Lecture 3:
Markov Decision Processes and Dynamic Programming

Diana Borsa

January 15, 2021
Background

Sutton & Barto 2018, Chapter 3 + 4
Recap

- Reinforcement learning is the science of learning to make decisions.
- Agents can learn a policy, value function and/or a model.
- The general problem involves taking into account time and consequences.
- Decisions affect the reward, the agent state, and environment state.
This Lecture

- Last lecture: multiple actions, but only one state—no model
- This lecture:
  - Formalise the problem with full **sequential structure**
  - Discuss first class of solution methods which assume **true model is given**
  - These methods are called **dynamic programming**
- Next lectures: use similar ideas, but use sampling instead of true model
Formalising the RL interaction
Formalising the RL interface

- We will discuss a mathematical formulation of the agent-environment interaction.
- This is called a Markov Decision Process (MDP).
- Enables us to talk clearly about the objective and how to achieve it.
MDPs: A simplifying assumption

▶ For now, assume the environment is fully observable:
  ⇒ the current observation contains all relevant information

▶ Note: Almost all RL problems can be formalised as MDPs, e.g.,
  ▶ Optimal control primarily deals with continuous MDPs
  ▶ Partially observable problems can be converted into MDPs
  ▶ Bandits are MDPs with one state
Markov Decision Process

**Definition (Markov Decision Process - Sutton & Barto 2018)**

A Markov Decision Process is a tuple \((S, A, p, \gamma)\), where

- \(S\) is the set of all possible states
- \(A\) is the set of all possible actions (e.g., motor controls)
- \(p(r, s' | s, a)\) is the joint probability of a reward \(r\) and next state \(s'\), given a state \(s\) and action \(a\)
- \(\gamma \in [0, 1]\) is a discount factor that trades off later rewards to earlier ones

**Observations:**

- \(p\) defines the **dynamics** of the problem
- Sometimes it is useful to marginalise out the state transitions or expected reward:

\[
p(s' | s, a) = \sum_r p(s', r | s, a) \quad \mathbb{E}[R | s, a] = \sum_r r \sum_{s'} p(r, s' | s, a).\]
Definition (Markov Decision Process)

A Markov Decision Process is a tuple \((S, A, p, r, \gamma)\), where

- \(S\) is the set of all possible states
- \(A\) is the set of all possible actions (e.g., motor controls)
- \(p(s' \mid s, a)\) is the probability of transitioning to \(s'\), given a state \(s\) and action \(a\)
- \(r : S \times A \rightarrow \mathbb{R}\) is the expected reward, achieved on a transition starting in \((s, a)\)

\[
r = \mathbb{E}[R \mid s, a]
\]

- \(\gamma \in [0, 1]\) is a discount factor that trades off later rewards to earlier ones

Note: These are equivalent formulations: no additional assumptions w.r.t the previous def.
Markov Property: *The future is independent of the past given the present*

### Definition (Markov Property)

Consider a sequence of random variables, \( \{S_t\}_{t \in \mathbb{N}} \), indexed by time. A state \( s \) has the Markov property when for states \( \forall s' \in S \)

\[
p (S_{t+1} = s' \mid S_t = s) = p (S_{t+1} = s' \mid h_{t-1}, S_t = s)
\]

for all possible histories \( h_{t-1} = \{S_1, \ldots, S_{t-1}, A_1, \ldots, A_{t-1}, R_1, \ldots, R_{t-1}\} \)

In a Markov Decision Process all states are assumed to have the Markov property.

- The state captures all relevant information from the history.
- Once the state is known, the history may be thrown away.
- The state is a sufficient statistic of the past.
Markov Property in a MDP: Test your understanding

In a Markov Decision Process all states are assumed to have the Markov property.

Q: In an MDP this property implies: (Which of the following statements are true?)

\[
p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \ldots, S_{t-1}, A_1, \ldots, A_t, S_t = s) \quad (1)
\]

\[
p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \ldots, S_{t-1}, S_t = s, A_t = a) \quad (2)
\]

\[
p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \ldots, S_{t-1}, S_t = s) \quad (3)
\]

\[
p(R_{t+1} = r, S_{t+1} = s' \mid S_t = s) = p(R_{t+1} = r, S_{t+1} = s' \mid S_1, \ldots, S_{t-1}, S_t = s) \quad (4)
\]
Example: cleaning robot

- Consider a robot that cleans soda cans
- Two states: high battery charge or low battery charge
- Actions: \{wait, search\} in high, \{wait, search, recharge\} in low
- Dynamics may be stochastic
  - \( p(S_{t+1} = \text{high} \mid S_t = \text{high}, A_t = \text{search}) = \alpha \)
  - \( p(S_{t+1} = \text{low} \mid S_t = \text{high}, A_t = \text{search}) = 1 - \alpha \)
- Reward could be expected number of collected cans (deterministic), or actual number of collected cans (stochastic)

Reference: Sutton and Barto, Chapter 3, pg 52-53.
### Example: Robot MDP

| \(s\) | \(a\)     | \(s'\) | \(p(s' | s, a)\) | \(r(s, a, s')\) |
|-------|----------|--------|-----------------|-----------------|
| high  | search   | high   | \(\alpha\)     | \(r_{\text{search}}\) |
| high  | search   | low    | \(1 - \alpha\) | \(r_{\text{search}}\) |
| low   | search   | high   | \(1 - \beta\)  | \(-3\)          |
| low   | search   | low    | \(\beta\)      | \(r_{\text{search}}\) |
| high  | wait     | high   | 1               | \(r_{\text{wait}}\) |
| high  | wait     | low    | 0               | \(r_{\text{wait}}\) |
| low   | wait     | high   | 0               | \(r_{\text{wait}}\) |
| low   | wait     | low    | 1               | \(r_{\text{wait}}\) |
| low   | recharge | high   | 1               | 0               |
| low   | recharge | low    | 0               | 0               |
Example: robot MDP
Formalising the objective
Returns

- Acting in a MDP results in immediate rewards $R_t$, which leads to returns $G_t$:
  - Undiscounted return (episodic/finite horizon pb.)
    \[
    G_t = R_{t+1} + R_{t+2} + \ldots + R_T = \sum_{k=0}^{T-t-1} R_{t+k+1}
    \]
  - Discounted return (finite or infinite horizon pb.)
    \[
    G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t} R_T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}
    \]
  - Average return (continuing, infinite horizon pb.)
    \[
    G_t = \frac{1}{T-t-1} (R_{t+1} + R_{t+2} + \ldots + R_T) = \frac{1}{T-t-1} \sum_{k=0}^{T-t-1} R_{t+k+1}
    \]

Note: These are random variables that depend on MDP and policy.
Discounted Return

- Discounted returns $G_t$ for infinite horizon $T \to \infty$:

\[ G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \]

- The discount $\gamma \in [0, 1]$ is the present value of future rewards
  - The marginal value of receiving reward $R$ after $k + 1$ time-steps is $\gamma^k R$
  - For $\gamma < 1$, immediate rewards are more important than delayed rewards
  - $\gamma$ close to 0 leads to "myopic" evaluation
  - $\gamma$ close to 1 leads to "far-sighted" evaluation
Most Markov decision processes are discounted. Why?

- **Problem specification:**
  - Immediate rewards may actually be more valuable (e.g., consider earning interest)
  - Animal/human behaviour shows preference for immediate reward

- **Solution side:**
  - Mathematically convenient to discount rewards
  - Avoids infinite returns in cyclic Markov processes

- The way to think about it: **reward and discount together determine the goal**
## Goal of an RL agent

To find a behaviour policy that maximises the (expected) return $G_t$

- A **policy** is a mapping $\pi: S \times A \rightarrow [0, 1]$ that, for every state $s$ assigns for each action $a \in A$ the probability of taking that action in state $s$. Denoted by $\pi(a|s)$.
- For deterministic policies, we sometimes use the notation $a_t = \pi(s_t)$ to denote the action taken by the policy.
The value function $v(s)$ gives the long-term value of state $s$

$$v_{\pi}(s) = \mathbb{E} [G_t \mid S_t = s, \pi]$$

We can define (state-)action values:

$$q_{\pi}(s, a) = \mathbb{E} [G_t \mid S_t = s, A_t = a, \pi]$$

(Connection between them) Note that:

$$v_{\pi}(s) = \sum_a \pi(a \mid s)q_{\pi}(s, a) = \mathbb{E} [q_{\pi}(S_t, A_t) \mid S_t = s, \pi], \forall s$$
The optimal state-value function $v^*(s)$ is the maximum value function over all policies

$$v^*(s) = \max_{\pi} v_\pi(s)$$

The optimal action-value function $q^*(s, a)$ is the maximum action-value function over all policies

$$q^*(s, a) = \max_{\pi} q_\pi(s, a)$$

- The optimal value function specifies the best possible performance in the MDP
- An MDP is “solved” when we know the optimal value function
Optimal Policy

Define a partial ordering over policies

\[ \pi \geq \pi' \iff v_{\pi}(s) \geq v_{\pi'}(s), \forall s \]

Theorem (Optimal Policies)

For any Markov decision process

- There exists an optimal policy \( \pi^* \) that is better than or equal to all other policies,
  \[ \pi^* \geq \pi, \forall \pi \]
  (There can be more than one such optimal policy.)
- All optimal policies achieve the optimal value function, \( v_{\pi^*}(s) = v^*(s) \)
- All optimal policies achieve the optimal action-value function, \( q_{\pi^*}(s, a) = q^*(s, a) \)
Finding an Optimal Policy

An optimal policy can be found by maximising over \( q^*(s, a) \),

\[
\pi^*(s, a) = \begin{cases} 
1 & \text{if } a = \arg\max_{a \in A} q^*(s, a) \\
0 & \text{otherwise}
\end{cases}
\]

Observations:
- There is always a deterministic optimal policy for any MDP
- If we know \( q^*(s, a) \), we immediately have the optimal policy
- There can be multiple optimal policies
- If multiple actions maximize \( q^*(s, \cdot) \), we can also just pick any of these (including stochastically)
Bellman Equations
The value function $v(s)$ gives the long-term value of state $s$

$$v_\pi(s) = \mathbb{E} [G_t \mid S_t = s, \pi]$$

It can be defined recursively:

$$v_\pi(s) = \mathbb{E} [R_{t+1} + \gamma G_{t+1} \mid S_t = s, \pi]$$

$$= \mathbb{E} [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)]$$

$$= \sum_a \pi(a \mid s) \sum_r \sum_{s'} p(r, s' \mid s, a) (r + \gamma v_\pi(s'))$$

The final step writes out the expectation explicitly
Action values

- We can define state-action values

\[ q_\pi(s, a) = \mathbb{E} [G_t \mid S_t = s, A_t = a, \pi] \]

- This implies

\[
q_\pi(s, a) = \mathbb{E} [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
= \mathbb{E} [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\
= \sum_r \sum_{s'} p(r, s' \mid s, a) \left( r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a') \right)
\]

- Note that

\[
v_\pi(s) = \sum_a \pi(a \mid s) q_\pi(s, a) = \mathbb{E} [q_\pi(S_t, A_t) \mid S_t = s, \pi], \forall s
\]
Bellman Equations

**Theorem (Bellman Expectation Equations)**

Given an MDP, $\mathcal{M} = \langle S, A, p, r, \gamma \rangle$, for any policy $\pi$, the value functions obey the following expectation equations:

\[
v_\pi(s) = \sum_a \pi(s, a) \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v_\pi(s') \right]
\] (5)

\[
q_\pi(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \sum_{a' \in A} \pi(a'|s') q_\pi(s', a')
\] (6)
The Bellman Optimality Equations

**Theorem (Bellman Optimality Equations)**

Given an MDP, \( \mathcal{M} = \langle S, A, p, r, \gamma \rangle \), the **optimal value functions** obey the following **expectation equations**:

\[
\begin{align*}
    v^*(s) &= \max_a \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v^*(s') \right] \quad (7) \\
    q^*(s, a) &= r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a'} q^*(s', a') \quad (8)
\end{align*}
\]

There can be no policy with a higher value than \( v_*(s) = \max_\pi v_\pi(s), \forall s \)
Some intuition

(Reminder) Greedy on $v^* = \text{Optimal Policy}$

- An optimal policy can be found by maximising over $q^*(s, a)$,

$$
\pi^*(s, a) = \begin{cases} 
1 & \text{if } a = \arg\max_{a \in A} q^*(s, a) \\
0 & \text{otherwise}
\end{cases}
$$

- Apply the Bellman Expectation Eq. (6):

$$
q_{\pi^*}(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \sum_{a' \in A} \pi^*(a'|s') q_{\pi^*}(s', a')
= r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a' \in A} q^*(s', a')
$$
Solving RL problems using the Bellman Equations
Problems in RL

▶ *Pb1:* Estimating $v_\pi$ or $q_\pi$ is called policy evaluation or, simply, prediction
  ▶ Given a policy, what is my expected return under that behaviour?
  ▶ Given this treatment protocol/trading strategy, what is my expected return?

▶ *Pb2:* Estimating $v_*$ or $q_*$ is sometimes called control, because these can be used for policy optimisation
  ▶ What is the optimal way of behaving? What is the optimal value function?
  ▶ What is the optimal treatment? What is the optimal control policy to minimise time, fuel consumption, etc?
Consider the following MDP:

- The actions have a 0.9 probability of success and with 0.1 probably we remain in the same state.
- $R_t = 0$ for all transitions that end up in $S_0$, and $R_t = -1$ for all other transitions.
Exercise: (pause to work this out)

- Consider the following MDP:

![MDP Diagram]

- The actions have a 0.9 probability of success and with 0.1 probability we remain in the same state.
- $R_t = 0$ for all transitions that end up in $S_0$, and $R_t = -1$ for all other transitions.

Q: Evaluation problems (Consider a discount $\gamma = 0.9$)

- What is $v_\pi$ for $\pi(s) = a_1(\rightarrow), \forall s$?
- What is $v_\pi$ for the uniformly random policy?
- Same policy evaluation problems for $\gamma = 0.0$? (What do you notice?)
A solution
The Bellman value equation, for given $\pi$, can be expressed using matrices,

$$v = r^{\pi} + \gamma P^{\pi} v$$

where

$$v_i = v(s_i)$$

$$r_i^{\pi} = \mathbb{E}[R_{t+1} \mid S_t = s_i, A_t \sim \pi(S_t)]$$

$$P_{ij}^{\pi} = p(s_j \mid s_i) = \sum_a \pi(a \mid s_i)p(s_j \mid s_i, a)$$
The Bellman equation, for a given policy $\pi$, can be expressed using matrices,

$$v = r^\pi + \gamma P^\pi v$$

This is a linear equation that can be solved directly:

$$v = r^\pi + \gamma P^\pi v$$

$$(I - \gamma P^\pi) v = r^\pi$$

$$v = (I - \gamma P^\pi)^{-1} r^\pi$$

Computational complexity is $O(|S|^3)$ — only possible for small problems

There are iterative methods for larger problems
  - Dynamic programming
  - Monte-Carlo evaluation
  - Temporal-Difference learning
Solving the Bellman Optimality Equation

- The Bellman optimality equation is non-linear
- Cannot use the same direct matrix solution as for policy optimisation (in general)

- Many iterative solution methods:
  - Using models / dynamic programming
    - Value iteration
    - Policy iteration
  - Using samples
    - Monte Carlo
    - Q-learning
    - Sarsa
Dynamic Programming
The 1950s were not good years for mathematical research. I felt I had to shield the Air Force from the fact that I was really doing mathematics. What title, what name, could I choose? I was interested in planning, in decision making, in thinking. But planning is not a good word for various reasons. I decided to use the word ‘programming.’ I wanted to get across the idea that this was dynamic, this was time-varying—I thought, let’s kill two birds with one stone. Let’s take a word that has a precise meaning, namely dynamic, in the classical physical sense. It also is impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

– Richard Bellman

(slightly paraphrased for conciseness)
Dynamic programming

Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).

Sutton & Barto 2018

- We will discuss several dynamic programming methods to solve MDPs
- All such methods consist of two important parts:
  - policy evaluation
  - policy improvement
We start by discussing how to estimate $v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | s, \pi]$

Idea: turn this equality into an update

Algorithm

First, initialise $v_0$, e.g., to zero

Then, iterate

$\forall s : v_{k+1}(s) \leftarrow \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | s, \pi]$

Stopping: whenever $v_{k+1}(s) = v_k(s)$, for all $s$, we must have found $v_\pi$

Q: Does this algorithm always converge?

Answer: Yes, under appropriate conditions (e.g., $\gamma < 1$). More next lecture!
Example: Policy evaluation

$$R_t = -1$$
on all transitions
Policy evaluation

<table>
<thead>
<tr>
<th></th>
<th>$k = 0$</th>
<th></th>
<th>$k = 1$</th>
<th></th>
<th>$k = 2$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td>0.0</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td>0.0</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td>0.0</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td>0.0</td>
<td></td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0.0</th>
<th>-1.0</th>
<th>-1.0</th>
<th>-1.0</th>
<th></th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Policy evaluation

<table>
<thead>
<tr>
<th>$k = 3$</th>
<th>0.0</th>
<th>-2.4</th>
<th>-2.9</th>
<th>-3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-2.4</td>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
</tr>
<tr>
<td></td>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
</tr>
<tr>
<td></td>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k = 10$</th>
<th>0.0</th>
<th>-6.1</th>
<th>-8.4</th>
<th>-9.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-6.1</td>
<td>-7.7</td>
<td>-8.4</td>
<td>-8.4</td>
</tr>
<tr>
<td></td>
<td>-8.4</td>
<td>-8.4</td>
<td>-7.7</td>
<td>-6.1</td>
</tr>
<tr>
<td></td>
<td>-9.0</td>
<td>-8.4</td>
<td>-6.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k = \infty$</th>
<th>0.0</th>
<th>-14.</th>
<th>-20.</th>
<th>-22.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-22.</td>
<td>-20.</td>
<td>-14.</td>
<td>0.0</td>
</tr>
</tbody>
</table>
### Policy evaluation + Greedy Improvement

<table>
<thead>
<tr>
<th>$k = 0$</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

### $k = 1$

<table>
<thead>
<tr>
<th>0.0</th>
<th>-1.0</th>
<th>-1.0</th>
<th>-1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

### $k = 2$

<table>
<thead>
<tr>
<th>0.0</th>
<th>-1.7</th>
<th>-2.0</th>
<th>-2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.7</td>
<td>-2.0</td>
<td>-2.0</td>
<td>-2.0</td>
</tr>
<tr>
<td>-2.0</td>
<td>-2.0</td>
<td>-2.0</td>
<td>-1.7</td>
</tr>
<tr>
<td>-2.0</td>
<td>-2.0</td>
<td>-1.7</td>
<td>0.0</td>
</tr>
</tbody>
</table>

*random policy*
Policy evaluation + Greedy Improvement

$k = 3$

\[
\begin{array}{cccc}
0.0 & -2.4 & -2.9 & -3.0 \\
-2.4 & -2.9 & -3.0 & -2.9 \\
-2.9 & -3.0 & -2.9 & -2.4 \\
-3.0 & -2.9 & -2.4 & 0.0 \\
\end{array}
\]

$k = 10$

\[
\begin{array}{cccc}
0.0 & -6.1 & -8.4 & -9.0 \\
-6.1 & -7.7 & -8.4 & -8.4 \\
-8.4 & -8.4 & -7.7 & -6.1 \\
-9.0 & -8.4 & -6.1 & 0.0 \\
\end{array}
\]

$k = \infty$

\[
\begin{array}{cccc}
0.0 & -14. & -20. & -22. \\
-22. & -20. & -14. & 0.0 \\
\end{array}
\]
Policy Improvement

- The example already shows we can use evaluation to then improve our policy
- In fact, just being greedy with respect to the values of the random policy sufficed! (That is not true in general)

Algorithm

Iterate, using

\[ \forall s : \quad \pi_{\text{new}}(s) = \arg\max_a q_\pi(s, a) \]

\[ = \arg\max_a \mathbb{E} \left[ R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a \right] \]

Then, evaluate \( \pi_{\text{new}} \) and repeat

- Claim: One can show that \( v_{\pi_{\text{new}}}(s) \geq v_\pi(s) \), for all \( s \)
Policy Improvement: $q_{\pi_{\text{new}}}(s, a) \geq q_{\pi}(s, a)$
Policy Iteration

Policy evaluation Estimate $\nu^\pi$

Policy improvement Generate $\pi' \geq \pi$

$V = V^\pi$

$\pi = \text{greedy}(V)$

$V^* \rightarrow \pi^*$

$V \rightarrow V^\pi$

$\pi \rightarrow \text{greedy}(V)$

$\pi^* \rightarrow V^*$
Example: Jack’s Car Rental

- States: Two locations, maximum of 20 cars at each
- Actions: Move up to 5 cars overnight (-$2 each)
- Reward: $10 for each available car rented, $\gamma = 0.9$
- Transitions: 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
Example: Jack’s Car Rental – Policy Iteration
Policy Iteration

- Does policy evaluation need to converge to $v^\pi$?

- Or should we stop when we are ‘close’?
  (E.g., with a threshold on the change to the values)
  - Or simply stop after $k$ iterations of iterative policy evaluation?
  - In the small gridworld $k = 3$ was sufficient to achieve optimal policy

- **Extreme**: Why not update policy every iteration — i.e. stop after $k = 1$?
  - This is equivalent to value iteration
We could take the Bellman optimality equation, and turn that into an update

\[ \forall s : \quad v_{k+1}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s] \]

This is equivalent to policy iteration, with \( k = 1 \) step of policy evaluation between each two (greedy) policy improvement steps.

Algorithm: Value Iteration

- Initialise \( v_0 \)
- Update: \( v_{k+1}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s] \)
- Stopping: whenever \( v_{k+1}(s) = v_k(s) \), for all \( s \), we must have found \( v^* \)
Example: Shortest Path

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V3</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Problem

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>V7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Synchronous Dynamic Programming Algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>Bellman Equation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>Bellman Expectation Equation</td>
<td>Iterative Policy Evaluation</td>
</tr>
<tr>
<td>Control</td>
<td>Bellman Expectation Equation + (Greedy) Policy Improvement</td>
<td>Policy Iteration</td>
</tr>
<tr>
<td>Control</td>
<td>Bellman Optimality Equation</td>
<td>Value Iteration</td>
</tr>
</tbody>
</table>

**Observations:**

- Algorithms are based on state-value function $v_\pi(s)$ or $v^*(s) \Rightarrow$ complexity $O(|A||S|^2)$ per iteration, for $|A|$ actions and $|S|$ states
- Could also apply to action-value function $q_\pi(s, a)$ or $q^*(s, a) \Rightarrow$ complexity $O(|A|^2|S|^2)$ per iteration
Extensions to Dynamic Programming
Asynchronous Dynamic Programming

- DP methods described so far used **synchronous** updates (all states in parallel)

- **Asynchronous DP**
  - backs up states individually, in any order
  - can significantly reduce computation
  - guaranteed to converge if all states continue to be selected
Asynchronous Dynamic Programming

Three simple ideas for asynchronous dynamic programming:

- In-place dynamic programming
- Prioritised sweeping
- Real-time dynamic programming
In-Place Dynamic Programming

- Synchronous value iteration stores two copies of value function

  \[
  \text{for all } s \text{ in } S : \quad v_{\text{new}}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_{\text{old}}(S_{t+1}) | S_t = s]
  \]

  \[
  v_{\text{old}} \leftarrow v_{\text{new}}
  \]

- In-place value iteration only stores one copy of value function

  \[
  \text{for all } s \text{ in } S : \quad v(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) | S_t = s]
  \]
Prioritised Sweeping

- Use magnitude of Bellman error to guide state selection, e.g.

\[
\max_a \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) | S_t = s] - v(s)
\]

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue
Real-Time Dynamic Programming

- Idea: only update states that are relevant to agent
- E.g., if the agent is in state $S_t$, update that state value, or states that it expects to be in soon
Full-Width Backups

- Standard DP uses full-width backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using true model of transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers from **curse of dimensionality**
  - Number of states $n = |S|$ grows exponentially with number of state variables
- Even one full backup can be too expensive
Sample Backups

- In subsequent lectures we will consider **sample backups**
- Using sample rewards and sample transitions $\langle s, a, r, s' \rangle$
  (Instead of reward function $r$ and transition dynamics $p$)
- **Advantages:**
  - Model-free: no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of $n = |S|$
Summary
What have we covered today?

- Markov Decision Processes
- Objectives in an MDP: different notion of return
- Value functions - expected returns, condition on state (and action)
- Optimality principles in MDPs: optimal value functions and optimal policies
- Bellman Equations
- Two class of problems in RL: evaluation and control
- How to compute $v_\pi$ (aka solve an evaluation/prediction problem)
- How to compute the optimal value function via dynamic programming:
  - Policy Iteration
  - Value Iteration
Questions?

_The only stupid question is the one you were afraid to ask but never did._
-Rich Sutton

For questions that may arise during this lecture please use Moodle and/or the next Q&A session.