Chapter 4: Dynamic Programming

Objectives of this chapter:

- Overview of a collection of classical solution methods for MDPs known as dynamic programming (DP)
- Show how DP can be used to compute value functions, and hence, optimal policies
- Discuss efficiency and utility of DP
Policy Evaluation

**Policy Evaluation**: for a given policy \( \pi \), compute the state-value function \( V^\pi \)

Recall:

\[
v_\pi(s) \doteq \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s]
= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]
= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma v_\pi(s') \right],
\]

A system of \(|S|\) simultaneous linear equations in \(|S|\) unknowns.
Iterative Methods

\[ \nu_0 \rightarrow \nu_1 \rightarrow \cdots \rightarrow \nu_k \rightarrow \nu_{k+1} \rightarrow \cdots \rightarrow \nu^\pi \]

A "sweep"

A sweep consists of applying a backup operation to each state.

A full policy-evaluation backup:

\[
\nu_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma \nu_k(S_{t+1}) \mid S_t = s] \\
= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma \nu_k(s') \right],
\]
Iterative Policy Evaluation

### Iterative Policy Evaluation

Input $\pi$, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in S^+$
Repeat
- $\Delta \leftarrow 0$
  - For each $s \in S$:
    - $v \leftarrow V(s)$
    - $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
    - $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$
A Small Gridworld

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is \(-1\) until the terminal state is reached

\[ r = -1 \]
on all transitions
Iterative Policy Eval. for the Small Gridworld

\[ \pi = \text{equiprobable random action choices} \]
Policy Improvement

Suppose we have computed $v_\pi$ for a deterministic policy $\pi$.

For a given state $s$, would it be better to do an action $a \neq \pi(s)$?

The value of doing $a$ in state $s$ is:

$$q_\pi(s, a) = E_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a \right]$$

$$= \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right]$$

It is better to switch to action $a$ for state $s$ if and only if

$$q_\pi(s, a) > v_\pi(s)$$
Do this for all states to get a new policy $\pi'$ that is \textbf{greedy} with respect to $v_\pi$:

$$
\pi'(s) = \arg\max_a q_{\pi}(s, a)
$$

$$
= \arg\max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_{\pi}(s')]
$$

Then $v_{\pi'} \geq v_\pi$
What if $v_{\pi'} = v_\pi$?

i.e., for all $s \in S$, $v_{\pi'}(s) = \max_a \sum_{s'} p(s', r | s, a) \left[ r + \gamma v_{\pi'}(s') \right]$?

But this is the Bellman Optimality Equation.

So $v_{\pi'} = v_*$ and both $\pi$ and $\pi'$ are optimal policies.
Policy Iteration

\[ \pi_0 \rightarrow v_{\pi_0} \rightarrow \pi_1 \rightarrow v_{\pi_1} \rightarrow \cdots \pi_* \rightarrow v_* \]

policy evaluation \quad policy improvement

“greedification”
## Policy Iteration

### Policy iteration (using iterative policy evaluation)

1. **Initialization**  
   \[ V(s) \in \mathbb{R} \text{ and } \pi(s) \in \mathcal{A}(s) \text{ arbitrarily for all } s \in S \]

2. **Policy Evaluation**  
   Repeat  
   \[ \Delta \leftarrow 0 \]  
   For each \( s \in S \):  
   \[ v \leftarrow V(s) \]  
   \[ V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')] \]  
   \[ \Delta \leftarrow \max(\Delta, |v - V(s)|) \]  
   until \( \Delta < \theta \) (a small positive number)

3. **Policy Improvement**  
   \( \text{policy-stable} \leftarrow \text{true} \)  
   For each \( s \in S \):  
   \[ \text{old-action} \leftarrow \pi(s) \]  
   \[ \pi(s) \leftarrow \arg\max_{a} \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')] \]  
   If \( \text{old-action} \neq \pi(s) \), then \( \text{policy-stable} \leftarrow \text{false} \)  
   If \( \text{policy-stable} \), then stop and return \( V \approx v_* \) and \( \pi \approx \pi_* \); else go to 2

---

Example 4.2: Jack's Car Rental

Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning the probability that the number is \( n \) is \( \frac{n^n}{n!} e^{-n} \), where \( \mu \) is the expected number.

Suppose \( \mu = 3 \) and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can...
Jack’s Car Rental

- $10 for each car rented (must be available when request rec’d)
- Two locations, maximum of 20 cars at each
- Cars returned and requested randomly
  - Poisson distribution, $n$ returns/requests with prob $\frac{\lambda^n}{n!} e^{-\lambda}$
  - 1st location: average requests = 3, average returns = 3
  - 2nd location: average requests = 4, average returns = 2
- Can move up to 5 cars between locations overnight

- States, Actions, Rewards?
- Transition probabilities?
Figure 4.2: The sequence of policies found by policy iteration on Jack’s car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal.

Exercise 4.5 (programming)
Write a program for policy iteration and re-solve Jack’s car rental problem with the following changes. One of Jack’s employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs $2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of $4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. If your computer is too slow for the full problem, cut all the numbers of cars in half.
Value Iteration

Do we need to always completely evaluate a policy before improving it? No!

Recall the full policy-evaluation backup:

$$v_{k+1}(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_k(s')]$$

Here is the full value-iteration backup:

$$v_{k+1}(s) \leftarrow \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_k(s')]$$
Value Iteration

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in S^+$)

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that

$\pi(s) = \arg\max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$
Gambler’s Problem

- Gambler can repeatedly bet $ on a coin flip
- Heads he wins his stake, tails he loses it
- Initial capital $ \in \{1, 2, \ldots , 99\}$
- Gambler wins if his capital becomes $100$
  loses if it becomes $0$
- Coin is unfair
  - Heads (gambler wins) with probability $p = .4$

- States, Actions, Rewards?
4.5. ASYNCHRONOUS DYNAMIC PROGRAMMING

R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction
All the DP methods described so far require exhaustive sweeps of the entire state set.

Asynchronous DP does not use sweeps. Instead it works like this:

- Repeat until convergence criterion is met:
  - Pick a state at random and apply the appropriate backup

Still need lots of computation, but does not get locked into hopelessly long sweeps

Can you select states to backup intelligently? YES: an agent’s experience can act as a guide.
Generalized Policy Iteration (GPI):
any interaction of policy evaluation and policy improvement, independent of their granularity.

A geometric metaphor for convergence of GPI:

\[
\pi \rightarrow \text{greedy}(v) \rightarrow \pi \rightarrow \text{greedy}(v) \rightarrow \pi \rightarrow \text{greedy}(v) \rightarrow \cdots
\]

\[
\pi \rightarrow \nu \rightarrow \nu \rightarrow \cdots \rightarrow \nu \rightarrow \pi \rightarrow \nu \rightarrow \cdots
\]
Efficiency of DP

- To find an optimal policy is polynomial in the number of states…
- BUT, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (what Bellman called “the curse of dimensionality”).
- In practice, classical DP can be applied to problems with a few millions of states.
- Asynchronous DP can be applied to larger problems, and is appropriate for parallel computation.
- It is surprisingly easy to come up with MDPs for which DP methods are not practical.
Summary

- Policy evaluation: backups without a max
- Policy improvement: form a greedy policy, if only locally
- Policy iteration: alternate the above two processes
- Value iteration: backups with a max
- Full backups (to be contrasted later with sample backups)
- Generalized Policy Iteration (GPI)
- Asynchronous DP: a way to avoid exhaustive sweeps
- **Bootstrapping**: updating estimates based on other estimates