fast, low-memory
- **Con**: more thinking; need to find linear structure
Harder example: car and gas stations

- Locations 0, ..., n
- Car starts at location 0, wants to get to location n
- Each location \( i \): gas station selling \( g_i \) units of gas at \( c_i \) dollars per unit
- 1 unit of gas to move 1 unit right

**Challenge:**
How much gas should we buy at each location to minimize total cost?
Solution via recursion

- State: (location, gas left in tank)
- Define $f(loc, \text{gas}) = \text{minimum cost to get to end given current state} \text{ ("cost-to-go")}$
- Two options: buy 1 unit of gas (stay where we are), or go forward
Solution via recursion

- State: (location, gas left in tank)
- Define \( f(\text{loc}, \text{gas}) = \) minimum cost to get to end given current state ("cost-to-go")
- Two options: buy 1 unit of gas (stay where we are), or go forward

```python
def f(loc, gas):
    if loc == n:
        return 0
    if gas < 0:
        return -np.inf
    cost1 = f(loc, gas + 1) + price[loc]
    cost2 = f(loc + 1, gas - 1)
    return min(cost1, cost2)
```
Solution via dynamic programming

```python
def solve_dynamic_programming(n, price):
    f = np.zeros((n+1, n+1))
    for loc in range(n-1, -1, -1):
        for gas in range(n, -1, -1):
            cost1 = f[loc, gas+1] + price[loc]
            cost2 = f[loc+1, gas-1]
            f[loc, gas] = min(cost1, cost2)
```

Gas station problem is a special case of Markov decision process. Will define these next and see how to formulate a general dynamic programming solution.
Solution via dynamic programming

```python
f = np.zeros(shape=(n+1,n+1))
for loc in range(n-1, -1, -1):
    for gas in range(n, -1, -1):
        cost1 = f[loc, gas+1] + price[loc]
        cost2 = f[loc+1, gas-1]
        f[loc, gas] = min(cost1, cost2)
```

- Gas station problem is special case of *Markov decision process*
- Will define these next and see how to formulate a general dynamic programming solution
DS 102: Data, Inference, and Decisions

Lecture 23: Markov Decision Processes

Jacob Steinhardt
University of California, Berkeley

Slides thanks and credit: Fernando Perez, Anca Dragan, Dan Klein, Pieter Abbeel, and the Berkeley CS188 team: ai.berkeley.edu.
Markov Decision Processes

- An MDP is defined by:
  - A set of states \( s \in S \)
  - A set of actions \( a \in A \)
  - A transition function \( T(s, a, s') \)
    - Probability that \( a \) from \( s \) leads to \( s' \), i.e., \( P(s'|s, a) \)
    - Also called the model or the dynamics
  - A reward function \( R(s, a, s') \)
    - Sometimes just \( R(s) \) or \( R(s') \)
  - A start state
  - Maybe a terminal state
Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent’s path
- Noisy movement: 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
- The agent receives rewards each time step
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards
Grid World Actions

Deterministic Grid World

Stochastic Grid World
What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent.

- For Markov decision processes, “Markov” means action outcomes depend only on the current state.

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

\[
P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\]

- This is just like search, where the successor function could only depend on the current state (not the history).

Andrey Markov (1856-1922)
Policies

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

- For MDPs, we want an optimal policy $\pi^*$: $S \rightarrow A$
  - A policy $\pi$ gives an action for each state.
  - An optimal policy is one that maximizes expected utility if followed.
  - An explicit policy defines a reflex agent.

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

R(s) = -0.01

R(s) = -0.03

R(s) = -0.4

R(s) = -2.0
Discounting

- It’s reasonable to maximize the sum of rewards
- It’s also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

\begin{align*}
1 & \quad \text{Worth Now} \\
\gamma & \quad \text{Worth Next Step} \\
\gamma^2 & \quad \text{Worth In Two Steps}
\end{align*}
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:
  \[ \pi^*(s) = \text{optimal action from state } s \]
Values of States

- Recursive definition of value:

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

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

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

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

Recursion for $V^*$ is circular:

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

- Not a problem for gas stations because states were totally ordered
- Can’t assume this in general
- Solution: add a time component

$$V^*(s, t) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s', t - 1)],$$

$$V^*(s, 0) = 0$$
Solving the recursion

Recursion for $V^*$ is circular:

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

- Not a problem for gas stations because states were totally ordered
- Can’t assume this in general
- Solution: add a time component

$$V^*(s, t) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s', t - 1)],$$

$$V^*(s, 0) = 0$$

- Time $t$ creates total ordering!
- Can recover $V^*(s)$ by taking $t \to \infty$
Value learning via dynamic programming

\[ V = \text{np.zeros}(\text{shape}=(\text{num}\_\text{states}, t\_\text{max})) \]

```python
for t in range(1, t\_\text{max}):
    for s in range(num\_\text{states}):
        V[s, t] = max([sum([T(s, a, s2) * (R(s, a, s2) + \gamma * V[s2, t-1])
                         for s2 in num\_\text{states}])
                       for a in num\_\text{actions}])
```

J. Steinhardt
MDPs
November 9, 2021 22 / 24
Can save memory with “sliding window” trick:

```python
V = np.zeros(shape=(num_states, t_max))
for t in range(1, t_max):
    for s in range(num_states):
        V[s, t] = max([sum([T(s, a, s2) * (R(s, a, s2) + gamma * V[s2, t-1])
                          for s2 in num_states])
                        for a in num_actions])
```

J. Steinhardt

MDPs

November 9, 2021 22 / 24
Can save memory with “sliding window” trick:

\[
V = \text{np.zeros} \text{(num\_states)}
\]

```python
for t in range(1, t\_max):
    V\_old = \text{np.copy} (V)
    for s in range(num\_states):
        V[s] = \text{max} (\text{sum} (\text{T}(s, a, s2) \times (\text{R}(s, a, s2) + gamma \times V\_old[s2])
            for s2 in range(num\_states)
            for a in range(num\_actions))
```
Exploiting monotonicity

Since updates monotonically approach $V^*$, can update in place:

```python
V = np.zeros(num_states)
for t in range(1, t_max):
    for s in range(num_states):
        V[s] = max(
            sum([T(s, a, s2) * (R(s, a, s2) + gamma * V[s2])
                 for s2 in num_states]
            for a in num_actions)
```

J. Steinhardt

MDPs

November 9, 2021 23 / 24
Recap

- Defined Markov decision process:
  - states, actions, (stochastic) transitions, rewards
- Recursion (Bellman equations)
- Efficient solution via dynamic programming
- Even more efficient solution exploiting monotonicity (in-place updates)
- Next lecture: what if transitions need to be learned? (RL)