Applied Computational Mathematics Lab

TEDDY LAZEBNIK'S COURSE · LECTURE 3

Reinforcement Learning

Lecture 3 · MDPs and dynamic programming: Markov decision processes, returns, value functions, Bellman equations, policy iteration and value iteration



Teddy Lazebnik · Applied Computational Mathematics Laboratory

BEFORE WE START

Recap, and what this lecture does

- RL is the science of learning to make decisions.
- Agents can learn a policy, a value function and/or a model.
- Decisions affect the reward, the agent state and the environment state.
- Last lecture: many actions but only **one state**, and no model.



IN PLAIN WORDS

This lecture brings back the full sequential problem and writes it down exactly: the Markov decision process. Then we solve it, assuming we are **given the true model**.

READING

Sutton & Barto (2018), chapters 3 and 4.

WHERE WE ARE GOING

The methods today, called **dynamic programming**, use the model. The next lectures keep the same ideas but replace the model with **samples**.

BEFORE WE START

Today's map

PART 1

Markov decision processes

The formal model: states, actions, dynamics, discount.

PART 2

Returns and discounting

What exactly the agent tries to maximise.

PART 3

Policies and value functions

v, q, optimal values and optimal policies.

PART 4

Bellman equations

Expectation and optimality equations, and the matrix form.

PART 5

Dynamic programming

Policy evaluation, policy improvement, policy iteration.

PART 6

Value iteration and beyond

Value iteration, asynchronous DP, and sample backups.



IN PLAIN WORDS

By the end you should be able to write down an MDP, its Bellman equations, and run policy evaluation, policy iteration and value iteration by hand on a small example.

Applied Computational Mathematics Lab

PART 1

Markov decision processes

A precise way to describe the agent–environment interaction.



Formalising the RL interface

- We describe the agent–environment interaction mathematically as a **Markov decision process** (MDP).
- This lets us talk clearly about the objective, and about how to achieve it.
- For now we assume the environment is **fully observable**: the current observation contains all relevant information.



IN PLAIN WORDS

An MDP is a contract between agent and environment: what can happen, with what probability, and what it is worth.

HOW GENERAL IS THIS?

Almost all RL problems can be written as MDPs:

- **Optimal control** deals mainly with continuous MDPs (robots, rockets).
- **Partially observable** problems can be turned into MDPs over belief states.
- **Bandits** are MDPs with a single state.

TRY IT

Chess: states are board positions, actions are legal moves, the reward is +1 for a win at the end. The dynamics include the opponent's replies.

Markov decision process: the definition

MARKOV DECISION PROCESS (SUTTON & BARTO, 2018)

A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, p, \gamma)$, where

- $\mathcal{S}$ is the set of all possible states,
- $\mathcal{A}$ is the set of all possible actions (for example, motor controls),
- $p(r, s' \mid s, a)$ is the joint probability of reward r and next state s', given state s and action a,
- $\gamma \in [0, 1]$ is a discount factor that weighs later rewards against earlier ones.

- p defines the **dynamics**. We can marginalise it when useful:

$$p(s' \mid s, a) = \sum_r p(s', r \mid s, a), \qquad \mathbb{E}[R \mid s, a] = \sum_r r \sum_{s'} p(r, s' \mid s, a)$$

TRY IT

From state s, action a leads to s_1 with reward 2 (probability 0.3), to s_1 with reward 0 (probability 0.2), or to s_2 with reward 1 (probability 0.5).
Then $p(s_1 \mid s, a) = 0.5$, $p(s_2 \mid s, a) = 0.5$, and $\mathbb{E}[R \mid s, a] = 0.3 \cdot 2 + 0.2 \cdot 0 + 0.5 \cdot 1 = 1.1$.



IN PLAIN WORDS

The tuple is the whole problem: what the agent can be in, what it can do, what happens next, and how much it cares about the future.

An equivalent definition

ALTERNATIVE DEFINITION

An MDP can also be written as $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$, where

- $p(s' \mid s, a)$ is the probability of moving to s' from s with action a,
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the **expected** reward of a transition starting in (s, a): $r(s, a) = \mathbb{E}[R \mid s, a]$,
- $\gamma \in [0, 1]$ is the discount factor.

- The two definitions are **equivalent**: neither adds assumptions. The second just keeps the parts that matter for expected returns.

	JOINT FORM	SPLIT FORM
Dynamics	$p(r, s' \mid s, a)$	$p(s' \mid s, a)$
Reward	inside p	$r(s, a)$, the mean
Use	samples, full detail	expected values, planning

TRY IT

In the example on the previous slide, the split form keeps $p(s_1 \mid s, a) = p(s_2 \mid s, a) = 0.5$ and $r(s, a) = 1.1$. That is all a planner needs to compute expected returns.

The Markov property

MARKOV PROPERTY

A state s has the Markov property when, for all next states s' and all possible histories h_{t-1},

$$p\left(S_{t+1}=s' \mid S_t=s\right)=p\left(S_{t+1}=s' \mid h_{t-1}, S_t=s\right)$$

- In an MDP, **all** states are assumed to be Markov.
- The state captures all relevant information from the history; once it is known, the history can be thrown away.
- The state is a **sufficient statistic** of the past.



IN PLAIN WORDS

The future is independent of the past, given the present.

MARKOV

A board game with dice: the board position and whose turn it is are Markov. How you got there doesn't change what the next roll can do.

NOT MARKOV, AND A FIX

A thermostat reading alone may not be Markov: whether the heater is warming up or cooling down depends on the recent past. Adding the last reading fixes it.

Test your understanding: Markov

CHECK YOURSELF

In an MDP, which of these statements follow from the Markov property? Here $\mathcal{H}$ stands for everything that happened before time t.

	STATEMENT
(a)	$p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid \mathcal{H}, S_t = s, A_t = a)$
(b)	$p(R_{t+1} = r, S_{t+1} = s' \mid S_t = s, A_t = a) = p(R_{t+1} = r, S_{t+1} = s' \mid \mathcal{H}, S_t = s, A_t = a)$
(c)	$p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid \mathcal{H}, S_t = s)$



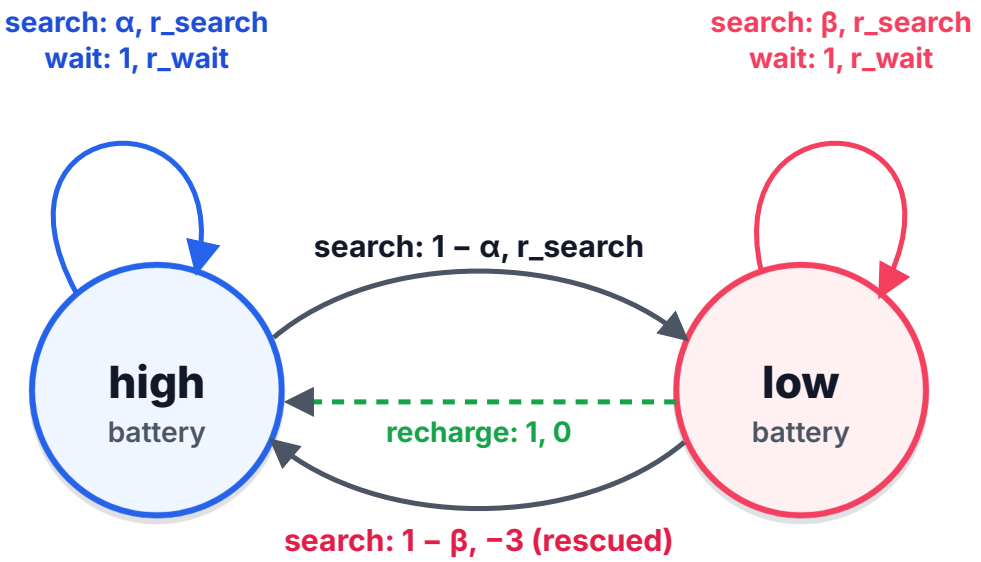
HINT

Look carefully at what each right-hand side conditions on. One of them quietly drops something important.

ANSWER

Answer at the end of the deck (question 3).

Example: a recycling robot



labels: action: probability, reward

Our drawing of the recycling robot (Sutton & Barto, 2018, example 3.3).

- A robot collects empty soda cans.
- Two states: **high** or **low** battery.
- Actions: search or wait when high; search, wait or recharge when low.
- Dynamics can be stochastic: searching on a high battery keeps it high with probability α.
- Searching on a low battery may run it flat: the robot is rescued and gets -3.
- The reward can be the expected number of cans (deterministic) or the actual number (stochastic).

The robot MDP in numbers

STATE	ACTION	NEXT STATE: PROBABILITY	REWARD
high	search	high: α, low: 1 − α	r_search
high	wait	high: 1	r_wait
low	search	low: β, high: 1 − β	r_search, or −3 if rescued
low	wait	low: 1	r_wait
low	recharge	high: 1	0

ILLUSTRATIVE NUMBERS

We pick $\alpha = 0.8$, $\beta = 0.6$, $r_{\text{search}} = 2$, $r_{\text{wait}} = 1$ and $\gamma = 0.9$. These numbers are ours, for illustration.

STATE	ACTION	$q_*(s, a)$
high	search	**16.95**
high	wait	16.25
low	search	14.34
low	wait	14.73
low	recharge	**15.25**

Our computation (value iteration). $v_*(\text{high}) = 16.95$, $v_*(\text{low}) = 15.25$.



IN PLAIN WORDS

Best plan: search while the battery is high; recharge as soon as it is low. Searching on a low battery risks the −3 rescue.

Applied Computational Mathematics Lab

PART 2

Returns and discounting

What exactly the agent is trying to maximise.



Three kinds of return

- Acting in an MDP gives rewards $R_{t+1}, R_{t+2}, \ldots$. We summarise them as a return G_t:

RETURN	FORMULA	USED FOR
Undiscounted	$G_t = \sum_{k=0}^{T-t-1} R_{t+k+1}$	episodic, finite horizon
Discounted	$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$	finite or infinite horizon
Average	$G_t = \frac{1}{T-t-1} \sum_{k=0}^{T-t-1} R_{t+k+1}$	continuing, infinite horizon

- Returns are **random variables**: they depend on the MDP and the policy.

TRY IT

Rewards after time t: 1, 0, 2, 4, then the episode ends.
Undiscounted: $1 + 0 + 2 + 4 = 7$.
Discounted, $\gamma = 0.5$: $1 + 0 + 0.25 \cdot 2 + 0.125 \cdot 4 = 2$.
Average: $7/4 = 1.75$.



IN PLAIN WORDS

Different returns encode different goals. Choosing one is part of defining the problem.

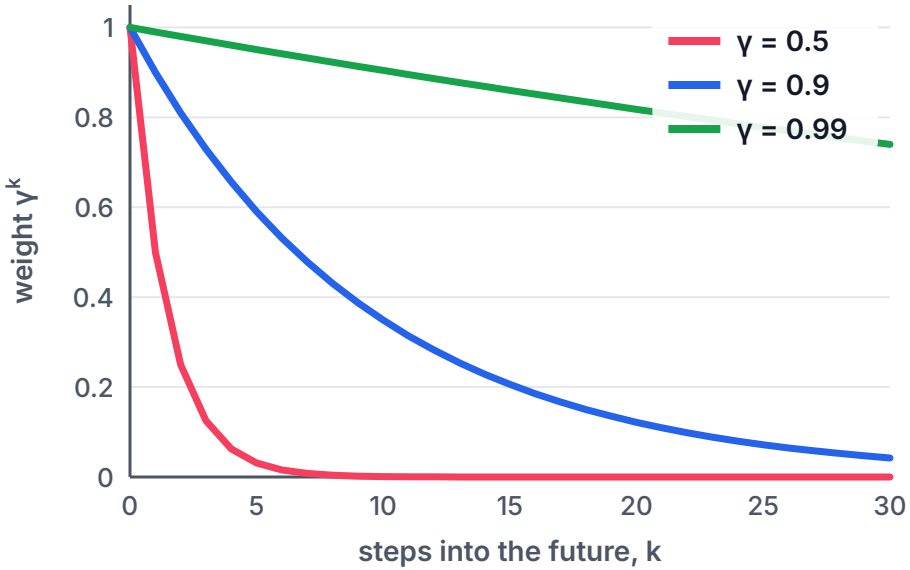
The discounted return

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

- γ is the present value of future rewards: a reward R received after $k+1$ steps is worth $\gamma^k R$ now.
- For $\gamma < 1$, immediate rewards matter more than delayed ones.
- γ close to 0: **myopic** evaluation. γ close to 1: **far-sighted** evaluation.

TRY IT

Reward +1 on every step, forever: $G = 1/(1-\gamma)$. That is 2 for $\gamma = 0.5$, 10 for 0.9, and 100 for 0.99. A rough rule: γ looks about $1/(1-\gamma)$ steps ahead.



How much a reward k steps after the next one counts. With $\gamma = 0.5$ it has almost vanished after 5 steps; with 0.99 it barely fades in 30.

Why discount?

THE PROBLEM SIDE

It can be part of the goal

Money earns interest, so money now is worth more than money later. Animals and people prefer immediate rewards.

THE SOLUTION SIDE

It makes the maths work

Discounting is convenient, and it avoids infinite returns in processes that cycle forever.



IN PLAIN WORDS

Reward and discount together define the goal. Changing γ can change which behaviour is best.

TRY IT

A process that gives +1 on every step, forever. Undiscounted, the return is infinite. With $\gamma = 0.9$ it is $1 + 0.9 + 0.81 + \cdots = 1/(1 - 0.9) = 10$.

INTEREST AS DISCOUNTING

A bank pays 5% interest per year. Receiving 100 next year is worth about $100/1.05 \approx 95$ today: a yearly discount of about 0.95.

When γ changes the answer

- Two choices from the same state:
- **Option A**: reward +1 now, then nothing.
- **Option B**: nothing for one step, then +2, then nothing.

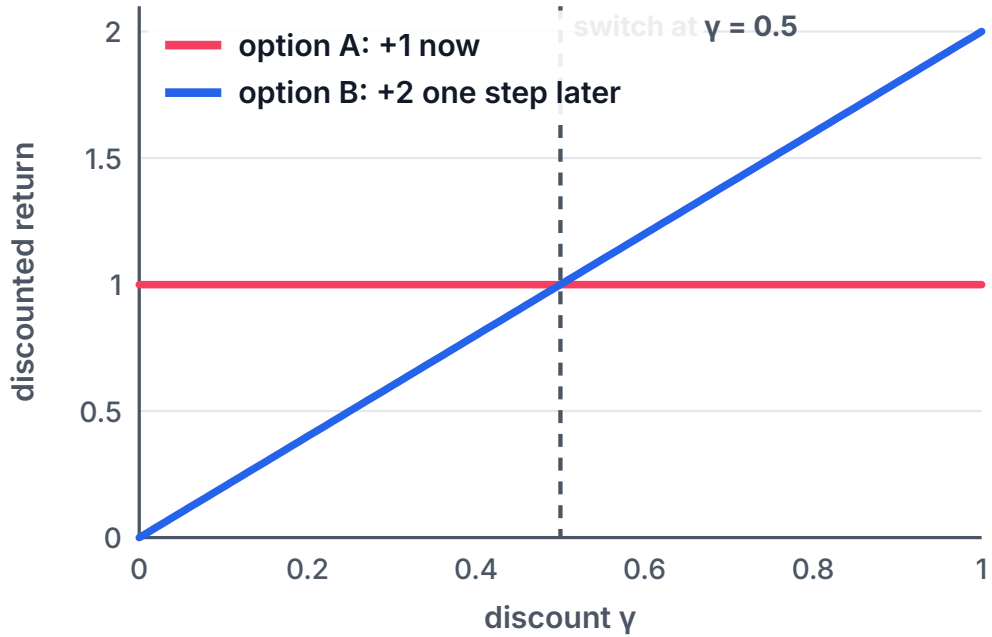
$$G^A = 1, \qquad G^B = \gamma \cdot 2$$

- B is better when $2\gamma > 1$, that is when $\gamma > 0.5$.



IN PLAIN WORDS

A patient agent (γ near 1) waits for the bigger reward; an impatient one (γ near 0) grabs what is in front of it.



Our example. The best choice flips at $\gamma = 0.5$.

Check yourself: Parts 1 and 2

CHECK YOURSELF

1. Name the four parts of an MDP and say what each one means.

CHECK YOURSELF

2. A transition gives reward 3 with probability 0.4 and reward −1 otherwise. What is $r(s, a)$?

CHECK YOURSELF

3. Which of statements (a), (b), (c) on the “test your understanding” slide are true?

CHECK YOURSELF

4. Rewards 2, 2, 2 then the episode ends. Compute the discounted return with $\gamma = 0.5$.





HINT

Answers are at the end of the deck.

Applied Computational Mathematics Lab

PART 3

Policies and value functions

How an agent behaves, how good that is, and what the best possible is.



Policies

GOAL

The goal of an RL agent is to find a behaviour policy that maximises the expected return G_t.

- A policy is a mapping $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$: for every state s it gives the probability $\pi(a \mid s)$ of each action.
- For a deterministic policy we write $a_t = \pi(s_t)$.

STATE	$\pi(\text{search} \mid s)$	$\pi(\text{wait} \mid s)$	$\pi(\text{recharge} \mid s)$
high	0.7	0.3	–
low	0.1	0.2	0.7

A stochastic policy for the robot: each row sums to 1.

STATE	$\pi(s)$
high	search
low	recharge

A deterministic policy: one action per state.

Value functions

- The state-value function gives the long-term value of state s under policy π:

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

- The action-value function also fixes the first action:

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

- They are connected: the state value is the policy-weighted average of the action values.

$$v_\pi(s) = \sum_a \pi(a \mid s)\, q_\pi(s, a) = \mathbb{E}\left[q_\pi(S_t, A_t) \mid S_t = s, \pi\right]$$

TRY IT

In some state, π picks left with probability 0.25 and right with 0.75. If $q_\pi(s, \text{left}) = -2$ and $q_\pi(s, \text{right}) = 4$, then $v_\pi(s) = 0.25(-2) + 0.75(4) = 2.5$.



IN PLAIN WORDS

v asks "how good is it to be here, if I keep following π?". q asks the same after one chosen first move.

Optimal value functions

OPTIMAL VALUE FUNCTIONS

The optimal state-value function is the maximum value over all policies, and likewise for action values:

$$v_*(s) = \max_\pi v_\pi(s), \qquad q_*(s,a) = \max_\pi q_\pi(s,a)$$

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

TRY IT

For the robot with our numbers: $v_*(\text{high}) = 16.95$ and $v_*(\text{low}) = 15.25$. No policy can do better from either state.



IN PLAIN WORDS

v_* is a ceiling: it tells you how good things could possibly get, before you know how to get there.

Optimal policies

- Define a partial order on policies:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s$$

THEOREM (OPTIMAL POLICIES)

For any MDP:

- there is an optimal policy π_* that is better than or equal to all others, $\pi_* \geq \pi$ for all π (there may be several);
- every optimal policy achieves $v_{\pi_*}(s) = v_*(s)$ and $q_{\pi_*}(s, a) = q_*(s, a)$.

POLICY	$v(s_1)$	$v(s_2)$
π_1	5	2
π_2	3	4
π_*	**5**	**4**

Illustration: π_1 and π_2 can't be compared, since each wins in one state. The theorem says a single policy exists that is best in **every** state at once.



IN PLAIN WORDS

It is "partial" because some pairs of policies can't be ranked. Remarkably, the top always exists.

Finding an optimal policy

- An optimal policy can be read off q_*:

$$\pi_*(a \mid s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{b \in \mathcal{A}} q_*(s, b) \\ 0 & \text{otherwise} \end{cases}$$

- Every MDP has a **deterministic** optimal policy.
- If we know q_*, we immediately have an optimal policy.
- There can be several: if several actions maximise $q_*(s, \cdot)$, pick any of them, even at random.

STATE	ACTION	q_*
high	search	**16.95**
high	wait	16.25
low	search	14.34
low	wait	14.73
low	recharge	**15.25**

The robot again: in each state, take the action with the largest q_*.



IN PLAIN WORDS

Knowing q_* is knowing what to do: no look-ahead, no model needed, just a lookup and a max.

Applied Computational Mathematics Lab

PART 4

Bellman equations

Writing values recursively, and solving the equations.



The value function, recursively

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}\left[G_t \mid S_t = s, \pi\right] = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, \pi\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)\right] \\
&= \sum_a \pi(a \mid s) \sum_r \sum_{s'} p(r, s' \mid s, a)\left(r + \gamma v_\pi(s')\right)
\end{aligned}
$$

- The last line writes the expectation out in full: average over the policy's action, then over what the environment does.

TRY IT

A state with one action. It leads to s_1 with reward 1 (probability 0.5) or to s_2 with reward 0 (probability 0.5). With $\gamma = 0.9$, $v(s_1) = 10$ and $v(s_2) = 0$:
$v(s) = 0.5(1 + 9) + 0.5(0 + 0) = 5.$



IN PLAIN WORDS

Today's value is tomorrow's reward plus tomorrow's value, averaged over everything that might happen.

Action values, recursively

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

- And as before, $v_\pi(s) = \sum_a \pi(a \mid s)\, q_\pi(s,a)$.

TRY IT

The robot, with the optimal values $v(\text{high}) = 16.95$ and $v(\text{low}) = 15.25$:
$q(\text{high}, \text{search}) = 0.8(2 + 0.9 \cdot 16.95) + 0.2(2 + 0.9 \cdot 15.25) = 16.95$.



IN PLAIN WORDS

q starts one step earlier than v: first the chosen action, then the policy takes over.

WHICH ONE TO USE

Compute q when you need to compare actions; compute v when you only need to know how good a state is. Each can be built from the other plus one step of the model.

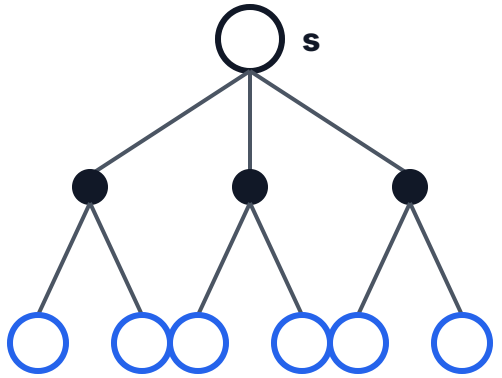
The Bellman expectation equations

THEOREM (BELLMAN EXPECTATION EQUATIONS)

Given an MDP $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, for any policy π the value functions satisfy

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

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



every action, every next state

The backup behind the equation for v_π: average over the actions (weights π), then over the next states (weights p).



IN PLAIN WORDS

These are just the recursive definitions with the split-form model plugged in.

The Bellman optimality equations

THEOREM (BELLMAN OPTIMALITY EQUATIONS)

The optimal value functions satisfy

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

$$q_*(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} q_*(s',a')$$

- No policy can do better than $v_*(s) = \max_\pi v_\pi(s)$ in any state.

WHERE IT COMES FROM

Plug the optimal (greedy) policy into the expectation equation for q: the average over $\pi_*(a' \mid s')$ picks out the best action, so $\sum_{a'} \pi_*(a' \mid s') q_*(s',a') = \max_{a'} q_*(s',a')$. That gives the optimality equation.



IN PLAIN WORDS

The only change from the expectation equations: the average over the policy's action becomes a **max**.

Prediction and control

PROBLEM 1

Prediction (policy evaluation)

Estimate v_π or q_π. Given this behaviour, what return do I expect?

PROBLEM 2

Control

Estimate v_* or q_*, and use them to find the best behaviour.



IN PLAIN WORDS

Prediction judges a fixed plan. Control searches for the best plan. Most control methods use prediction as an inner step.

PREDICTION EXAMPLES

Given this treatment protocol, or this trading strategy, what is my expected outcome?

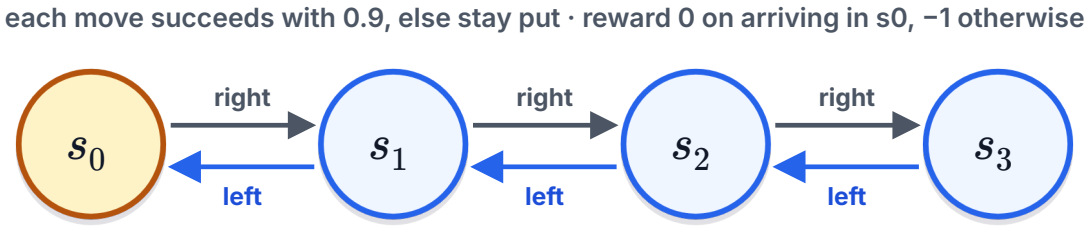
CONTROL EXAMPLES

What is the best treatment, or the control policy that minimises time or fuel?

BOTH, TODAY

Dynamic programming today solves both, given the model. Later lectures solve them from experience.

Exercise: a small MDP



- Four states in a row. Actions: move left or move right.
- Each move succeeds with probability 0.9; with 0.1 the agent stays where it is.
- Reward 0 for every transition that ends in s_0, and -1 for every other transition.

PAUSE AND WORK IT OUT

With $\gamma = 0.9$:

- What is v_π for "always move left"?
- What is v_π for the uniformly random policy?
- What changes with $\gamma = 0$?



HINT

This is our own small example. Try writing the Bellman equation for each state first.

Exercise: a solution

POLICY, γ	s_0	s_1	s_2	s_3
always left, 0.9	0.0	-0.11	-1.2	-2.16
random, 0.9	-6.57	-7.08	-7.98	-8.38
always left, 0	0.0	-0.1	-1.0	-1.0
random, 0	-0.45	-0.55	-1.0	-1.0

Our computation, solving the Bellman equations exactly.

- Moving left is far better than wandering at random.
- With $\gamma = 0$, only the next reward counts: s_2 and s_3 look equally bad, because the agent is blind to how far away the goal is.

TRY IT

Always left, $\gamma = 0$: from s_1 the move succeeds with 0.9 (reward 0, landing in s_0) and fails with 0.1 (reward −1). So $v(s_1) = 0.9 \cdot 0 + 0.1 \cdot (-1) = -0.1$.



IN PLAIN WORDS

A discount of 0 makes the agent short-sighted: it can't tell "one step from the goal" from "three steps".

The Bellman equation in matrix form

- For a given π, stack the values of all states into a vector:

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

- $v_i = v(s_i)$, $r_i^\pi = \mathbb{E}[R_{t+1} \mid S_t = s_i, A_t \sim \pi(S_t)]$, and $P_{ij}^\pi = \sum_a \pi(a \mid s_i)\, p(s_j \mid s_i, a)$.
- It is a linear equation, so it can be solved directly:

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

THE EXERCISE, AS MATRICES

"Always left" on the exercise chain: $\mathbf{r}^\pi = (0, -0.1, -1, -1)$ and P^π has 0.9 on the cell to the left and 0.1 on the diagonal (s_0 stays put with 1). Solving gives $\mathbf{v} = (0.0, -0.11, -1.2, -2.16)$.

P^π	s_0	s_1	s_2	s_3
s_0	**1.0**	0	0	0
s_1	**0.9**	**0.1**	0	0
s_2	0	**0.9**	**0.1**	0
s_3	0	0	**0.9**	**0.1**

The transition matrix of "always left": each row is where you go from that state.



IN PLAIN WORDS

One linear solve gives all the values at once.

Solving it directly: the cost

- Solving $(I - \gamma P^{\pi})\mathbf{v} = \mathbf{r}^{\pi}$ costs about $O(|\mathcal{S}|^3)$ operations.
- That is fine for small problems, and impossible for large ones.
- Iterative methods scale better:
  - dynamic programming (this lecture),
  - Monte Carlo evaluation (Lecture 5),
  - temporal-difference learning (Lecture 5).

STATES n	ABOUT n^3 OPERATIONS	AT A BILLION PER SECOND
100	10^6	a millisecond
10,000	10^{12}	about 17 minutes
1,000,000	10^{18}	about 32 years

Rough arithmetic, to show the growth.



IN PLAIN WORDS

A Go board alone has more positions than atoms in the universe. Direct solving is out of the question there.

Solving the optimality equation

- The Bellman optimality equation is **non-linear**, because of the max.
- So we can't use the direct matrix solution in general.
- Instead there are many **iterative** methods:

WITH A MODEL

Dynamic programming

Value iteration and policy iteration: this lecture.

WITH SAMPLES

Learning from experience

Monte Carlo, Q-learning, SARSA: Lectures 5 and 6.



IN PLAIN WORDS

Expectation equations are linear (solve them like a system of equations). Optimality equations have a max, so we iterate instead.

WHY NON-LINEAR

A max of straight lines is a bent line: $\max(x, 2 - x)$ has a kink at $x = 1$. No single matrix can represent that, which is why we iterate.

Check yourself: Parts 3 and 4

CHECK YOURSELF

5. π picks a_1 with probability 0.4 and a_2 with 0.6. If $q_\pi(s, a_1) = 10$ and $q_\pi(s, a_2) = 5$, what is $v_\pi(s)$?

CHECK YOURSELF

6. What is the difference between the Bellman expectation and optimality equations?

CHECK YOURSELF

7. Why can't we solve the optimality equation with one matrix inversion?

CHECK YOURSELF

8. Given q_*, how do you get an optimal policy? Is it unique?



Applied Computational Mathematics Lab

PART 5

Dynamic programming

Policy evaluation, policy improvement and policy iteration, using a model.



What is dynamic programming?

DYNAMIC PROGRAMMING

A collection of algorithms that compute optimal policies, given a **perfect model** of the environment as an MDP (after Sutton & Barto, 2018).

- All the methods we discuss have two parts: **policy evaluation** and **policy improvement**.
- "Programming" here means planning, as in a TV programme schedule, not writing code.

WHERE THE NAME COMES FROM (PARAPHRASED)

Richard Bellman later explained, in his autobiography (1984), that he chose the name in the 1950s partly to describe planning over time, and partly because nobody could object to it: it sounded impressive and hid the fact that his funders were paying for mathematics.



IN PLAIN WORDS

Dynamic: things change over time.
Programming: planning a sequence of decisions.

Policy evaluation

- We want $v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid s, \pi]$.
- Idea: turn the equality into an **update**.

ITERATIVE POLICY EVALUATION

1. Initialise v_0, for example to zero.
2. Repeat for all states: $v_{k+1}(s) \leftarrow \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid s, \pi]$.
3. Stop when $v_{k+1}(s) = v_k(s)$ for all s: then we have found v_π.

- Does it always converge? Yes, under suitable conditions such as $\gamma < 1$. Lecture 4 proves it.



IN PLAIN WORDS

Each sweep pushes value information one step further back from the rewards, until nothing changes.

TRY IT

On the exercise chain, "always left", $\gamma = 0.9$, start at 0. After one sweep: $v_1(s_1) = -0.1$, $v_1(s_2) = -1$, $v_1(s_3) = -1$. After many sweeps it settles at the exact values $(0.0, -0.11, -1.2, -2.16)$.

Example: evaluating a random walk

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

v_0

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

v_1

0.0	−1.8	−2.0	−2.0
−1.8	−2.0	−2.0	−2.0
−2.0	−2.0	−2.0	−1.8
−2.0	−2.0	−1.8	0.0

v_2

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

v_3

- A 4×4 grid. The two grey corners end the episode; every other move costs −1.
- The policy picks one of four directions at random; moving into the edge leaves you in place.
- Values after 0, 1, 2 and 3 sweeps of policy evaluation ($\gamma = 1$).



IN PLAIN WORDS

After one sweep, every state knows it costs at least 1. After more sweeps, states far from the corners discover they are expensive.

CREDIT

This gridworld is example 4.1 of Sutton & Barto (2018). All values here are from our own run.

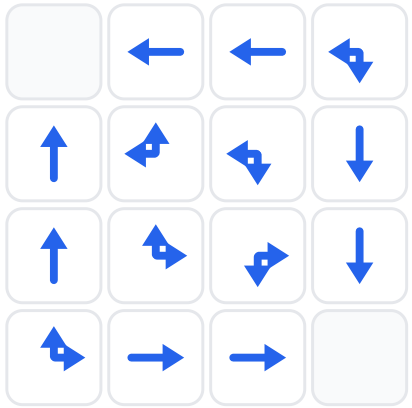
Evaluation, then greedy improvement

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

v_{10}

0.0	−14.0	−20.0	−22.0
−14.0	−18.0	−20.0	−20.0
−20.0	−20.0	−18.0	−14.0
−22.0	−20.0	−14.0	0.0

v_π (converged)



greedy on v_π

- The converged values of the **random** policy.
- Acting greedily on them already gives an optimal policy here: every arrow heads for the nearest corner.



IN PLAIN WORDS

Even the values of a terrible policy carry useful information: they tell you which neighbours are closer to the goal.

TRY IT

Greedy on v_3 already gives the same arrows (our run). Evaluating the random policy fully was not needed to find the best one.

CAREFUL

That greedy on the random policy is optimal is luck of this example, not true in general.

Policy improvement

- Use the evaluation to improve the policy. Act greedily:

$$\pi_{\text{new}}(s) = \operatorname*{argmax}_a q_\pi(s, a)$$
$$= \operatorname*{argmax}_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right]$$

- Then evaluate π_{new} and repeat.
- Claim: $v_{\pi_{\text{new}}}(s) \geq v_\pi(s)$ for all s. The new policy is never worse.

TRY IT

The robot under "always wait" has $v = 10$ in both states. Then $q(\text{high}, \text{search}) = 2 + 0.9 \cdot 10 = 11 > q(\text{high}, \text{wait}) = 10$, so improvement switches to searching when high. When low: search 9.0, wait 10, recharge 9.0, so it keeps waiting.

WHY IT NEVER GETS WORSE (SKETCH)

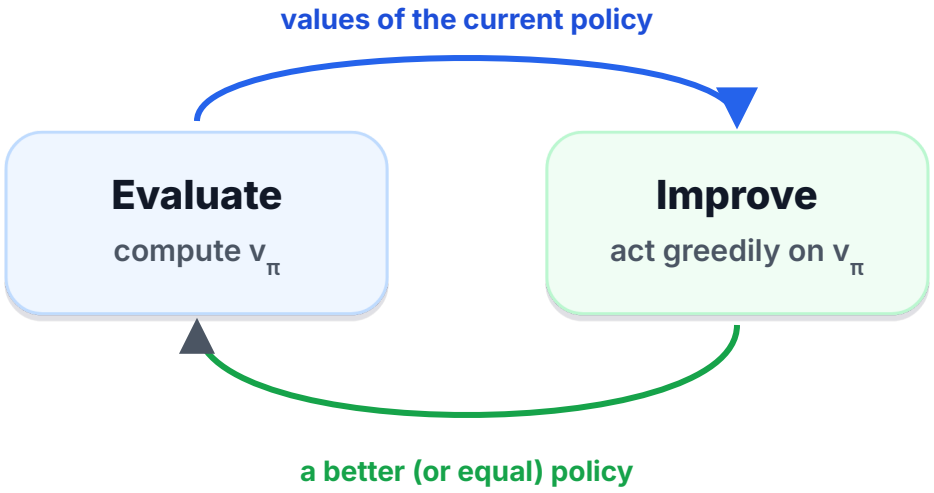
In state s, greedy picks a with $q_\pi(s, a) \geq v_\pi(s)$: taking a once and then following π is at least as good as following π. Doing this in every state, again and again, can only help: $v_\pi(s) \leq q_\pi(s, \pi_{\text{new}}(s)) \leq \cdots \leq v_{\pi_{\text{new}}}(s)$.



IN PLAIN WORDS

If greedy improvement changes nothing, the Bellman optimality equation holds, and the policy is optimal.

Policy iteration



POLICY ITERATION

1. Start with any policy π_0.
2. **Evaluate**: compute v_{π_k}.
3. **Improve**: $\pi_{k+1}(s) = \operatorname{argmax}_a q_{\pi_k}(s, a)$.
4. Stop when the policy no longer changes.



IN PLAIN WORDS

Evaluate, improve, evaluate, improve. Each round gives a policy at least as good as the last.

WHY IT TERMINATES

A finite MDP has finitely many deterministic policies, and each round strictly improves until it stops. So policy iteration ends at an optimal policy after finitely many rounds.

THE BIG PICTURE

Most RL algorithms can be read as some version of this loop: evaluate a little, improve a little.

Example: Jack's car rental

- Two rental locations, at most 20 cars each.
- Overnight, Jack can move up to 5 cars between them, at $2 per car.
- Reward: $10 for each car rented. $\gamma = 0.9$.
- Requests and returns are random (Poisson):
  - location 1: on average 3 requests and 3 returns a day;
  - location 2: on average 4 requests and 2 returns a day.
- States: cars at each location. Actions: net cars moved, from −5 to +5.

CREDIT

This is example 4.2 of Sutton & Barto (2018). We re-ran policy iteration ourselves.

TRY IT

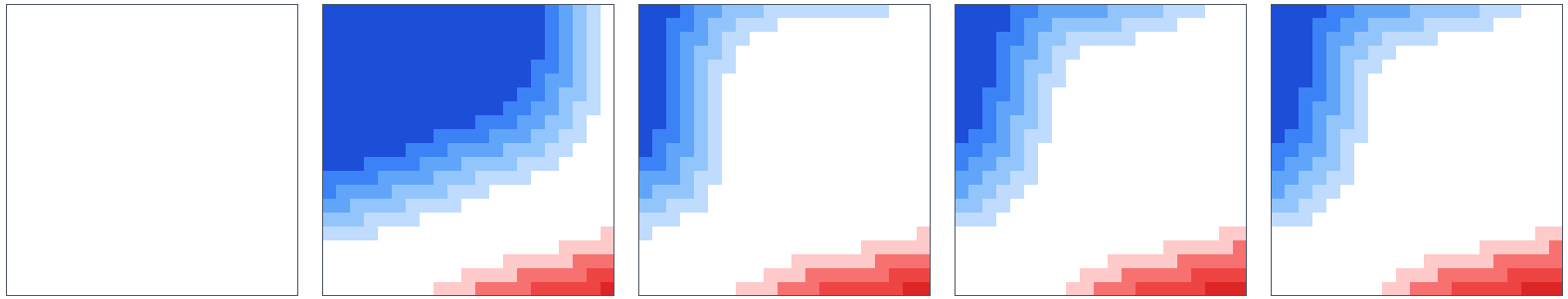
In state (15 cars, 5 cars), after one evaluation of "never move", the action values are 0: 519, 2: 530, 5: 541 (moving 0, 2 or 5 cars to location 2). Moving cars where demand is higher pays, even at $2 each.



IN PLAIN WORDS

Location 2 rents more and gets fewer returns, so Jack should move cars there.

Jack’s car rental: policy iteration, our run



π_0 π_1 π_2 π_3 π_4

Each square map: cars at location 1 (up) against cars at location 2 (right). Colour: cars moved from location 1 to 2 (blue, up to +5) or back (red, down to −5); white means none.

- Policy iteration stops after 5 policies. Evaluation sweeps per round: 126, 91, 80, 45, 14.
- Optimal values range from about $421 to $637.



IN PLAIN WORDS

Each round the policy gets better. The first evaluation needs many sweeps; later ones start close to the answer and need fewer.

WHY THE SWEEPS SHRINK

Later rounds evaluate a policy that is already close to the previous one, so the values barely move.

Do we need to evaluate fully?

- Does policy evaluation need to converge to v_π before improving?
- Or should we stop when we are "close", for example when the values change by less than a threshold?
- Or simply stop after k sweeps?
- In the small gridworld, $k = 3$ sweeps were enough for the optimal policy.
- The extreme: improve the policy after **every** sweep, $k = 1$. That is **value iteration**.

SWEEPS PER EVALUATION	NAME
until converged	policy iteration
a few, k	modified policy iteration
exactly 1	value iteration



IN PLAIN WORDS

Full evaluation is often a waste: you only need values good enough to pick the right greedy action.

Applied Computational Mathematics Lab

PART 6

Value iteration and beyond

One sweep, then improve; and smarter ways to sweep.



Value iteration

- Turn the Bellman **optimality** equation into an update:

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

- This is policy iteration with $k = 1$ sweep of evaluation between greedy improvements.

VALUE ITERATION

1. Initialise v_0.
2. Update every state with the max over actions.
3. Stop when $v_{k+1}(s) = v_k(s)$ for all s: then we have v_*.



IN PLAIN WORDS

No explicit policy is stored: the max inside the update plays the role of improvement.

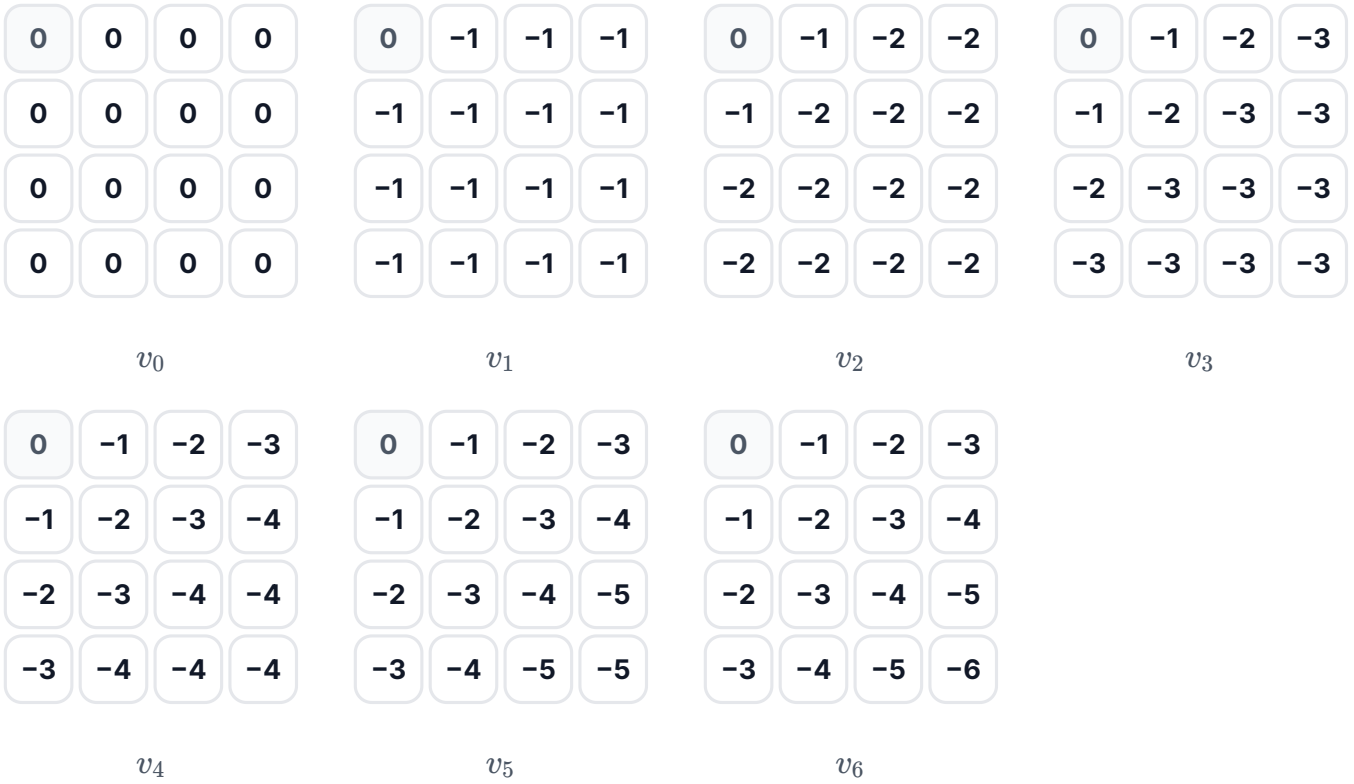
k	0	1	2	3	7	∞
high	0	2	3.62	4.95	9.08	16.95
low	0	1	1.9	3.26	7.38	15.25

Value iteration on the robot (our run).

TRY IT

The robot: start at 0. One update gives $v_1(\text{high}) = \max(2, 1) = 2$ and $v_1(\text{low}) = \max(0.6 \cdot 2 + 0.4 \cdot (-3), 1, 0) = 1$. Repeating converges to 16.95 and 15.25.

Example: shortest path



- A 4×4 grid; the goal is the top-left corner. Every step costs −1.
- Each sweep of value iteration lets the goal's information travel one more step.
- After 6 sweeps (v_6) every square knows its distance: up to 6 steps.



IN PLAIN WORDS

Value iteration discovers distances by spreading outward from the goal, like ripples in a pond.

Synchronous dynamic programming

PROBLEM	BELLMAN EQUATION	ALGORITHM
Prediction	expectation equation	iterative policy evaluation
Control	expectation equation + greedy improvement	policy iteration
Control	optimality equation	value iteration

- Based on $v_\pi(s)$ or $v_*(s)$: about $O(|\mathcal{A}||\mathcal{S}|^2)$ per sweep.
- Based on $q_\pi(s,a)$ or $q_*(s,a)$: about $O(|\mathcal{A}|^2|\mathcal{S}|^2)$ per sweep.

TRY IT

A problem with 1,000 states and 4 actions: one sweep over v costs about $4 \times 1000^2 = 4$ million operations; over q, about 16 million.



IN PLAIN WORDS

“Synchronous” means every state is updated in each sweep, using the old values of all the others.

Asynchronous dynamic programming

- So far, updates were **synchronous**: all states in parallel.
- Asynchronous DP updates states one at a time, in any order:
  - it can save a lot of computation,
  - it still converges if every state keeps being selected.
- Three simple ideas: in-place DP, prioritised sweeping, and real-time DP.

IN-PLACE VALUE ITERATION

1. Keep **one** copy of the values (not old and new).
2. For each s: $v(s) \leftarrow \max_a \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$, using the newest values immediately.

SWEEP ORDER	SWEEPS NEEDED
synchronous	10
in place, starting next to the goal	**1**
in place, starting far from the goal	10

Our check on a corridor of 10 squares with the goal at one end and −1 per step. Order matters when updating in place.



IN PLAIN WORDS

Using fresh values right away lets information travel further in a single sweep, if the order is good.

Prioritised sweeping and real-time DP

PRIORITISED SWEEPING

Update where it matters most

Use the size of the Bellman error, $|\max_a \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] - v(s)|$, to choose the next state. Back up the largest, then update the errors of its predecessors (this needs the reverse dynamics). A priority queue makes it efficient.

REAL-TIME DP

Update where the agent is

Only update states that matter to the agent now: the state it is in, or the states it expects to visit soon.



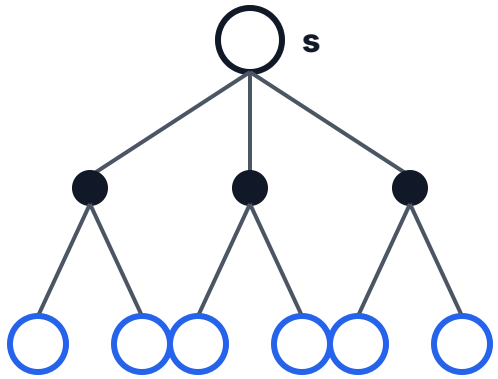
IN PLAIN WORDS

Sweeping a whole state space evenly is wasteful. Focus effort where values are wrong, or where the agent actually goes.

REAL-TIME DP, IN WORDS

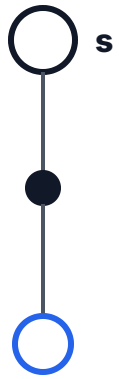
A robot planning its way across a building doesn't need accurate values for rooms it will never enter.

Full-width and sample backups



every action, every next state

Full-width backup: dynamic programming.



one sampled action and next state

Sample backup: next lectures.

- Standard DP considers **every** action and successor, using the true model. It works for medium problems (millions of states).
- For large problems it suffers from the **curse of dimensionality**: the number of states grows exponentially with the number of state variables.

NEXT LECTURES

Sample backups use sampled transitions $\langle s, a, r, s' \rangle$ instead of the model:

- model-free: no advance knowledge of the MDP needed;
- they break the curse of dimensionality through sampling;
- each backup has a constant cost, independent of $|\mathcal{S}|$.

TRY IT

A robot state with 10 variables, each taking 10 values, has 10^{10} states. One full sweep would visit ten billion states.

Check yourself: Parts 5 and 6

CHECK YOURSELF

9. What are the two steps of policy iteration, and why does it never get worse?

CHECK YOURSELF

10. How is value iteration related to policy iteration?

CHECK YOURSELF

11. In the shortest-path example, how many sweeps does a square 4 steps from the goal need before its value is correct?

CHECK YOURSELF

12. Give one advantage of sample backups over full-width backups.



WRAP-UP

Summary

- **MDPs** formalise sequential decisions: states, actions, dynamics, discount.
- **Returns**: undiscounted, discounted or average; reward and discount define the goal.
- **Value functions** v_π, q_π and the optimal v_*, q_*; greedy on q_* is optimal.
- **Bellman equations**: expectation (linear) and optimality (with a max).
- **Dynamic programming** solves MDPs with a model: policy evaluation, policy iteration, value iteration.

ALGORITHM	EQUATION	GIVES
policy evaluation	expectation	v_π
policy iteration	expectation + greedy	π_*, v_*
value iteration	optimality	v_*, then π_*



IN PLAIN WORDS

Next lectures keep these ideas but replace the model with samples.

ANSWERS

Answers to check yourself

- **1.** States $\mathcal{S}$, actions $\mathcal{A}$, dynamics $p(r, s' \mid s, a)$, discount γ.
- **2.** $0.4 \cdot 3 + 0.6 \cdot (-1) = 0.6$.
- **3.** (a) and (b) are true. (c) is false: it drops A_t on the right, and the next state depends on the action.
- **4.** $2 + 0.5 \cdot 2 + 0.25 \cdot 2 = 3.5$.
- **5.** $0.4 \cdot 10 + 0.6 \cdot 5 = 7$.
- **6.** The expectation equation averages over the policy's actions; the optimality equation takes the max.
- **7.** The max makes it non-linear.
- **8.** Act greedily: $\pi_*(s) = \operatorname{argmax}_a q_*(s, a)$. Not necessarily unique when actions tie.
- **9.** Evaluate, then improve greedily; acting greedily on q_π is at least as good as π in every state.
- **10.** It is policy iteration with a single evaluation sweep between improvements.
- **11.** 4 sweeps: its value is correct from v_4 on.
- **12.** Any of: no model needed; cost per backup independent of the number of states; avoids the curse of dimensionality.

CREDITS

Sources and credits

- The order of topics follows *Lecture 3: Markov Decision Processes and Dynamic Programming* of the DeepMind × UCL Reinforcement Learning Lecture Series (2021), by Diana Borsa. This deck is not an official DeepMind or UCL material.
- The text, figures, examples and simulations were written, drawn and run for ACML. Numbers labelled “our run” come from our own code; examples taken from books or papers are credited where they appear.
- Sutton & Barto (2018), chapters 3 and 4 (the recycling robot, the 4×4 gridworld, Jack’s car rental); our own re-runs.
- Bellman, R. (1957). *Dynamic Programming*; Bellman, R. (1984). *Eye of the Hurricane* (autobiography).
- Companion videos: **ACML Explains**, season 1, episodes 10–14. More from the lab: acml.teddylazebnik.com



Applied Computational Mathematics Lab

END OF LECTURE 3

Next: why dynamic programming works

Lecture 4 proves that these algorithms converge, using contractions and fixed points, and asks what happens when values are only approximate.

