Lecture 3:
Markov Decision Processes and Dynamic Programming

Matteo Hessel, Hado van Hasselt

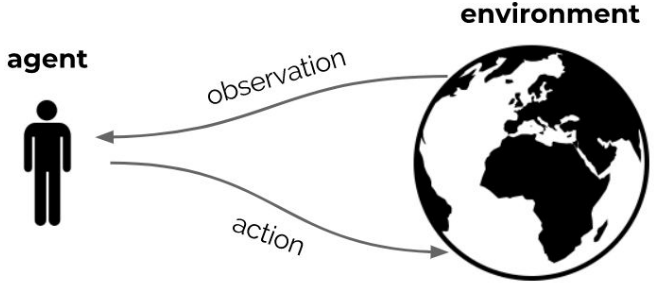
DeepMind

Resources

References, downloadable for free, for today's material

- ***[Reinforcement Learning an Introduction (2nd Ed.)](#)***, Sutton and Barto, Chapter 3/4

Recap [Reinforcement Learning]



Reinforcement learning is the problem of learning to control the environment

- choosing among multiple possible **actions**
- despite any **stochasticity** in the outcome of these action,
- considering different **contexts** or **states** in which the actions are taken,
- accounting for the **sequential nature** of decision making in complex domains,

Recap [State]

The **environment state** is the environment's internal state.

- May not be visible to the agent
- May contain lots of irrelevant information

The **agent state** is a summary of everything that the agent has observed up to now.

$$H_t = O_0, A_0, R_1, O_1, ..., O_{t-1}, A_{t-1}, R_t, O_t$$
$$S_t = f(H_t)$$

Recap [Bandits]

The **multi-armed bandit problem**, that we considered last lecture, is a simplification of the full RL problem, isolating the fundamental issue of **exploration** vs **exploitation**.

- multiple **actions** ✓
- **stochasticity** ✓
- multiple **states** ✗
- **sequential** structure ✗

Today we discuss how to formalize the full RL problem **(**with **Markov Decision Processes),** and a class of solutions to this problem (**Dynamic Programming**).

Formalizing RL environments: MDPs

Markov decision processes (MDP) formalize **environments** as tuples (S, A, p), where

- S is the set of all possible **states**,
- A is the set of all possible **actions,**
- $p(r, s' | s, a)$ is the problem's **dynamics**, specifying the joint probability of ending in each state s' with a reward r, after taking action a in a previous state s,
- The dynamics is such that the **markov property** is satisfied.

Markov Property

The **Markov Property** is a key assumption of Markov Decision Processes,

$$\boxed{p(r, s' | S_t = s) = p(r, s' | S_1, S_2, \ldots, S_t = s) \qquad \forall r, s', s \quad \forall S_1, \ldots, S_{t-1}}$$

This is often expressed by saying that

- 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.

A very general formalism

Markov Decision Processes are a very powerful conceptual tool.

- Directly models **fully observable** RL environments
- Bandits are a special case of MDPs (with a **single** state)
- Optimal Control Theory primarily deals with **continuous** MDPs
- **Partially observable** environments can also be converted to MDPs

Example [The cleaning robot]

Consider a cleaning robot that must collect empty cans

- **States**: 1) high battery charge 2) low battery charge
- **Actions**: {wait, search} in high, {wait, search, recharge} in low

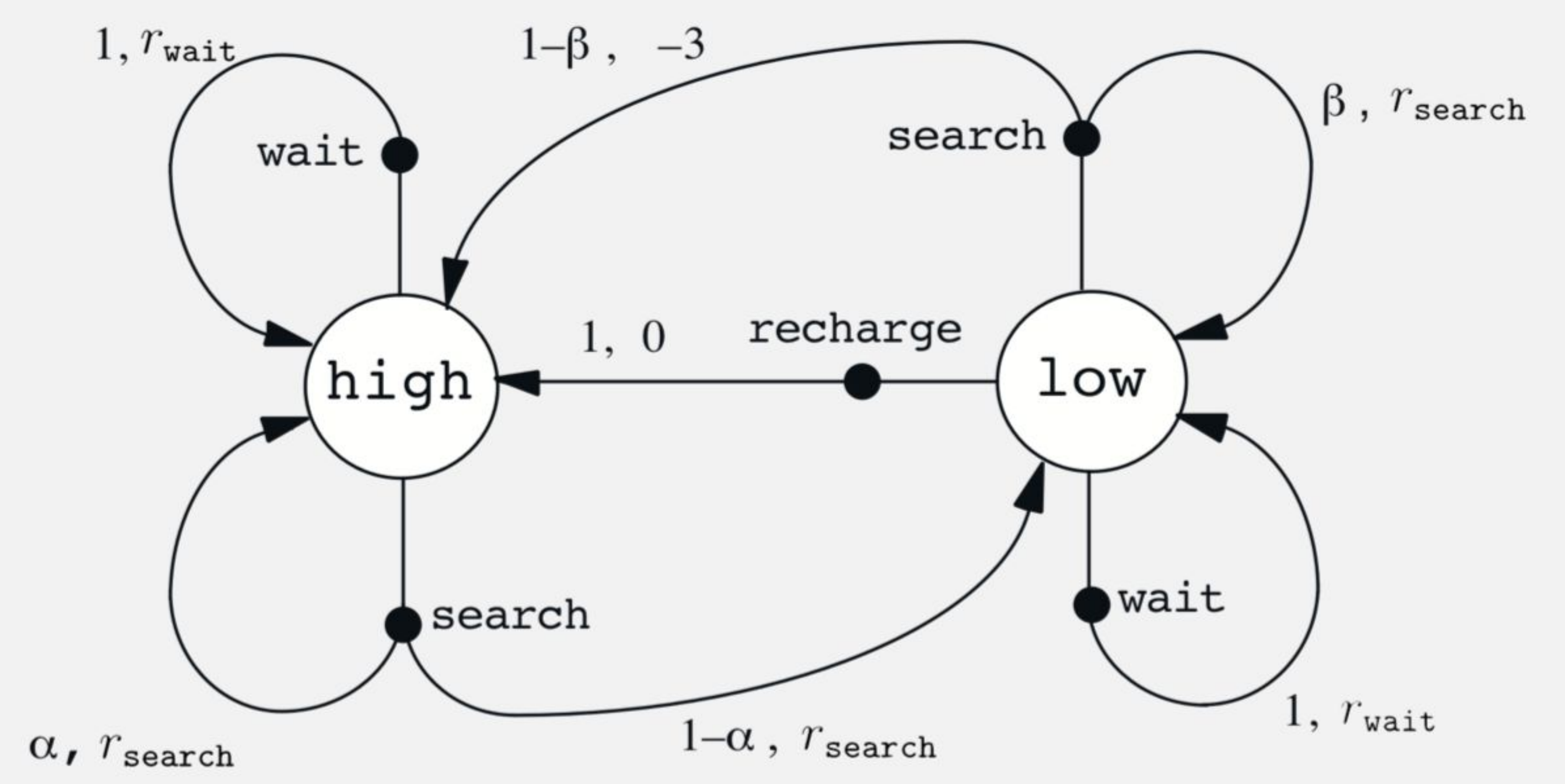
We then need to specify the dynamics:

- **Reward** is the expected or actual number of collected items
- **Transitions** are some probability distribution over triplets (s, a s')

Example [The cleaning robot]

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	$r_{\texttt{wait}}$
low	wait	high	0	$r_{\texttt{wait}}$
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	0

Example [The cleaning robot]



Policy

A **policy** defines the agent's **behaviour**

- It is a function from the (agent/environment) states onto the action space
- **Deterministic** policy → $A = \pi(S)$
- **Stochastic** policy → $\pi(A|S) = P(A|S)$

The Return

Taking actions in an MDP results in observing sequences of rewards.

As objective we typically consider their **cumulative discounted sum** (a.k.a. the **return**):

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n R_{t+n+1}$$

This is a **random variable**, that depends on:

- the environment's dynamics *p*
- the agent's policy π

Why discounting?

The **discount** factor $\gamma \in [0, 1]$ trades off immediate versus distant rewards

The discount effectively defines an **horizon** for the return:

- $\gamma = 0 \rightarrow$ we only care about the immediate reward,
- $0 < \gamma < 1, \quad r = 1 \rightarrow$ the returns totals a cumulative reward of $\frac{1}{1-\gamma}$

In **continuing** environments $\gamma < 1$ ensures the return is well defined

In **episodic** environments discounted returns are often simpler to maximize

Caveat

You will often find the discount as part of the MDP specification:

Indeed, some environments define themselves a **natural discounting** of future rewards

- E.g., *inflation* in a financial setting

Most often, however, agents maximize for a different (often lower) discount

- Simpler learning problem → superior performance even in terms of *real* objective

Therefore it's useful to consider it part of the **agent's objective**

Values

Since G_t is a random variable we typically consider some *statistics* of the return:

- a simple popular choice is the *expectation* of the return;

The **value** $v_\pi(s)$ of a state s is the **expected return** when starting in state s and sampling actions according to policy π

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

This is a function of **dynamics** *p*, **policy** π and the chosen **discount** factor γ

Recursive decomposition of values

The **value** function $v_\pi(s)$ gives the expected long-term value of state S:

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

It **decompose** as sum of the immediate reward and the long-term value of next state:

$$\begin{aligned}
v_\pi(s) = E_\pi[G_t | S_t = s] &= E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s] \\
&= E_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots) | S_t = s] \\
&= E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
&= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]
\end{aligned}$$

Action-Value Functions

We can also define **action values**:

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

As a result they also admit a **recursive** decomposition:

$$\begin{aligned} q_\pi(s, a) &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

Where we have used the **law of total expectation**, which in this case means that:

$$v_\pi(s) = E_\pi[q_\pi(s, A_t) | S_t = s]$$

Solving the Bellman Equation

Solving the Bellman equation for v_π and q_π is called the **prediction** problem.

Bellman equations, for given π, can be expressed using **matrices**,

for instance in the case of state values: $\mathbf{v} = \mathbf{r} + \gamma P^\pi \mathbf{v}$

where:

$$v_i = v_\pi(s_i)$$

$$r_i = E[R_t | S_t = s_i, A_t \sim \pi(S_t)]$$

$$P^\pi_{ij} = \sum_a \pi(a|s_i) p(s_j | s_i, a)$$

Solving the Bellman Equation

Equation $\mathbf{v} = \mathbf{r} + \gamma P^{\pi}\mathbf{v}$ defines a system of n linear equations in n variables,

It can be solved **directly**, yielding the value of each state under policy π (by def of ***v***)

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

Computationally too expensive for most problems $O(|S|^3)$

Optimal Values

The **optimal state-value function** is the maximum value function over all policies

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

The **optimal action-value function** is the maximum action-value function over all policies

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

- Optimal value functions specify the best possible performance in an MDP;
- Estimating v* or q* is referred to as **control** -- or **policy optimization.**

Optimal Policies

Values define a **partial ordering** over policies:

$$\pi \geq \pi' \iff v_\pi(s) \geq v'_\pi(s) \quad \forall s$$

Theorem:

For any Markov Decision Process

- There exists an **optimal policy** that is better or equal to all other policies
- There can be **more than one** such optimal policy
- They achieve the **same** optimal value function $v_{\pi^*}(s) = v_*(s)$
- They achieve the **same** optimal action-value function $q_{\pi^*}(s, a) = q_*(s, a)$

Deriving optimal policies

An **optimal policy** can be found by maximizing over $q_*(s,a)$

$$\pi_*(s,a) = \begin{cases} 1 & \text{if } \; a = \text{argmax}_a \; q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

There is always a **deterministic** optimal policy for any MDP

If multiple actions maximize q_* we can pick **any** of these (including stochastically)

Bellman Equations

There are four main Bellman equations:

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

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

prediction

$$v_*(s) = \max_a \; E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

$$q_*(s, a) = E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a]$$

control

Solving Bellman optimality equations

The Bellman equations for v^* and q^* are **non linear**

Cannot use the same (inefficient) matrix solution as for policy evaluation

Instead, **efficient iterative** solutions are available for both evaluation and control

Dynamic programming

- Value iteration, Policy iteration

TODAY

Sample methods

- Monte Carlo, Q-learning, Sarsa, ...

NEXT LECTURE

Dynamic Programming

I felt I had to shield the Air Force from the fact that I was doing mathematics. What title 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 time-varying. Let's take a word that has a precise meaning, dynamic, in the classical physical sense and that is impossible to use 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 not even a Congressman could object to.

– (slightly paraphrased) ***Richard Bellman***

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***

Prediction: Iterative Policy Evaluation

We start by discussing how to estimate values, that we observed must satisfy:

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

Key Idea:
turn equalities into updates

Algorithm

First: initialize v_0
e.g. set it to zero for all states

Then, iterate:

$$\forall s : \quad v_{k+1}(s) = E_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s]$$

Convergence

Does such policy evaluation algorithm **converge**?

- Yes, under mild assumptions (e.g., $\gamma < 1$ in the continuing case)

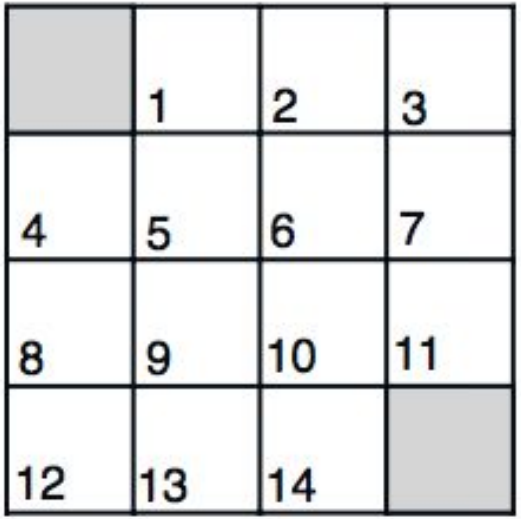
Simple proof-sketch:

$$\max_s |v_{k+1}(s) - v_\pi(s)| =$$
$$= \max_s |E_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] - E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]|$$
$$= \max_s |E_\pi[\gamma v_k(S_{t+1}) - \gamma v_\pi(S_{t+1})|S_t = s]|$$
$$= \gamma \max_s |E_\pi[v_k(S_{t+1}) - v_\pi(S_{t+1})|S_t = s]| \;\leq \gamma \max_s |v_k(s) - v_\pi(s)|$$

Hence in the limit $v_k \rightarrow v_\pi$

Example [Policy Evaluation]

actions



$$R_t = -1$$

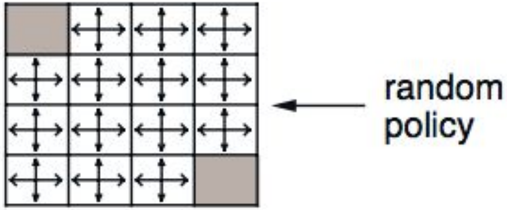
on all transitions

We will consider the undiscounted case for simplicity

Example [Policy Evaluation]

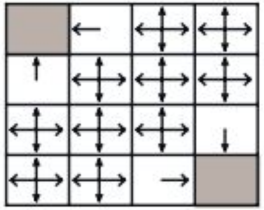
$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



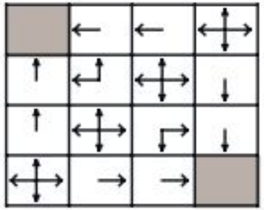
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



Example [Policy Evaluation]

$k = 3$

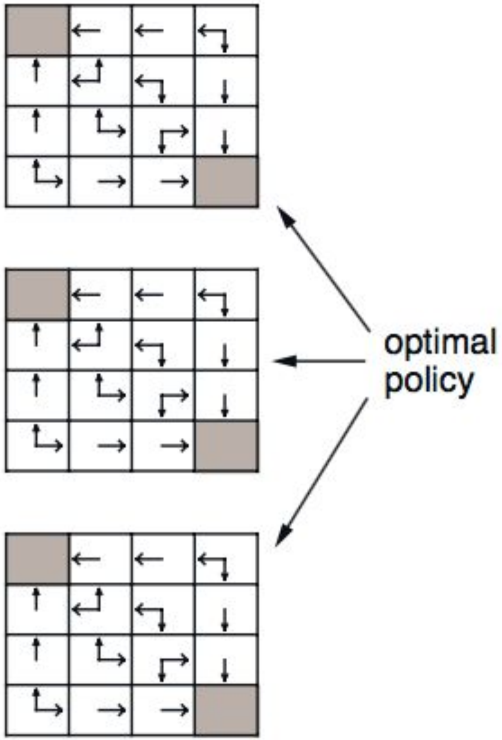
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



Policy Improvement

Acting greedily with respect to values of another policy is not optimal in general

Theorem: but the greedy policy wrt another policy values is a **policy improvement:**

$$\begin{aligned} \pi'(s) &= \text{argmax}_a q_\pi(s,a) \ \forall s \\ &= \text{argmax}_a E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \ \forall s \\ &\Rightarrow \ v_{\pi'}(s) \geq v_\pi(s) \ \forall s \end{aligned}$$

Note: if $v_{\pi'}(s) = v_\pi(s)$ then $v_{\pi'}(s) = \max_a E[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s]$

But that is the Bellman optimality equation!

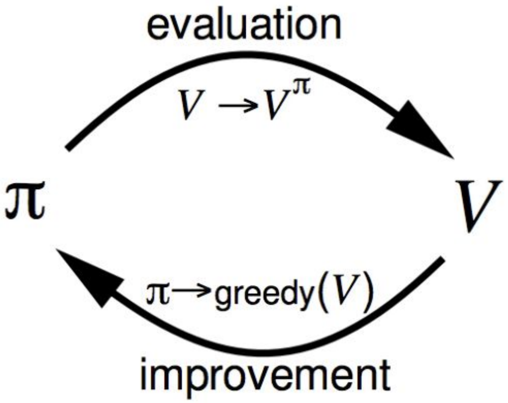
Hence, π' is either an **improvement** (when π' > π) or it is **optimal** (when π' = π)

Proof [Policy Improvement]

$$
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) q_\pi(s,a) \leq q_\pi(s, \pi'(s)) \\
&= E[R_{t+1} + v_\pi(S_{t+1}) | S_t = s, A_t \sim \pi'(s)] \\
&= E_{\pi'}[R_{t+1} + v_\pi(S_{t+1}) | S_t = s] \\
&\leq E_{\pi'}[R_{t+1} + q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= E_{\pi'}[R_{t+1} + \gamma E_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}] | S_t = s] \\
&= E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
&= E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&= \ldots \\
&= E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4}) + \ldots | S_t = s] = v_{\pi'}(s)
\end{aligned}
$$

Policy Iteration

This naturally suggests an **iterative** solution to **control** →



__**Algorithm:**__

First:

initialize $\pi(s)$

e.g. uniform random

Then, iterate:

$$v_\pi \leftarrow evaluate(\pi)$$

$$\pi(s) = \text{argmax}_a E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \ \forall s$$

Example [Jack's Car Rental]

States: two locations, max 20 cars at each

Actions: Move up to 5 cars overnight (-$2 each)

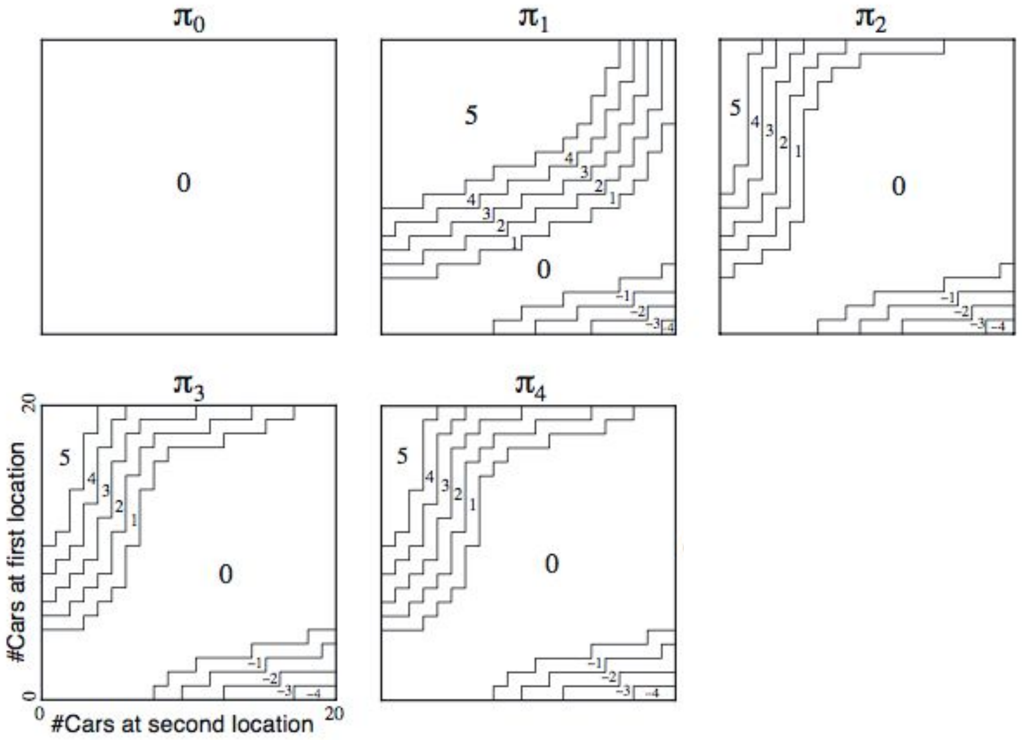
Reward: $10 for each available car rented

Transitions: Cars returned / requested with Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!}e^{-\lambda}$

- location 1: avg requests = 3, avg returns = 3
- location 2: avg requests = 4, avg returns = 2

Objective: $\gamma = 0.9$

Example [Jack's Car Rental]



Policy Iteration

Does policy evaluation need to converge fully to v_π ?

- Can we stop when we are **close**?
- E.g., with a threshold on the change to the values

Or should we simply stop after k iterations of policy evaluation?

- In the small gridworld $k = 3$ was sufficient to achieve optimal policy

Why not update policy **every** iteration — i.e. always stop after $k = 1$?

Value Iteration

This is equivalent to taking the Bellman **optimality equation** and taking that as update:

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

Or in the case of **action values**:

$$\forall s, a: \quad q_{k+1}(s, a) \leftarrow E[R_{t+1} + \gamma \max_{a'} q_k(S_{t+1}, a') | S_t = s, A_t = a]$$

Example [Shortest Path]

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Summary [Synchronous dynamic programming]

Problem	**Bellman Equation**	**Algorithm**
prediction	Expectation Equation	Policy Evaluation
control	Expectation Equation + Policy Improvement	Policy Iteration
control	Optimality Equation	Value Iteration

- **State-values complexity:** $O(mn^2)$ per iteration (for m actions and n states)
- **Action-values complexity:** $O(m^2n^2)$ per iteration

Asynchronous dynamic programming

Asynchronous algorithms back-up states **individually, in any order**

- can significantly reduce computation
- guaranteed to converge if all states continue to be selected

Three simple ideas in this space:

- **In-place** dynamic programming
- **Prioritised** sweeping
- **Real-time** dynamic programming

In-place dynamic programming

Synchronous value iteration stores two copies of the values

$$
\begin{aligned}
&\text{for } s \in S: \\
&\quad v_{new}(s) \leftarrow \max_a E_\pi[R_{t+1} + \gamma v_{old}(S_{t+1}) | S_t = s] \\
&v_{old} \leftarrow v_{new}
\end{aligned}
$$

In-place value iteration can be more **efficient** by always using the latest value estimate

$$
\begin{aligned}
&\text{for } s \in S: \\
&\quad v(s) \leftarrow \max_a E_\pi[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]
\end{aligned}
$$

Also saves **memory** by only storing one copy of value function.

Prioritised and real-time Sweeping

Can we **choose states** better? Use **magnitude of Bellman error** to guide state selection

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

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

Alternatively, in **real time DP** we use the agent **experience** to guide states selection

- If the agent is in state *S*, then update *S* or states easily reachable from S

Full width Back-ups

DP used **full-width, tabular** backups where for each backup (sync or async)

- Every successor state and action is considered (*full width*)
- A distinct value estimate is kept for each state (*tabular*)

Effective for medium-sized problems but suffers from **curse of dimensionality**

- Number of states $n = |S|$ grows exponentially with number of state variables
- Number of distinct values to estimate also grows exponentially

Key Ideas:

1. **Sample** updates instead of computing full expectations $\rightarrow$ **Lecture 4, 5**
2. **Approximate** the value function and generalize across states $\rightarrow$ **Lecture 8**

Approximate dynamic programming

Define a function approximator $v_\theta(s)$ with a parameter vector $\theta \in R^m$

- Use **dynamic programming** to construct a target $\tilde{v}_k(s)$ from $v_\theta(s)$
- Use gradient descent to update the parameters so as to minimize a loss

$$\sum_{s \in \tilde{S}} (v_\theta(s) - \tilde{v}_k(s))^2$$

Over some (subset?) of the states $\tilde{S} \in S$

For instance, in the case of **fitted value iteration** we minimize the loss with targets

$$\tilde{v}_k(s) = \max_a E_\pi[R_{t+1} + \gamma v_\theta(S_{t+1}) | S_t = s]$$

Bootstrapping

DP improves the value estimate at a state using the estimates at subsequent states

- This idea is **core** to RL — it is called **bootstrapping**
- Is this a sound thing to do? It depends...

There is a theoretical danger of **divergence** when combining

1. Bootstrapping
2. Function approximation
3. Updating values for a state distribution that doesn't match the MDP's dynamics

This theoretical danger is rarely encountered in practice

Questions?