SHANNON ROBOTICS

Reinforcement Learning for Robotics

The FCP Way

Complete mathematical foundations, interactive intuition, and working implementations in Rust — for robots that must act in a world that does not hold still.

Foundation · Conceptual · Practical

Reinforcement Learning for Robotics — The FCP Way

Print edition, generated from the interactive web edition.

This book is built on four works, cited throughout as baseline references: Sutton & Barto's *Reinforcement Learning: An Introduction* (2nd ed.) for the mathematical spine; Kober, Bagnell & Peters' *Reinforcement Learning in Robotics: A Survey* (IJRR 2013) for the robotics bridge; Tang et al.'s *Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes* (2024) for the modern taxonomy and the L0–L5 maturity rubric; and Akinola's Columbia lecture notes for pedagogical framing.

Every chapter of the web edition carries interactive simulations that run the real algorithms. In this print edition each appears as a described figure with its widget identifier, so the two editions can be read side by side.

Set in Space Grotesk and a Palatino-class serif. Composed with LaTeX.

Contents

Part I

Foundations of Sequential Decision-Making

CHAPTER 1

Why Reinforcement Learning for Robotics?

> *Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · TU Delft · Carnegie Mellon · TU Darmstadt
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Before any theorem, the case for the whole enterprise. Robots that must perceive, decide, and act in unstructured worlds outgrow hand-written controllers — not because engineers are careless, but because the number of situations to specify grows faster than anyone can enumerate them. Reinforcement learning is the discipline of replacing hand-written decision rules with optimized ones. This chapter gives you the vocabulary, the cast of robots, and — most importantly — the felt experience of watching a perfectly good controller fail in your own browser.

FOUNDATION

The perception–decision–actuation loop written as a discrete-time dynamical system, with every informal word flagged for the chapter that makes it precise.

CONCEPTUAL

A hand-coded robot controller you can break yourself, and the field's real-world results arranged on an honesty ladder.

PRACTICAL

The unicycle kinematics and controller trait that Rusty runs on, plus the first reproducible experiment: a noise sweep with pinned seeds.

AFTER THIS CHAPTER YOU CAN

- Trace the see–think–act loop on a concrete robot and name which stage each robotics subfield automates
- Explain why direct programming stops scaling, using the four axes of structure, perception, contact, and deformation
- Position learning from demonstration, reinforcement learning, and hybrid pipelines relative to each other, and state each one's preconditions
- Describe the four informal ingredients of an RL problem — agent, environment, reward, policy — on three different robots
- Place any published robot-RL system in Tang's taxonomy: competency, formulation, solution approach, and level of real-world success
- Read the rest of this book knowing exactly which promises have been made and which are still IOUs

1.1 See, think, act

Every robot that does anything useful runs the same loop. It **senses** the world, it **decides** what to do, and it **acts** — changing the world, which it then senses again. Cameras, lidar, and joint encoders feed the first stage; planners, controllers, and policies occupy the second; motors and grippers execute the third.

Write it down with a little more care. At discrete time step t, the robot receives an observation o_t, chooses an action

$$a_t = \pi(o_t),$$

and the world advances to a new configuration influenced by a_t together with whatever disturbances the universe supplies. The function π is called a **policy**, and essentially all of robotics is the search for good policies.

Three words in that paragraph are doing unearned work, and it is worth naming the debt now.

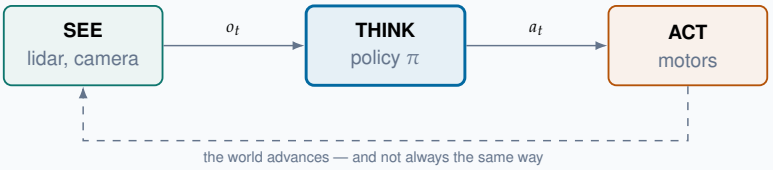
THREE IOUS

"Receives" and **"advances"** describe random processes — the same action in the same situation does not produce the same result twice. Chapter 2 builds the probability machinery; Chapter 4 turns "advances" into a conditional distribution $p(s' \mid s, a)$.

"Observation" is deliberately not "state". A robot never sees the true configuration of the world — only noisy, partial projections of it. We plant that distinction here and harvest it in Chapter 4, where the honest formalism turns out to be a *partially observable* Markov decision process.

The loop looks innocent because we have all watched robots work. The interesting question is what happens to it when the world stops cooperating.

FIGURE 1.1



The see–think–act loop. Perception maps the world to an observation, the policy maps the observation to an action, and actuation changes the world — which changes the next observation. Everything this book does happens inside the middle box.

1.2 Why direct programming breaks

Meet **Rusty**, a differential-drive mobile robot and the first member of this book's cast. Rusty's job is to reach a goal without hitting anything. His controller is the textbook potential-field method: steer toward the goal, steer away from obstacles, sum the two.

It is a good controller. It is also about to fail in front of you.

INTERACTIVE · `ch01-drive-rusty`

Drive Rusty — the case for learning

A hand-written potential-field controller steers Rusty toward a goal and away from obstacles. Choose the room (empty, cluttered, concave trap) and the driver (hand-coded or your own arrow keys). In the empty room the controller is optimal; in the concave trap it parks in a local minimum and never escapes, while a human drives out immediately.

Run this simulation in the web edition: `/chapters/why-rl-for-robotics #ch01-drive-rusty`

Nothing went wrong with the code. The potential-field method is correct, well understood, and provably convergent under assumptions — assumptions the concave trap violates. And here is the uncomfortable part: you fixed it instantly. You looked at the screen, saw that backing up was progress, and drove out. Encoding *that* judgment — for every trap, in every room, under every lighting condition — is the job direct programming asks of us.

A controller is a function $\pi : \mathcal{O} \to \mathcal{A}$ from observations to actions. Programming means specifying π by hand, case by case. The effort required grows not with the size of the code but with **the diversity of inputs the robot actually**

encounters. Four axes generate that diversity, and they explain why some robots shipped decades ago while others remain research projects:

Axis	Easy end	Hard end
Structure & consistency	Vacuum robot: flat floor, same room nightly	Home-cleaning robot: novel clutter every time
Perception	Pool cleaner: featureless tiled basin	Cooking robot: identify ingredients by sight and state
Manipulation contact	Manufacturing arm: fixtured parts, known poses	Assembly under uncertainty: contact decides the outcome
Deformation	Warehouse robot: rigid boxes	Laundry robot: cloth has effectively infinite configurations

Robots that shipped commercially live at the left of every row. Robots that remain research projects live at the right of at least one.

> **THE SCALE OF THE PROBLEM, IN NUMBERS**
>
> It is tempting to think the state space is small because the robot is small. Kober, Bagnell and Peters make the point concrete: **10–30 dimensional continuous state–action spaces are entirely normal in robotics**, and already count as large. A modest 7-degree-of-freedom arm has a 14-dimensional state — positions and velocities — before you add a single object to the scene. Chapter 14 turns that observation into arithmetic that ends the argument permanently.

Here is what Rusty's world actually looks like in code — the unicycle kinematics and the controller trait that the sandbox above runs on.

RUST `rl-envs/src/rusty/diff_drive.rs`

```rust
pub struct Pose { pub x: f64, pub y: f64, pub theta: f64 }
pub struct Twist { pub v: f64, pub omega: f64 }   // forward m/s, yaw rad/s

/// Unicycle kinematics, advanced by one control period `dt` (zero-order hold).
pub fn step(p: &Pose, u: &Twist, dt: f64) -> Pose {
    Pose {
        x: p.x + u.v * p.theta.cos() * dt,
        y: p.y + u.v * p.theta.sin() * dt,
        theta: p.theta + u.omega * dt,
    }
}

pub trait Controller {
    fn act(&mut self, scan: &LidarScan) -> Twist;
}

/// The hand-coded baseline this chapter exists to break.
pub struct WallFollower { pub target_gap: f64, pub k_p: f64 }

```

```rust
impl Controller for WallFollower {
    fn act(&mut self, scan: &LidarScan) -> Twist {
        let gap = scan.min_range_deg(-95.0..=-85.0); // right-side beams
        Twist { v: 0.4, omega: self.k_p * (gap - self.target_gap) }
    }
}
```

Rusty's body and the interface every controller implements. The hand-coded WallFollower is deliberately simple — it exists to be broken.

Notice what the `Controller` trait does *not* say: nothing about how `act` arrives at its answer. A hand-written proportional rule satisfies it. So does a neural network trained for three days on a GPU cluster. That interface is the seam along which this entire book is built.

1.3 Three roads to a policy

If hand-writing π does not scale, what does? There are exactly three families of answers, and they differ in what they demand from you rather than in how clever they are.

Demonstrate. Collect a set $D = \{(s, a)\}$ of situations and the actions an expert took in them, then fit a function to it. This is *learning from demonstration*, and it is supervised learning wearing a robotics hat. Its preconditions are strong and worth stating aloud: **a teacher must exist**, and **demonstration must be physically possible**. You can demonstrate driving. Demonstrating a backflip on a quadruped, or the precise force profile of an insertion, is another matter.

Reinforce. Provide no teacher at all — only a scalar judgment of how well things are going. The robot practices, receives reward, and improves. This is *reinforcement learning*. The precondition is subtler: you must be able to write down what success means as a number. Reward specifies **what** you want, never **how** to achieve it, and that gap is simultaneously the method's power and its most reliable source of disaster (Chapter 14 catalogues the disasters).

Hybridize. Use demonstrations to get a competent starting point, then let reinforcement learning refine it beyond what the demonstrator could do. Nearly every real-world success in manipulation takes this road, and Chapters 16 and 17 build it properly.

	Teacher required	Sample cost	Reward-design burden
Demonstration	Yes — and must be able to perform the task	Low (one-time collection)	None
Reinforcement	No	High — the robot must practice	High — and unforgiving
Hybrid	Yes, for initialization only	Medium	Medium

WHEN DOES RL APPLY AT ALL?

Kober and colleagues give the cleanest test in the literature: reinforcement learning applies when a task **can be phrased as an optimization problem** and **exhibits temporal structure** — when what you do now changes what you will face later.
Both halves matter. Choosing the best grasp for a stationary object is an optimization problem with no temporal structure; that is a *bandit* problem, and Chapter 3 solves it with far simpler machinery. The moment grasping the object moves it, and the next decision depends on where it moved, you need the full apparatus.

1.4 The anatomy of an RL problem

Four ingredients, stated informally here and made precise in Chapter 4.

The **agent** is the thing that chooses. The **environment** is everything else — including, importantly, the robot's own motors, since the agent controls them only through commands that may not be obeyed exactly. The **reward** $r_{t+1} \in \mathbb{R}$ is a scalar arriving after action a_t; the indexing is deliberate and follows Sutton and Barto, because the reward is a consequence of the action, not a property of the situation that preceded it. The **policy** π maps what the agent perceives to what it does.

From these, one derived quantity dominates everything: the **return**, the accumulated reward over an episode. An agent that maximizes immediate reward is myopic; an agent that maximizes return is strategic. Making that distinction rigorous — and finite, since sums of infinitely many rewards need care — occupies Chapter 4.

Sutton and Barto identify four elements of any RL system. Each lands somewhere concrete on Rusty:

- **Policy** — Rusty's rule for choosing a heading given what the lidar reports.
- **Reward signal** — +25 for reaching the dock, −1 per step elapsed, −10 for hitting a shelf.
- **Value function** — how promising a corridor is *in the long run*, accounting for where it leads. Not the same as its immediate reward, and vastly more useful.
- **Model** — a floor map, if the robot has one. Chapters 5 and 12 show what becomes possible when it does; Chapters 6 and 7 show what to do when it does not.

THE TENSION THAT GENERATES THE ENTIRE FIELD

Rusty knows one corridor that reaches the dock in forty steps. There may

be a twenty-step route through an aisle he has never entered. Taking the known route earns a reliable return. Trying the unknown one might earn more, or might waste an episode.
He cannot do both. This is the **exploration–exploitation dilemma**, and it is not an engineering annoyance to be tuned away — it is the structural feature that separates reinforcement learning from every other kind of machine learning. Chapter 3 isolates it in the simplest setting that still contains it, and derives what optimal exploration actually looks like.

1.5 What actually worked

Enthusiasm is cheap. The useful question is what reinforcement learning has genuinely achieved on physical hardware, and the honest answer is: a great deal in a few competencies, and remarkably little in others.

Tang and colleagues surveyed the field with an unusually disciplined instrument — a six-level maturity ladder running from "validated only in simulation" to "deployed in a commercial product". It is worth internalizing, because it converts vague claims into checkable ones, and because this book applies it to its own capstone in Chapter 22.

INTERACTIVE · `ch01-success-levels`

Levels of real-world success

Tang et al.'s L0–L5 maturity ladder with surveyed systems placed on it, filterable by competency. Locomotion reaches L5; human–robot interaction sits at L1.

Run this simulation in the web edition: `/chapters/why-rl-for-robotics #ch01-success-levels`

Read the pattern rather than the individual entries. **Locomotion reached commercial deployment** because its dynamics simulate faithfully, its rewards shape naturally, and a stumble costs a fall rather than a lawsuit. **Manipulation stalls below the real-world tiers** because object diversity defeats simulation and contact is where physics engines are least trustworthy. **Human–robot interaction sits near the bottom** for a reason no algorithm will fix: humans are the part of the environment nobody can simulate, so neither zero-shot transfer nor cheap practice is available.

That spread — not any particular algorithm — is the map this book navigates. Part IV devotes a chapter to each region of it.

1.6 The method, the cast, and the contract

Three commitments run through every remaining chapter.

Foundation. Derivations are complete. When this book states a theorem, it either proves it or states it precisely and says exactly where the proof lives. "It can be shown" is not an argument, and you should not accept it from a textbook any more than from a colleague.

Conceptual. Every hard concept gets something you can manipulate. This is not decoration. A discount factor you have dragged from 0.5 to 0.99, watching the value function's reach stretch across a warehouse, is a discount factor you understand in a way that reading $1/(1-\gamma)$ does not deliver.

Practical. Every algorithm is implemented in Rust, with seeded randomness and real tests. You will finish with working code, not pseudocode with the hard parts elided.

The cast is small and fixed. **Rusty** you have met. **Pendle**, a pendulum, arrives in Chapter 2 — simple enough that we can do every calculation exactly, which makes him the perfect vehicle for the mathematics. **Reacher**, a two-link arm, arrives in Chapter 3 and carries the manipulation thread. **Ferris**, a quadruped, arrives in Chapter 15 and gets a full chapter of his own plus the capstone.

> **RUSTY'S STATE AT THE END OF THIS CHAPTER**
>
> Rusty has a body (unicycle kinematics), a sensor (2-D lidar with a noise model), an environment (a warehouse pen), a hand-coded controller with documented failure modes, and a reward configuration. He has everything he needs except the ability to learn. Chapter 4 formalizes his world as a Markov decision process; Chapters 5 through 7 teach him.

1.7 Chapter bridge

We have made claims and issued IOUs. Let us be precise about which is which.

Claimed and defended: hand-written controllers fail through input diversity rather than programmer error; there are three roads to a policy with different preconditions; robot RL's real-world successes cluster in competencies that simulate well.

Still owed: the loop is not yet mathematics. "Reward" is not yet a random variable. "The agent eventually finds the best policy" is not yet a limit theorem — and until it is, it is marketing.

Chapter 2 pays these debts. It supplies probability spaces and conditional expectation, contraction mappings and the fixed-point theorem that every convergence proof in this book eventually invokes, the discretization that turns a robot's continuous dynamics into discrete decisions, and the Robbins–Monro theorem — the single result that makes learning from noisy samples work at all. Pendle arrives to carry all of it, because he is simple enough that we can compute everything exactly and check the theory against arithmetic.

Exercises

1 FOUNDATION • **Name your own hard tasks**

For each of the four diversity axes (structure, perception, contact, deformation), give one easy/hard task pair not used in this chapter, and justify each placement in two sentences.

2 FOUNDATION •• **Optimization without time**

Kober's test says RL applies when a task is an optimization problem AND exhibits temporal structure. Give a robot task that is genuinely an optimization problem but has no temporal structure. Name the problem class it belongs to — you have just previewed Chapter 3.

3 FOUNDATION • **The thermostat**

Write the see–think–act loop for a household thermostat: identify o_t, a_t, and r_{t+1} explicitly. Is exploration present in a standard thermostat? Should it be? Argue both sides in a paragraph.

4 CONCEPTUAL •• **Find the failure boundary**

In the Rusty sandbox, work out which room geometry first defeats the hand-coded controller. Then describe, in one sentence, the property of that geometry that the potential-field method cannot represent.

5 CONCEPTUAL •• **Diagnose a stalled competency**

In the success-levels explorer, find a competency with no entry above L2. Propose one reason grounded in the survey's own analysis — is the bottleneck simulation fidelity, reward design, sample cost, or safety?

6 PRACTICAL •• **A second hand-coded controller**

Implement a bang-bang gap keeper alongside the proportional WallFollower and add it to the parameter sweep. Does it dominate the P-controller anywhere in the noise–slip plane, or is it uniformly worse?

7 PRACTICAL ••• **Tag a recent paper**

Take a robot-RL paper from the last two years and tag it with all four taxonomy axes: competency, problem formulation, solution approach, and level of real-world success. Defend the level assignment in a paragraph, citing only the experimental evidence the

paper actually reports — not what it claims in the abstract.

8 CONCEPTUAL •• **Judge your own driving**

Drive Rusty manually for sixty seconds, then score that same trajectory under three reward configurations: dock-only sparse, step-penalty dense, and collision-heavy

CODING TASK `rl-envs, rl-core`

Reproduce the failure cliff

Build the noise sweep described above using `rayon` for parallel rollouts and a seeded `StdRng` per run. The point is not the plot — it is that you now own a reproducible experimental harness, and every later chapter assumes you can run one.

Success is not "the controller fails". Success is knowing *at which noise level* it fails, and being able to show someone else the same number tomorrow.

DELIVERABLES

- A seeded sweep over lidar noise $\sigma \in [0, 0.5]$ m across 100 layout seeds
- A success-rate plot showing the cliff, exported as the page's static figure
- Byte-identical results on re-run from the pinned seed — the book's reproducibility standard

1.8 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapter 1, especially §1.1–1.3 (elements of RL) and §1.5 (tic-tac-toe as a first complete example).

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§1.1 RL in the context of machine learning; §1.2 versus optimal control; §1.3 in the context of robotics — dimensionality, partial observability, and the cost of real experience.

BASELINE Tang, C., Abbatematteo, B., Hu, J., Chandra, R., Martín-Martín, R. & Stone, P. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.1 competencies, §3.2 problem formulation, §3.3 solution approach, §3.4 the L0–L5 levels of real-world success used throughout this book.

BASELINE Akinola, I. (n.d.). Reinforcement Learning for Robotics *Columbia University lecture notes*.
The see–think–act framing, the direct-programming difficulty axes, and the LfD / RL / hybrid taxonomy.

FURTHER READING AND MODERN SOURCES

Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V. & Scaramuzza, D. (2023). Champion-level drone racing using deep reinforcement learning *Nature 620*.
The clearest demonstration that RL can beat expert humans on a physical, dynamic, safety-critical task.

Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain *Science Robotics 5(47)*.
The teacher–student recipe rebuilt in Chapters 15 and 18.

Miki, T., Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2022). Learning robust perceptive locomotion for quadrupedal robots in the wild *Science Robotics 7(62)*.
Perceptive locomotion at L4 — diverse, representative real-world conditions.

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V. & Hutter, M. (2019). Learning agile and dynamic motor skills for legged robots *Science Robotics 4(26)*.
Where the modern sim-to-real locomotion pipeline begins.

Akkaya, I. et al. (OpenAI) (2019). Solving Rubik's Cube with a Robot Hand *arXiv:1910.07113*.
The in-hand manipulation line traced in Chapter 20 — and a case study in how much randomization dexterity costs.

Sünderhauf, N. et al. (2018). The limits and potentials of deep learning for robotics *International Journal of Robotics Research 37(4–5)*.
The pre-success-era assessment that the 2024 survey positions itself against — useful for calibrating how fast this field moves.

CHAPTER 2

The Mathematical Toolkit

> *The relationship between disciplines has sufficient promise to be likened to that between physics and mathematics.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On the relationship between robotics and reinforcement learning
>
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Chapter 1 left three debts: the loop is not yet mathematics, reward is not yet a random variable, and 'eventually finds the best policy' is not yet a theorem. This chapter pays all three. We build the probability that makes expectations meaningful, the contraction machinery that every convergence proof in this book eventually invokes, the discretization that turns a robot's continuous physics into discrete decisions, and the Robbins–Monro theorem — the single result that explains why learning from noisy samples works at all. Pendle arrives to carry it, because he is simple enough that every calculation can be checked by hand.

FOUNDATION

Probability spaces, conditional expectation and the tower property, contraction mappings with Banach proved in full, ODE discretization with error orders, and Robbins–Monro convergence.

CONCEPTUAL

A contraction you can iterate at any γ, and Pendle's dynamics integrated three ways so you can watch explicit Euler manufacture energy from nothing.

PRACTICAL

Pendle's dynamics and integrators in Rust with nalgebra, plus a Robbins–Monro estimator you can drive with noisy samples.

◎ AFTER THIS CHAPTER YOU CAN

- State what a random variable and a conditional expectation actually are, and use the tower property fluently
- Prove the Banach fixed-point theorem and explain why it guarantees a unique optimal value function
- Discretize a continuous robot dynamics with Euler and RK4, and predict which one will explode and when
- State the Robbins–Monro conditions and recognize them in every learning rate you will meet later
- Read the incremental update rule New $\leftarrow$ Old + α(Target − Old) as an instance of stochastic approximation

2.1 Probability, quickly but honestly

Chapter 1 said the world "advances" after an action. Making that precise requires almost no machinery, but it does require the right machinery.

A **probability space** is a triple $(\Omega, \mathcal{F}, \mathbb{P})$: a set of outcomes Ω, a collection $\mathcal{F}$ of events we can assign probabilities to, and a measure $\mathbb{P}$ assigning each event a number in $[0, 1]$ with $\mathbb{P}(\Omega) = 1$. A **random variable** X is a function from Ω to $\mathbb{R}$ — not a number, a *function*. When we write $X = 3$ we mean the event $\{\omega : X(\omega) = 3\}$.

This matters for us because Rusty's next position is exactly such a function: it depends on the wheel slip that happened to occur, which is the outcome ω. Two identical commands from two identical positions produce two different results, and the difference is not error — it is the environment.

The **expectation** of X is its probability-weighted average, $\mathbb{E}[X] = \sum_\omega \mathbb{P}(\omega) X(\omega)$ in the discrete case. Expectation is linear, which is the property we will lean on constantly:

$$\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y]$$

— and note this holds whether or not X and Y are independent. A surprising amount of reinforcement learning is linearity of expectation applied patiently.

Definition 2.1 — Conditional expectation. For random variables X and Y, the conditional expectation $\mathbb{E}[X \mid Y]$ is a random variable — a function of Y — whose value at $Y = y$ is the average of X over the outcomes where $Y = y$:

$$\mathbb{E}[X \mid Y = y] = \sum_{\omega} \mathbb{P}(\omega \mid Y = y)\, X(\omega).$$

The single most useful fact about conditional expectation is that averaging it recovers the unconditional average.

Theorem 2.2 — Tower property (law of total expectation).

$$\mathbb{E}\big[\mathbb{E}[X \mid Y]\big] = \mathbb{E}[X].$$

PROOF

Write the outer expectation as a sum over values of Y, then expand the inner one:

$$\mathbb{E}\big[\mathbb{E}[X \mid Y]\big] = \sum_{y} \mathbb{P}(Y = y)\, \mathbb{E}[X \mid Y = y] = \sum_{y} \mathbb{P}(Y = y) \sum_{\omega} \mathbb{P}(\omega \mid Y = y) X(\omega).$$

By the definition of conditional probability, $\mathbb{P}(Y = y)\,\mathbb{P}(\omega \mid Y = y) = \mathbb{P}(\omega,\, Y = y)$. Substituting and exchanging the order of summation:

$$= \sum_{\omega} X(\omega) \sum_{y} \mathbb{P}(\omega,\, Y = y) = \sum_{\omega} X(\omega)\, \mathbb{P}(\omega) = \mathbb{E}[X].$$

■

Every Bellman equation in this book is the tower property applied to a return, conditioned on the first action. That is genuinely all it is. When Chapter 4 derives

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

the derivation will consist of splitting the return into "first reward" plus "the rest", conditioning on where you land, and invoking Theorem 2.2.

2.2 Markov chains and the memory assumption

A sequence of random variables $S_0, S_1, S_2, \ldots$ has the **Markov property** if the future is conditionally independent of the past given the present:

$$\mathbb{P}(S_{t+1} = s' \mid S_t, S_{t-1}, \ldots, S_0) = \mathbb{P}(S_{t+1} = s' \mid S_t).$$

The state is then a **sufficient statistic** for prediction: knowing it, the history adds nothing.

⚠ ROBOTS VIOLATE THIS CONSTANTLY

The Markov property is an assumption about your *state representation*, not about the world. Rusty's position alone is not Markov if his wheels slip in a

way that depends on accumulated dust. A camera image is not Markov if velocity matters and a single frame cannot reveal it.
The standard fixes are to enlarge the state (stack frames, add velocities) or to admit partial observability and carry a belief. Chapter 4 formalizes the second option; Chapter 19 builds recurrent policies that learn to track a belief without being told to.

2.3 Contraction mappings and the theorem behind everything

Here is the mathematical heart of the chapter — the result that will justify value iteration, Q-learning, and half of Part I.

Definition 2.3 — Contraction mapping. Let $(\mathcal{X}, d)$ be a metric space. An operator $T : \mathcal{X} \to \mathcal{X}$ is a γ**-contraction** for $\gamma \in [0, 1)$ if for all $x, y \in \mathcal{X}$,

$$d(Tx, Ty) \le \gamma\, d(x, y).$$

Applying T moves any two points strictly closer together. Intuitively, repeated application must crush everything toward a single point. The theorem says exactly that, and adds a computable error bound.

Theorem 2.4 — Banach fixed-point theorem. Let $(\mathcal{X}, d)$ be a **complete** metric space and T a γ-contraction. Then:

1. T has exactly one fixed point x^* with $Tx^* = x^*$;
2. for any starting point x_0, the sequence $x_{k+1} = Tx_k$ converges to x^*;
3. the convergence is geometric, with the a-priori bound

$$d(x_k, x^*) \le \frac{\gamma^k}{1-\gamma}\, d(x_1, x_0).$$

PROOF

The sequence is Cauchy. From the contraction property, consecutive iterates satisfy $d(x_{k+1}, x_k) \le \gamma\, d(x_k, x_{k-1})$, and by induction

$$d(x_{k+1}, x_k) \le \gamma^k\, d(x_1, x_0).$$

For any $m > k$, the triangle inequality and the geometric series give

$$d(x_m, x_k) \le \sum_{i=k}^{m-1} d(x_{i+1}, x_i) \le \sum_{i=k}^{m-1} \gamma^i \, d(x_1, x_0) \le \frac{\gamma^k}{1-\gamma} \, d(x_1, x_0).$$

Since $\gamma < 1$, the right-hand side goes to zero as $k \to \infty$, so the sequence is Cauchy. Completeness gives a limit x^*.

The limit is a fixed point. A contraction is continuous (take $\delta = \varepsilon$ in the definition), so

$$Tx^* = T\big(\lim_k x_k\big) = \lim_k Tx_k = \lim_k x_{k+1} = x^*.$$

Uniqueness. Suppose $Tx^* = x^*$ and $Ty^* = y^*$. Then $d(x^*, y^*) = d(Tx^*, Ty^*) \le \gamma \, d(x^*, y^*)$, so $(1-\gamma)\, d(x^*, y^*) \le 0$. Since $\gamma < 1$, this forces $d(x^*, y^*) = 0$, hence $x^* = y^*$.

The bound. Let $m \to \infty$ in the Cauchy estimate above. $$ ■

Three payoffs, all collected later: **existence** (an optimal value function exists), **uniqueness** (there is only one, so "the" optimal value function is well defined), and **an algorithm with a stopping rule** (iterate, and the bound tells you when to stop).

INTERACTIVE · `ch02-contraction-map`

A contraction, iterated

Iterates of $T(x) = \gamma x + 2$ converge to the same fixed point from any start. Drag γ toward 1 and convergence slows; the a-priori bound is plotted beside the actual error and never dips below it.

Run this simulation in the web edition: `/chapters/mathematical-toolkit #ch02-contraction-map`

A PUN THAT IS ACTUALLY A THEOREM

The contraction modulus is written γ, and so is the discount factor. That is not a coincidence or a notational collision. Chapter 4 proves that the Bellman operator is a γ-contraction **where γ is precisely the discount factor**. Discounting the future is what makes the mathematics converge; a robot that values the infinite future equally is a robot whose value function may not exist.

2.4 From continuous physics to discrete decisions

Robots obey differential equations. Reinforcement learning acts at discrete time steps. Something must bridge them, and the bridge is not free.

Meet **Pendle**, a torque-actuated pendulum, measured with $\theta = 0$ at the upright position:

$$m\ell^2\ddot{\theta} = mg\ell \sin\theta - b\,\dot{\theta} + \tau.$$

Writing the state as $x = [\theta, \dot{\theta}]^\top$ turns this into a first-order system $\dot{x} = f(x, \tau)$. To simulate it we need to advance x by a finite step Δt. The simplest choice is **explicit Euler**:

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, \tau_k),$$

which is exact if f is constant over the interval and wrong otherwise, with local error $O(\Delta t^2)$ and global error $O(\Delta t)$. **Fourth-order Runge–Kutta** samples the derivative four times per step:

$$x_{k+1} = x_k + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

with global error $O(\Delta t^4)$ — four extra derivative evaluations buying three extra orders of accuracy.

The difference is not academic. With $\tau = 0$ the true pendulum conserves total mechanical energy exactly. Explicit Euler does not merely approximate that conservation — it systematically *violates* it, injecting energy at every step until the pendulum spins up out of nothing.

INTERACTIVE · `ch02-integrator-playground`

Pendle: continuous dynamics, discrete steps

The pendulum integrated by explicit Euler, semi-implicit Euler and RK4, with total energy plotted. At $\tau = 0$ energy should be flat: explicit Euler's climbs without bound, RK4's holds.

Run this simulation in the web edition: `/chapters/mathematical-toolkit#ch02-integrator-playground`

THIS IS A SIM-TO-REAL FAILURE IN MINIATURE

A policy trained against an integrator that manufactures energy has learned to exploit a physics that does not exist. Chapter 15 shows this exact failure at full scale, where contact solvers — far harder than a pendulum — are where simulation and reality part company. Everything you just watched happen to Pendle happens to quadrupeds, with more expensive consequences.

RUST `rl-envs/src/pendle.rs`

```rust
use nalgebra::Vector2;

pub struct PendleParams {
    pub mass: f64,
    pub length: f64,
    pub gravity: f64,
    pub damping: f64,
}

/// x_dot = f(x, tau) for the torque-actuated pendulum, theta = 0 at upright.
pub fn dynamics(x: &Vector2<f64>, tau: f64, p: &PendleParams) -> Vector2<f64> {
    let (theta, omega) = (x[0], x[1]);
    let inertia = p.mass * p.length * p.length;
    let alpha = (p.mass * p.gravity * p.length * theta.sin()
                 - p.damping * omega
                 + tau) / inertia;
    Vector2::new(omega, alpha)
}

pub fn euler_step(x: &Vector2<f64>, tau: f64, dt: f64, p: &PendleParams) -> Vector2<f64>
   {
    x + dt * dynamics(x, tau, p)
}

pub fn rk4_step(x: &Vector2<f64>, tau: f64, dt: f64, p: &PendleParams) -> Vector2<f64> {
    let k1 = dynamics(x, tau, p);
    let k2 = dynamics(&(x + 0.5 * dt * k1), tau, p);
    let k3 = dynamics(&(x + 0.5 * dt * k2), tau, p);
    let k4 = dynamics(&(x + dt * k3), tau, p);
    x + (dt / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
}
```

Pendle's dynamics and two integrators. The state is $[\theta, \dot\theta]$; note that rk4_step is a drop-in replacement for euler_step, which is exactly how the widget above swaps them.

2.5 Learning from noisy samples: Robbins–Monro

Now the result that makes learning possible at all.

Suppose you want to find the root of a function $h(\theta) = 0$, but you can never evaluate h — only a noisy sample $H(\theta) = h(\theta) + \text{noise}$. This is precisely our situation: we want the *expected* return, and we only ever observe one noisy trajectory at a time.

Robbins and Monro's answer, from 1951, is to take small steps in the direction each noisy sample suggests, with a step size that shrinks:

$$\theta_{k+1} = \theta_k + \alpha_k \, H(\theta_k).$$

Theorem 2.5 — Robbins–Monro conditions. Under regularity conditions on h and bounded noise variance, the iterates θ_k converge almost surely to the root θ^* provided the step sizes satisfy

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \text{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

The two conditions have clean interpretations, and it is worth being able to recite them:

- $\sum \alpha_k = \infty$ — **the steps must be able to travel infinitely far.** If they sum to something finite, the iterate can never reach a root that lies beyond that distance, no matter how many samples arrive.
- $\sum \alpha_k^2 < \infty$ — **the steps must shrink fast enough to average out the noise.** Otherwise the iterate rattles around the root forever without settling.

The canonical choice $\alpha_k = 1/k$ satisfies both: the harmonic series diverges, and the sum of $1/k^2$ converges to $\pi^2/6$. A constant α satisfies the first but not the second — which is why constant learning rates track a moving target instead of converging, exactly the trade you want when the environment is non-stationary.

YOU WILL SEE THIS RULE EVERYWHERE

Every learning rule in this book has the shape

$$\text{New estimate} \leftarrow \text{Old estimate} + \alpha\,(\text{Target} - \text{Old estimate}).$$

Sample-average action values in Chapter 3, TD learning in Chapter 6, Q-learning updates, even the gradient steps of Chapter 10 — all are Robbins–Monro with different targets. When you meet a new algorithm, the first useful question is: *what is its target, and is that target biased?*

2.6 Linear algebra and gradients, in the amount we need

Two more tools, stated without ceremony because they will be used constantly.

Matrices as operators. Given a policy on a finite state space, the transition probabilities form a matrix P_π, and expected rewards a vector r_π. Chapter 4 will show the value function satisfies $v_\pi = r_\pi + \gamma P_\pi v_\pi$, whose solution is

$$v_\pi = (I - \gamma P_\pi)^{-1} r_\pi.$$

The inverse exists because γP_π has spectral radius at most $\gamma < 1$, so the Neumann series $\sum_k (\gamma P_\pi)^k$ converges — the same geometric argument as the Banach bound, wearing matrix clothes.

Gradients. For $J : \mathbb{R}^n \to \mathbb{R}$, the gradient $\nabla J(\theta)$ points in the direction of steepest increase, and gradient ascent takes $\theta \leftarrow \theta + \alpha \nabla J(\theta)$. Chapter 10 spends its entire length computing one specific gradient — that of expected return with

respect to policy parameters — which is hard precisely because the distribution you are averaging over depends on the parameters you are differentiating.

2.7 Chapter bridge

The debts of Chapter 1 are paid. "The world advances" is a conditional distribution. "Reward" is a random variable with an expectation. "Eventually finds the best policy" will be Banach's theorem applied to an operator we have not yet built.

Chapter 3 puts the toolkit to work on the smallest problem that still contains the essential difficulty: a single decision, repeated, with no state at all. Stripping away state exposes the exploration–exploitation dilemma in its pure form, and lets us derive real regret bounds with the concentration inequalities we now have the vocabulary for. Reacher joins the cast there, choosing among grasp primitives where every attempt costs hardware wear.

Exercises

1 FOUNDATION • **Tower property by hand**

Let X be the total reward from a two-step episode and Y the first action. Compute E[X] two ways — directly, and via E[E[X | Y]] — for a small example you construct. They must agree; make sure you see why.

2 FOUNDATION •• **Is the operator a contraction?**

Show that T(x) = γx + c is a γ-contraction on with the usual metric, and find its fixed point in closed form. Then show that T(x) = x + c is not a contraction for any c ≠ 0, and explain what goes wrong with Banach's conclusion.

3 FOUNDATION •• **Robbins–Monro schedules**

Which of these satisfy both Robbins–Monro conditions? (a) α_k = 1/k, (b) α_k = $1/k^2$, (c) α_k = $1/\sqrt{k}$, (d) α_k = 0.1. For each failure, say which condition breaks and what behaviour that produces in practice.

4 CONCEPTUAL • **The $\gamma \to 1$ slowdown**

In the contraction widget, record the number of iterations needed to reach an error below 0.01 for γ = 0.5, 0.9, 0.95, 0.99. Plot iterations against $1/(1-\gamma)$. What relationship do you find, and why does the a-priori bound predict it?

5 CONCEPTUAL •• **Find the stability boundary**

In the Pendle playground with $\tau = 0$, find the largest Δt at which explicit Euler keeps energy within 10% over ten seconds. Repeat for RK4. Express the ratio — that is the compute budget RK4 buys you per step.

6 PRACTICAL •• **Implement Robbins–Monro**

Write a seeded estimator that finds the mean of a noisy signal using $\alpha\_k = 1/k$, and a second using constant $\alpha = 0.1$. Feed both a signal whose mean jumps halfway through the run. Plot both trajectories and explain which you would deploy on a robot whose payload changes.

7 PRACTICAL ••• **Verify the error orders**

Empirically confirm that Euler's global error is $O(\Delta t)$ and RK4's is $O(\Delta t)$: simulate Pendle for a fixed horizon at halving step sizes, measure error against a very-fine-step reference, and fit the slope on a log–log plot. The slopes should come out near 1 and 4.

2.8 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

§2.4 incremental implementation — the Robbins–Monro special case that opens Chapter 3; §3.1 for the Markov property.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§1.3 for the continuous state–action setting that motivates the discretization material here.

FURTHER READING AND MODERN SOURCES

Robbins, H. & Monro, S. (1951). A Stochastic Approximation Method *Annals of Mathematical Statistics 22(3)*.

The original paper. Short, readable, and the ancestor of every learning rate in this book.

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales *Fundamenta Mathematicae 3*.

The fixed-point theorem, in the original.

Bertsekas, D. P. & Tsitsiklis, J. N. (1996). Neuro-Dynamic Programming *Athena Scientific*.
The rigorous treatment of stochastic approximation applied to dynamic programming. Where to go when this book says "the proof lives elsewhere".

Hairer, E., Nørsett, S. P. & Wanner, G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems *Springer*.
The reference for integrator error orders and stability regions — including why symplectic integrators behave so much better on mechanical systems.

CHAPTER 3

Multi-Armed Bandits: Exploration & Exploitation

> *The agent must exploit what it already knows in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task.*
>
> **Richard S. Sutton & Andrew G. Barto** · University of Alberta · University of Massachusetts Amherst
> *Reinforcement Learning — An Introduction, 2nd edition*

Strip a reinforcement learning problem of everything except the difficulty that makes it reinforcement learning, and you get a bandit. There is no state, no sequence, no credit assignment across time — only a repeated choice among actions whose values you must estimate from the rewards they produce. What survives this stripping is the exploration–exploitation dilemma, and here it is simple enough to analyze exactly. We derive real regret bounds, meet the softmax rule that becomes the policy gradient in Chapter 10, and introduce Reacher, for whom every attempt costs hardware.

FOUNDATION

Action-value estimation, the incremental update as Robbins–Monro, regret decomposition, Hoeffding's inequality and the UCB bound, the gradient-bandit derivation, and Thompson sampling.

CONCEPTUAL

A live 10-armed testbed racing every algorithm on the three questions that matter: how much reward, how often right, how much lost to exploration.

PRACTICAL

The Bandit and BanditPolicy traits in rl-core — the book's first abstraction, and the shape every later agent follows.

AFTER THIS CHAPTER YOU CAN

- Define regret and explain why it, not reward, is the right measure of a bandit algorithm
- Derive the UCB action-selection rule from Hoeffding's inequality rather than accepting it as given
- Prove that constant-ε exploration incurs linear regret, and say what that means for a deployed robot
- Derive the gradient-bandit update and recognize it as the policy gradient theorem on a one-state problem
- Explain what a contextual bandit adds, and identify the precise moment a problem stops being a bandit at all

3.1 The simplest problem that is still hard

Reacher, a two-link arm, faces a mug. It has k grasp primitives available — top grasp, side grasp, rim hook, and so on. Each attempt either succeeds, partially succeeds, or knocks the mug over, and the outcome is noisy: the same primitive on the same mug does not always produce the same result.

Reacher must choose repeatedly, and wants to accumulate as much success as possible. Formally: at each step t it selects an action $A_t \in \{1, \dots, k\}$ and receives a reward R_t drawn from a fixed but unknown distribution associated with that action. Define the **true value** of an action as its expected reward:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a].$$

If Reacher knew q_*, the problem would be trivial: always pick the argmax. It does not know q_*, and the only way to learn about an action is to *take* it — spending a trial that could have gone to the currently-best-looking option.

WHY THIS IS NOT AN ACADEMIC TOY

For a software bandit, a wasted pull costs a fraction of a cent. For Reacher, every attempt costs gripper wear, a possible dropped mug, and thirty seconds of wall-clock time that a human must supervise. The cost asymmetry is why robotics cares more about sample efficiency than almost any other application area of RL, and it is the thread running from here to the

off-policy methods of Chapter 11 and the offline methods of Chapter 16.

3.2 Estimating values, incrementally

The obvious estimator is the sample average of the rewards received from action a so far:

$$Q_n(a) \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}.$$

Storing every reward would be wasteful, and it is unnecessary. Expand the average at step $n+1$ and rearrange:

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) = \frac{1}{n}\Big(R_n + (n-1)Q_n\Big)$$

$$= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) = Q_n + \frac{1}{n}\Big(R_n - Q_n\Big).$$

Which is exactly the master pattern promised in Chapter 2:

$$\text{New} \leftarrow \text{Old} + \alpha\,\big(\text{Target} - \text{Old}\big)\,.$$

With $\alpha_n = 1/n$ the Robbins–Monro conditions hold ($\sum 1/n = \infty$, $\sum 1/n^2 < \infty$), so $Q_n(a) \to q_*(a)$ almost surely. Sample averaging is stochastic approximation, and we already proved it converges.

Non-stationarity changes the calculus. Replace $1/n$ with a constant α and unroll:

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i.$$

The weights $\alpha(1-\alpha)^{n-i}$ decay geometrically into the past — an exponential recency-weighted average. The second Robbins–Monro condition now fails, so the estimate never fully settles. That is a feature when Reacher's gripper wears down and old data becomes actively misleading.

3.3 Regret: the right scorecard

Total reward is a bad way to compare algorithms, because it depends on how generous the bandit happens to be. The standard measure is **regret** — how much worse you did than an oracle that knew q_* from the start:

$$L_n \doteq n \max_a q_*(a) - \sum_{t=1}^{n} q_*(A_t) = \sum_a \Delta_a\, \mathbb{E}[N_n(a)],$$

where $\Delta_a = \max_{a'} q_*(a') - q_*(a)$ is the **gap** of action a and $N_n(a)$ counts how often it was chosen. That identity is worth sitting with: **regret is the sum over suboptimal actions of how bad they are, times how often you took them.** An algorithm is good precisely when it stops taking bad actions quickly.

Proposition 3.1 — Constant ε-greedy has linear regret. An ε-greedy policy with fixed $\varepsilon > 0$ explores uniformly at random on an ε fraction of steps regardless of what it has learned. Each such step incurs expected regret $\frac{1}{k}\sum_a \Delta_a$, so after n steps

$$\mathbb{E}[L_n] \geq \frac{\varepsilon\, n}{k} \sum_a \Delta_a = \Omega(n).$$

Linear regret means the *per-step* penalty never vanishes: however long Reacher runs, it keeps paying the same exploration tax forever. Any algorithm worth using achieves sublinear regret, so that the average penalty tends to zero. Decaying ε toward zero — slowly enough that every action is still tried infinitely often — recovers sublinearity, and is the seed of the **GLIE** conditions Chapter 6 needs for Q-learning to converge.

3.4 UCB: deriving optimism from a concentration bound

The most elegant answer to "which action should I try?" comes from asking a sharper question: **which action could plausibly be best, given what I have seen?**

Hoeffding's inequality gives the tool. For n independent samples of a bounded random variable with mean μ and sample mean $\hat{\mu}$,

$$\mathbb{P}\big(\hat{\mu} + u \leq \mu\big) \leq e^{-2nu^2}.$$

Set the failure probability to $p = e^{-2nu^2}$ and solve for the bonus:

$$u = \sqrt{\frac{-\log p}{2n}}.$$

Now choose how confident to be. If we want the bound to hold more tightly as we gather data — say $p = t^{-4}$ at time t — then $-\log p = 4\log t$ and

$$u = \sqrt{\frac{2\log t}{n}}.$$

That is the UCB bonus. The rule is: act greedily with respect to an *optimistic* estimate,

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right].$$

WHY OPTIMISM IS THE RIGHT INSTINCT

The bonus term is large when $N_t(a)$ is small — an action tried rarely could still be excellent, and the algorithm gives it the benefit of the doubt. As $N_t(a)$ grows the bonus shrinks and the estimate must stand on its own merits. Meanwhile $\ln t$ in the numerator grows slowly, so an action ignored for a long time gradually becomes attractive again.

The result is exploration that is *directed* rather than random. ε-greedy explores by occasionally throwing away the decision; UCB explores by preferring actions whose value it is genuinely uncertain about. Auer and colleagues proved this achieves regret $O(\sum_a \frac{\log n}{\Delta_a})$ — logarithmic, and matching the lower bound of Lai and Robbins up to constants.

INTERACTIVE · `ch03-bandit-testbed`

The 10-armed testbed

Average reward, percentage of optimal actions, and cumulative regret for ε-greedy, optimistic initialization, UCB, gradient bandits and Thompson sampling, averaged over independent runs.

Run this simulation in the web edition: `/chapters/multi-armed-bandits #ch03-bandit-testbed`

Three things are worth doing in that testbed before reading on. Set $\varepsilon = 0$ and watch the greedy curve plateau below every other method — it locks onto whichever arm happened to look good first. Raise ε and observe that early reward *drops* while the percent-optimal curve keeps climbing: exploration is paid for immediately and repaid later. Finally, compare the regret panel: UCB's curve visibly bends toward logarithmic while ε-greedy's stays a straight line, exactly as Proposition 3.1 predicts.

3.5 Gradient bandits: the seed of everything in Chapter 10

So far we have estimated values and acted greedily on them. There is a different approach: learn a **preference** $H_t(a)$ for each action, with no interpretation as a reward estimate, and act stochastically according to a softmax:

$$\pi_t(a) \doteq \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}}.$$

This is the book's first **stochastic policy**, and it is a genuinely different object from Q. Preferences are only meaningful relative to each other; adding a constant to all of them changes nothing.

The update rule is:

$$H_{t+1}(a) = H_t(a) + \alpha \left(R_t - \bar{R}_t\right) \left(\mathbb{1}[a = A_t] - \pi_t(a)\right),$$

where $\bar{R}_t$ is a running average of all rewards, serving as a **baseline**. This is not a heuristic — it is exact stochastic gradient ascent on expected reward. In expectation,

$$\mathbb{E}\left[H_{t+1}(a) - H_t(a)\right] = \alpha \frac{\partial\, \mathbb{E}[R_t]}{\partial H_t(a)}.$$

WHY THE BASELINE CHANGES VARIANCE BUT NOT DIRECTION

The key step is that the baseline term vanishes in expectation. Since $\sum_a \pi_t(a) = 1$, differentiating both sides with respect to $H_t(a)$ gives $\sum_b \frac{\partial \pi_t(b)}{\partial H_t(a)} = 0$. Therefore, for **any** baseline B_t that does not depend on the action taken,

$$\sum_b B_t \frac{\partial \pi_t(b)}{\partial H_t(a)} = B_t \sum_b \frac{\partial \pi_t(b)}{\partial H_t(a)} = 0.$$

Subtracting $\bar{R}_t$ therefore leaves the expected update direction untouched. What it *does* change is the variance of the sample estimate: if all rewards are around +10, then without a baseline every action gets its preference pushed up, and only the differences carry signal. Subtracting the mean centres the signal, and the noise shrinks accordingly. $$ ■

> **REMEMBER THIS PROOF**
>
> You have just seen, in its simplest possible setting, the argument that Chapter 10 will make for advantage estimation in full reinforcement learning. The score-function trick, the baseline that reduces variance without introducing bias, the stochastic policy — all of it is here, on a problem with one state. When the policy gradient theorem arrives with states and returns and discounting, the skeleton will be this.

3.6 Thompson sampling, and what comes next

One more algorithm, because it is both the oldest (1933) and among the best performing in practice. **Thompson sampling** maintains a posterior distribution over each action's value, then at each step draws one sample from each posterior and acts greedily on the samples.

The elegance is that exploration falls out for free. An action with a wide posterior sometimes draws a high sample and gets tried; as evidence accumulates its posterior narrows and it is chosen only if it deserves to be. There is no bonus term to tune — the uncertainty *is* the exploration mechanism.

Contextual bandits add the one ingredient we have been suppressing: a context x_t observed before choosing. Reacher now sees the mug's estimated pose and picks a grasp conditioned on it. Regret is redefined against the best *policy* in a class rather than the best fixed action.

And here is the cliff the chapter ends on, stated precisely because it defines the rest of the book:

> **⚠ THE MOMENT A BANDIT STOPS BEING A BANDIT**
>
> In a contextual bandit, the context process is **independent of your actions**. The next mug arrives however it arrives, regardless of what you just did. The instant your action influences the next context — grasping the mug *moves* it, driving down a corridor *changes* which corridor you face next — the problem is no longer a bandit. Your choice now has consequences that outlive its immediate reward, and maximizing immediate reward becomes provably suboptimal.
> That requirement has a name, and Chapter 4 derives its equations.

RUST **rl-core/src/bandit.rs**

```rust
use rand::Rng;

/// A k-armed bandit problem with unknown true action values.
pub trait Bandit {
    fn arms(&self) -> usize;
    fn pull<R: Rng>(&self, arm: usize, rng: &mut R) -> f64;
    fn optimal_arm(&self) -> usize;
}

/// Any rule for choosing among arms and learning from what it observes.
pub trait BanditPolicy {
    fn select<R: Rng>(&mut self, step: usize, rng: &mut R) -> usize;
    fn update(&mut self, arm: usize, reward: f64);
}

pub struct Ucb1 {
    q: Vec<f64>,
    counts: Vec<u64>,
    c: f64,
}

impl BanditPolicy for Ucb1 {
    fn select<R: Rng>(&mut self, step: usize, _rng: &mut R) -> usize {
        // Untried arms have an infinite bound -- try each one once first.
        if let Some(a) = self.counts.iter().position(|&n| n == 0) {
            return a;
        }
        let t = (step.max(2)) as f64;
        self.q
            .iter()
            .zip(&self.counts)
            .map(|(&q, &n)| q + self.c * (t.ln() / n as f64).sqrt())
            .enumerate()
```

```
            .max_by(|a, b| a.1.partial_cmp(&b.1).unwrap())
            .map(|(a, _)| a)
            .unwrap()
    }

    fn update(&mut self, arm: usize, reward: f64) {
        self.counts[arm] += 1;
        // Robbins-Monro with alpha = 1/n: the sample average, computed incrementally.
        self.q[arm] += (reward - self.q[arm]) / self.counts[arm] as f64;
    }
}
```

The book's first abstraction. Note the shape: a policy selects, observes, and updates — the same three-verb interface that every agent in this book implements, all the way to PPO in Chapter 10.

3.7 Chapter bridge

We isolated the exploration–exploitation dilemma by removing state, and found that even then the problem has real mathematical content: regret decomposes into gaps times counts, optimism in the face of uncertainty is derivable from a concentration inequality rather than merely sensible, and a softmax over learned preferences performs exact gradient ascent on expected reward.

Chapter 4 puts state back. Once actions influence what you face next, "the value of an action" must account for the future it leads to, and the equations that express this — the Bellman equations — are the mathematical centre of the entire subject. Rusty's warehouse becomes a formal object, and the contraction theorem from Chapter 2 finally does the job it was built for.

Exercises

1 FOUNDATION • **Derive the incremental update**

Reproduce the algebra from §3.2 turning the sample average into Q_{n+1} = Q_n + (1/n)(R_n − Q_n), showing every step. Then do the same for a weighted average where recent rewards count double.

2 FOUNDATION •• **Recency weights sum to one**

For the constant-α update, verify that the weights $(1-\alpha)^n$ on Q_1 plus $\Sigma\ \alpha(1-\alpha)^{n-i}$ sum to exactly 1. Why does this matter for the interpretation of Q as an average?

3 FOUNDATION ••• **UCB from scratch**

Rederive the UCB bonus from Hoeffding's inequality, this time choosing the failure probability p = t^{−2} instead of t^{−4}. How does the constant change, and what does that mean for how aggressively the rule explores?

4 FOUNDATION •• **The baseline argument**

Prove that $\Sigma_b \partial\pi(b)/\partial H(a) = 0$ for the softmax, and use it to show that ANY action-independent baseline leaves the expected gradient-bandit update unchanged. Then construct a baseline that would break this — what property does it violate?

5 CONCEPTUAL •• **The optimism transient**

In the testbed, run optimistic greedy ($Q_1 = 5$) against ε-greedy. Optimistic initialization wins early then is overtaken. Explain the mechanism: what is optimism doing at step 20 that it has stopped doing at step 500?

6 CONCEPTUAL •• **Find where UCB overtakes**

Determine the step at which UCB's cumulative regret first drops below ε-greedy's at $\varepsilon = 0.1$. Then repeat at $\varepsilon = 0.01$. Explain the direction the crossover moves and why.

7 PRACTICAL ••• **Implement Thompson sampling**

Add a Thompson sampling policy for Gaussian rewards with known variance, using conjugate Normal updates. Race it against UCB in the parameter study. Does the ranking depend on the number of arms?

8 PRACTICAL •• **A non-stationary bandit**

Make the true values $q^*(a)$ drift by a random walk each step. Compare sample-average against constant-α estimation on this problem. The winner should reverse relative to the stationary case — explain the reversal in terms of the Robbins–Monro conditions.

3.8 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapter 2 in full: §2.1 the k-armed bandit, §2.4 incremental implementation, §2.6 optimistic initial values, §2.7 UCB, §2.8 gradient bandits, §2.9 associative search.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§3.2 the curse of real-world samples — why a wasted pull costs so much more on hardware than in simulation.

FURTHER READING AND MODERN SOURCES

Lai, T. L. & Robbins, H. (1985). Asymptotically efficient adaptive allocation rules *Advances in Applied Mathematics 6(1)*.
The logarithmic regret lower bound — proof that UCB is optimal in order, not merely good.

Auer, P., Cesa-Bianchi, N. & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem *Machine Learning 47*.
UCB1 and its finite-time regret bound, derived in the form used in §3.4.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples *Biometrika 25*.
Ninety years old and still competitive. Worth reading for the framing alone.

Lattimore, T. & Szepesvári, C. (2020). Bandit Algorithms *Cambridge University Press*.
The comprehensive modern treatment. Where to go for the proofs this chapter states but does not carry to completion.

Li, L., Chu, W., Langford, J. & Schapire, R. E. (2010). A Contextual-Bandit Approach to Personalized News Article Recommendation *WWW 2010*.
LinUCB — the contextual extension sketched in §3.6.

CHAPTER 4

Markov Decision Processes: The Formalism

> *The Markov property recapitulates the idea of state — a state is a sufficient statistic for predicting the future.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · TU Delft · Carnegie Mellon · TU Darmstadt
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Chapter 3 removed state to isolate exploration. Now we put it back, and the whole subject changes character. Once an action influences what you face next, the value of an action must account for the future it leads to — and that self-reference, embraced rather than avoided, is the Bellman equation. This chapter derives it in full, proves the fixed-point machinery of Chapter 2 applies to it, and then confesses the thing most textbooks postpone: robots never see the state, so the honest formalism is a POMDP.

FOUNDATION

The finite MDP tuple, returns and discounting, value functions, Bellman expectation and optimality equations derived in full, γ-contraction proofs, and the belief-MDP construction.

CONCEPTUAL

Rusty's warehouse as a formal object: click any cell to read its transition distribution, and watch the optimal policy respond to the model rather than to a hyperparameter.

PRACTICAL

The rl-core gym — Space, Env and Mdp traits — the abstraction every later chapter builds on, plus exact value solving by linear algebra.

AFTER THIS CHAPTER YOU CAN

- Write down a finite MDP and compute its dynamics, expected rewards, and returns from the four-argument function p(s′,r | s,a)
- Derive the Bellman expectation equations for v_π and q_π line by line, naming every step
- Derive the Bellman optimality equations and explain exactly where linearity dies
- Prove that both Bellman operators are γ-contractions, and cash the Chapter 2 theorem for existence and uniqueness
- Explain what a belief state is and why partial observability is the normal case in robotics, not the exception
- Recognize the same robot task as a different MDP depending on the action-space level you choose

4.1 The finite MDP

A **finite Markov decision process** is a tuple $(\mathcal{S}, \mathcal{A}, p, \gamma)$ where $\mathcal{S}$ and $\mathcal{A}$ are finite sets of states and actions, $\gamma \in [0, 1)$ is a discount factor, and the dynamics are captured by a single function

$$p(s', r \mid s, a) \doteq \mathbb{P}\big(S_{t+1} = s',\ R_{t+1} = r \mid S_t = s,\ A_t = a\big),$$

satisfying $\sum_{s'} \sum_r p(s', r \mid s, a) = 1$ for every (s, a). Everything else is derived from it. Marginalizing the reward gives the state-transition probabilities,

$$p(s' \mid s, a) = \sum_r p(s', r \mid s, a),$$

and weighting by reward gives the expected immediate reward,

$$r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] = \sum_r r \sum_{s'} p(s', r \mid s, a).$$

Note the indexing convention, inherited from Chapter 1: the reward R_{t+1} arrives *after* action A_t. It is a consequence, not a property of the state you were in.

WHERE TO DRAW THE AGENT–ENVIRONMENT BOUNDARY

The boundary sits at the limit of the agent's *arbitrary control*, not at the robot's skin. Rusty's motors are part of the environment: he commands a wheel velocity, and whether the wheels deliver it depends on friction, battery voltage and load. Anything the agent cannot change at will by fiat

> belongs on the environment side of the line.
> This is not pedantry. It is why "the robot's own body" appears in the transition function, and why Chapter 13 spends a chapter on what that body actually does.

Rusty's warehouse, fixed once for the rest of Part I. A 12×9 grid with shelf blocks removed, leaving about 80 traversable cells. Four actions $\{N, E, S, W\}$. Slip probability $p_{\text{slip}} = 0.2$, split evenly between the two lateral directions. Rewards: +25 for reaching the dock, -1 per step, -10 for bumping a shelf. Discount $\gamma = 0.95$. Chapters 5, 6 and 7 use these exact numbers, so results are comparable across algorithms.

> INTERACTIVE · ch04-mdp-editor
>
> **The warehouse as an MDP**
>
> Click any floor cell to read its full transition distribution $p(s', r \mid s, a)$ and the four action values. Changing γ or p_{slip} re-solves the MDP live.
>
> Run this simulation in the web edition: `/chapters/markov-decision-processes #ch04-mdp-editor`

4.2 Returns, and why we discount

The agent's goal is not to maximize immediate reward but the **return** — the accumulated reward from time t onward:

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

For a continuing task this is an infinite sum, and we need it to be finite. If rewards are bounded by $R_{\max}$, the geometric series gives

$$|G_t| \le \sum_{k=0}^{\infty} \gamma^k R_{\max} = \frac{R_{\max}}{1-\gamma},$$

which is finite exactly because $\gamma < 1$. Discounting is not a modelling nicety; without it the objective may not be defined.

The single most useful property of the return is its recursion, obtained by factoring γ out of everything after the first term:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \cdots \right) \\ &= R_{t+1} + \gamma \, G_{t+1}. \end{aligned}$$

Everything that follows in this book is, in some sense, this one line taken seriously.

> **γ IS A MODELLING DECISION, NOT A HYPERPARAMETER**
>
> The quantity $1/(1-\gamma)$ is the **effective horizon** — roughly how many steps into the future the agent meaningfully cares about. At $\gamma = 0.9$ that is 10 steps; at $\gamma = 0.99$ it is 100.
>
> For a robot this is a physical statement. A battery-limited patrol has a real horizon; setting γ to match it is modelling. Setting $\gamma = 0.999$ because it made a benchmark number look better is not. Drag γ in the widget above and watch how far value propagates from the dock — that reach *is* the horizon, drawn.

4.3 Value functions and the Bellman expectation equations

A **policy** $\pi(a \mid s)$ is a distribution over actions in each state. The **state-value function** is the expected return from following it:

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

and the **action-value function** fixes the first action before following π thereafter:

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

Now the derivation. Every step is either the return recursion, the tower property from Chapter 2, or the definition of expectation.

> **Theorem 4.1** — Bellman expectation equation for v_π. For any policy π and all $s \in \mathcal{S}$,
>
> $$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma \, v_\pi(s') \Big].$$

PROOF

Start from the definition and substitute the return recursion $G_t = R_{t+1} + \gamma G_{t+1}$:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s \right].$$

Expand the expectation over the immediate randomness — which action π chooses, and which (s', r) the environment returns:

$$= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma \, \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \Big].$$

Two justifications are needed for that line. First, conditioning on $S_{t+1} = s'$ inside the bracket is the **tower property** (Theorem 2.2): we average the return-from-next-step over where we land. Second, replacing $\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s', S_t = s, A_t = a]$ with $\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']$ uses the **Markov property** — given s', the past is irrelevant.

The inner expectation is $v_\pi(s')$ by definition, giving the result. $$ ∎

The same argument applied to q_π yields its twin:

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

This is a linear system. With $|\mathcal{S}|$ unknowns and $|\mathcal{S}|$ equations, we can write it in matrix form as $v_\pi = r_\pi + \gamma P_\pi v_\pi$ and solve directly:

$$v_\pi = (I - \gamma P_\pi)^{-1} r_\pi,$$

with the inverse guaranteed to exist by the Neumann-series argument from §2.6. For Rusty's 80-state warehouse this is a trivial linear solve. For a robot with a continuous state space it is hopeless, which is what drives Part II.

4.4 Optimality, and the moment linearity dies

Define a partial order on policies: $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for **all** states. There always exists an **optimal policy** π_* that is at least as good as every other, and all optimal policies share the same value functions:

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

The optimal value function satisfies a self-consistency condition of its own — but now with a maximum where the policy average used to be.

Theorem 4.2 — Bellman optimality equations.

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

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

The argument is that $v_*(s)$ must equal the value of the best action available at s, evaluated under optimal behaviour thereafter. If some action had a higher one-step-lookahead value, the policy that took it and then behaved optimally would beat π_* — contradiction.

Two consequences follow immediately, and they are the reason this equation is worth its fame.

Greedy is optimal. Any policy that acts greedily with respect to v_* is optimal. A one-step lookahead using v_* already accounts for all future consequences, because that is what v_* encodes. With q_* it is even easier — no lookahead at all, just $\pi_*(s) = \arg\max_a q_*(s, a)$. This is why so much of RL is the pursuit of q_*: get it, and the policy is free.

A deterministic optimal policy always exists for a finite MDP: pick any argmax at each state.

> **⚠ WHERE THE TROUBLE STARTS**
>
> The max operator destroys linearity. There is no matrix inverse for the optimality equation, no closed-form solve. For $|\mathcal{S}|$ states with $|\mathcal{A}|$ actions each, the number of deterministic policies is $|\mathcal{A}|^{|\mathcal{S}|}$ — for Rusty's modest warehouse, 4^{80}, which is around 10^{48}.
> Iteration is therefore not a concession to practicality. It is the method, and Chapter 5 is about making it work.

4.5 The contraction, at last

Chapter 2 built the fixed-point machinery. Here is what it was for.

Define the **Bellman optimality operator** T_* on value functions by

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

and the policy-evaluation operator T_π analogously with $\sum_a \pi(a \mid s)$ in place of the max. The Bellman equations say precisely that v_* and v_π are the **fixed points** of these operators.

> **Theorem 4.3** — Both Bellman operators are γ-contractions in the sup norm. For any value functions u, v,
>
> $$\| T_* u - T_* v \|_\infty \le \gamma \, \| u - v \|_\infty,$$
>
> and likewise for T_π.

PROOF

Fix a state s. We use the elementary inequality $\left|\max_a f(a) - \max_a g(a)\right| \le \max_a |f(a) - g(a)|$, which holds because the maximizer of one function is a feasible choice for the other.

$$
\begin{aligned}
|(T_*u)(s) - (T_*v)(s)| &\le \max_a \left| \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma u(s')\Big] - \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v(s')\Big] \right| \\
&= \max_a \left| \gamma \sum_{s'} p(s' \mid s, a)\big[u(s') - v(s')\big] \right| \\
&\le \gamma \max_a \sum_{s'} p(s' \mid s, a)\, |u(s') - v(s')| \\
&\le \gamma \max_a \sum_{s'} p(s' \mid s, a)\, \|u - v\|_\infty = \gamma\, \|u - v\|_\infty,
\end{aligned}
$$

using $\sum_{s'} p(s' \mid s, a) = 1$ in the last step. The rewards cancel exactly because they do not depend on the value function. Taking the maximum over s on the left gives the claim. $$ ∎

Now cash Chapter 2's theorem. The space of bounded value functions with the sup norm is complete, so Banach applies and delivers, at no additional cost:

1. v_* **exists** and is the unique fixed point of T_*;
2. **iterating** $v_{k+1} = T_* v_k$ **converges to it from any starting point** — including the all-zeros initialization every implementation uses;
3. **convergence is geometric at rate** γ, with a computable error bound that tells you when to stop.

That last item is not decoration: it is the stopping rule for value iteration, and Chapter 5's dashboard displays it live.

> **THE PUN RESOLVED**
>
> Chapter 2 noted that the contraction modulus and the discount factor share the letter γ, and promised it was not a coincidence. Theorem 4.3 is the promised statement: **the Bellman operator's contraction modulus IS the discount factor.**
> Discounting the future is what makes the mathematics converge. Push $\gamma \to 1$ and the contraction weakens, convergence slows, and the error bound $2\gamma\Delta/(1-\gamma)$ blows up. A robot that cares equally about all future time is a robot whose planning problem may be ill-posed.

4.6 The confession: robots never see the state

Everything above assumes the agent observes S_t. Rusty does not. He has wheel encoders that drift, a lidar that returns noisy ranges, and no oracle telling him which cell he occupies.

The honest formalism is a **partially observable MDP** — an MDP plus an observation space Ω and an observation model $O(o \mid s', a)$. The agent sees o_t, never s_t.

The standard repair is to maintain a **belief** $b_t(s) = \mathbb{P}(S_t = s \mid o_{1:t}, a_{1:t-1})$, a distribution over states. It updates by Bayes' rule: after taking a and observing o,

$$b'(s') = \frac{O(o \mid s', a) \sum_s p(s' \mid s, a)\, b(s)}{\sum_{s''} O(o \mid s'', a) \sum_s p(s'' \mid s, a)\, b(s)},$$

where the numerator predicts forward and reweights by the observation likelihood, and the denominator normalizes.

The remarkable fact is that **the belief is Markov even though the observation is not**: b_{t+1} depends only on b_t, a_t and o_{t+1}. So a POMDP is an MDP over beliefs — all the theory above still applies. The catch is that the belief space is continuous even when $\mathcal{S}$ is finite, and solving POMDPs exactly is PSPACE-hard.

> **⚠ WHAT PRACTITIONERS ACTUALLY DO**
>
> Nobody solves belief MDPs exactly on robots. The three practical answers, each developed later in this book:
>
> **Enlarge the observation** until it is approximately Markov — stack recent frames, append velocities. Cheap, and works more often than it has any right to.
>
> **Learn a belief tracker** — give the policy a recurrent network and let it discover what to remember. Chapter 19 does this for lidar navigation.
>
> **Train with privileged information** — let a teacher policy see the true state in simulation, then distill into a student that sees only what the real robot sees. This is the ANYmal recipe, and Chapter 15 builds it.

4.7 The same robot, different MDPs

One more idea before the code, because it reframes everything: **the MDP is a modelling choice, not a property of the robot.**

Tang and colleagues organize this along three axes. The **action-space level** may be low (joint torques), mid (task-space velocity commands), or high (temporally extended subroutines). The **observation space** may be a low-dimensional

estimated state vector or raw high-dimensional sensing. The **reward** may be sparse or dense.

Each combination is a *different MDP for the same physical task*. Rusty at grid level — this chapter — has 80 states and four actions. Rusty at velocity level has a continuous state and continuous actions. Rusty at torque level, Chapter 13's territory, adds motor dynamics.

> **DIFFICULTY MOVES; IT DOES NOT VANISH**
>
> Choosing a high-level action space makes the RL problem easier — but only because a hand-written low-level controller is absorbing the difficulty. The composite system is exactly as complicated as it was; you have decided which part learns and which part you write.
>
> That decision has more effect on whether a robot project succeeds than the choice of algorithm. Chapter 17 studies it directly, and Chapters 18–20 show what each competency actually settled on.

RUST `rl-core/src/env.rs`

```rust
/// A set an observation or action can be drawn from.
pub trait Space {
    type Item;
    fn contains(&self, x: &Self::Item) -> bool;
    fn sample<R: rand::Rng>(&self, rng: &mut R) -> Self::Item;
}

pub struct Transition<S> {
    pub next_state: S,
    pub reward: f64,
    pub done: bool,
}

/// The black-box view: you may sample the dynamics, never inspect them.
/// This is all a real robot ever offers.
pub trait Env {
    type State: Clone;
    type Action: Copy;

    fn reset<R: rand::Rng>(&mut self, rng: &mut R) -> Self::State;
    fn step<R: rand::Rng>(
        &mut self,
        state: &Self::State,
        action: Self::Action,
        rng: &mut R,
    ) -> Transition<Self::State>;
    fn gamma(&self) -> f64;
}

/// The white-box view: the full distribution p(s', r | s, a) is available.
/// Simulators can offer this; reality cannot.
pub trait Mdp: Env {
    fn states(&self) -> &[Self::State];
    fn actions(&self, state: &Self::State) -> &[Self::Action];
```

```rust

    /// Every (next_state, reward, probability) triple, summing to 1.
    fn transitions(
        &self,
        state: &Self::State,
        action: Self::Action,
    ) -> Vec<(Self::State, f64, f64)>;
}
```

The book's gym. Note that Mdp extends Env with the transition distribution: Chapter 5 plans against the white-box view, Chapter 6 learns against the black-box one, and the same warehouse implements both.

4.8 Chapter bridge

The formalism is complete. An MDP is four objects; the return has a recursion; value functions satisfy Bellman equations; those equations are fixed-point conditions for operators that contract at rate γ; and Banach hands us existence, uniqueness, and a convergent algorithm with a stopping rule.

We also admitted that the state is not observable, that solving the honest version is intractable, and that the MDP itself is something an engineer chooses rather than discovers.

Chapter 5 takes the algorithm Banach promised and makes it real. With p and r known, we can compute v_* by iteration and watch value ripple outward from Rusty's dock — and we will find that evaluation and improvement, alternated, form a pattern so general that every remaining algorithm in this book is a variation on it.

Exercises

1 FOUNDATION • **Derive the marginals**

From p(s′,r | s,a), derive expressions for p(s′ | s,a), r(s,a), and r(s,a,s′). Then compute all three by hand for a slip cell of Rusty's warehouse using the constants in §4.1.

2 FOUNDATION •• **The return bound is tight**

Prove $|G_t| \le R_{max}/(1-\gamma)$ and construct an MDP where the bound is attained exactly. What does that MDP look like physically?

3 FOUNDATION •• **Bellman for q_π**

Carry out the derivation of the Bellman expectation equation for q_π in the same detail as Theorem 4.1, naming where the tower property and the Markov property are each used.

4 FOUNDATION •• **The max inequality**

Prove the lemma used in Theorem 4.3: $|\max_a f(a) - \max_a g(a)| \le \max_a |f(a) - g(a)|$. Where would the contraction proof fail without it?

5 CONCEPTUAL •• **Find the risk-aversion threshold**

In the MDP explorer, start at p_slip = 0 and increase it. Find the slip probability at which the optimal policy first stops hugging the shelves. Explain the trade-off in terms of the q-values shown for that cell.

6 CONCEPTUAL • **Horizon and reach**

Set γ to 0.5, 0.9, and 0.99 in the explorer and record how many cells away from the dock the value function remains meaningfully non-zero. Compare against the effective horizon $1/(1-\gamma)$.

7 PRACTICAL •• **Exact policy evaluation**

Implement solve_v_pi using the matrix form $v_\pi = (I - \gamma P_\pi)^{-1} r_\pi$ with nalgebra, and verify it satisfies the Bellman equation pointwise to 1e−10 with a property test over random MDPs.

8 PRACTICAL ••• **A POMDP wrapper**

Write NoisyOdometryEnv: a wrapper that hides the true state and emits a noisy observation. Implement a discrete Bayes filter over cells, and measure how localization entropy evolves after a "kidnapping" that teleports Rusty without telling him.

4.9 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapter 3 in full: §3.1 the agent–environment interface and the Markov property, §3.3 returns, §3.5 policies and value functions, §3.6 optimal value functions.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§1.3 and §2 — the state-as-sufficient-statistic framing, and why robotic state is never fully observed.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.2 problem formulation — the action-space level, observation space, and reward-density axes used in §4.7.

FURTHER READING AND MODERN SOURCES

Bellman, R. (1957). Dynamic Programming *Princeton University Press*.
The origin of the principle of optimality and the equations that carry his name.

Puterman, M. L. (1994). Markov Decision Processes: Discrete Stochastic Dynamic Programming *Wiley*.
The definitive reference. Existence of optimal stationary deterministic policies, and the contraction analysis, in complete rigour.

Kaelbling, L. P., Littman, M. L. & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains *Artificial Intelligence 101(1–2)*.
The belief-MDP construction of §4.6, and the classical algorithms for exact POMDP solution.

Papadimitriou, C. H. & Tsitsiklis, J. N. (1987). The Complexity of Markov Decision Processes *Mathematics of Operations Research 12(3)*.
The PSPACE-hardness result that explains why nobody solves POMDPs exactly on hardware.

CHAPTER 5

Dynamic Programming: Planning with a Known Model

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*
>
> **Richard Bellman** · RAND Corporation · The principle of optimality
> *Dynamic Programming, 1957*

Suppose Rusty is handed the warehouse blueprint: the exact transition probabilities and rewards from Chapter 4. No sensors needed, no trial and error — finding the best policy becomes pure computation. But the naive computation, enumerating all 4^80 policies, is beyond astronomical. This chapter shows how Bellman structure replaces enumeration with iteration, proves that greedy improvement can never make a policy worse, and extracts the pattern — generalized policy iteration — that every remaining algorithm in this book is a variation on.

FOUNDATION

Iterative policy evaluation, the policy improvement theorem with proof, policy and value iteration, convergence rates, the Bellman-error bound, and asynchronous DP.

CONCEPTUAL

The book's signature dashboard: value rippling outward from the dock sweep by sweep, with Δ and the suboptimality bound streaming beside it.

PRACTICAL

rl-tabular's DP module operating on the Chapter 4 Mdp trait, with the sweep-by-sweep generator that drives the dashboard.

◎ AFTER THIS CHAPTER YOU CAN

- Implement iterative policy evaluation and explain why in-place sweeps converge faster than two-array ones
- Prove the policy improvement theorem and understand why it makes greedy improvement safe
- Explain the difference between policy iteration and value iteration as truncation of the same process
- Use the Bellman-error stopping rule to bound how suboptimal your current policy is
- State generalized policy iteration and identify it in every later algorithm in the book
- Quantify why dynamic programming cannot scale to a real robot

5.1 The blueprint changes everything

Overnight, Rusty receives the warehouse blueprint — the exact MDP from Chapter 4. The 12×9 grid, $p_{\text{slip}} = 0.2$, rewards of $+25$ at the dock, -1 per step, -10 per shelf bump, and $\gamma = 0.95$.

With p and r in hand, "what should Rusty do?" is no longer an experimental question. It is arithmetic. But which arithmetic?

The direct approach is to enumerate every deterministic policy, evaluate each, and keep the best. With roughly 80 states and four actions that is $4^{80} \approx 10^{48}$ policies. At a billion evaluations per second you would finish some 10^{30} times after the heat death of the universe.

Dynamic programming is the family of methods that exploits Bellman structure to replace that enumeration with iteration. The key insight is Bellman's own principle of optimality, quoted above: an optimal policy's tail must itself be optimal. That means we can build the answer up from local consistency conditions rather than searching the space of whole policies.

5.2 Iterative policy evaluation

Start with the easier problem: given a policy π, what is v_π?

Chapter 4 showed this is a linear system solvable by matrix inversion, at cost $O(|\mathcal{S}|^3)$. Iteration is cheaper. Turn the Bellman expectation equation into an assignment:

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

which is exactly applying the operator T_π. Theorem 4.3 proved T_π is a γ-

contraction, so by Banach the sequence converges to the unique fixed point v_π from any initialization — including all zeros.

> **IN-PLACE SWEEPS CONVERGE FASTER**
>
> The equation above describes a *two-array* (Jacobi) update: compute all new values from the old array, then swap. The alternative is to update in place, so later states in the sweep already see the freshly-computed values of earlier ones — a Gauss–Seidel sweep.
> In-place converges strictly faster and uses half the memory, and it still converges to v_π: the operator remains a contraction. It is also what makes *asynchronous* DP possible (§5.6). Every implementation in this book sweeps in place.

When do we stop? Track $\Delta = \max_s |v_{k+1}(s) - v_k(s)|$ and halt when it falls below a threshold θ. Banach's bound converts Δ into a guarantee: if the greedy policy with respect to v_k is π, then

$$\|v_\pi - v_*\|_\infty \le \frac{2\gamma\,\Delta}{1-\gamma}.$$

This is not decoration. It is the difference between "the numbers stopped changing much" and "my policy is provably within ϵ of optimal", and the dashboard below displays it live.

5.3 The policy improvement theorem

Now the result that makes everything work. Suppose we have v_π and construct a new policy π' that acts greedily with respect to it:

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

Is π' better? It is *locally* better by construction — but each state's improvement changes which states you visit, so the argument is not obvious. It is, however, true.

> **Theorem 5.1** — Policy improvement theorem. Let π and π' be deterministic policies such that for all s,
>
> $$q_\pi(s, \pi'(s)) \ge v_\pi(s).$$
>
> Then π' is at least as good as π: $v_{\pi'}(s) \ge v_\pi(s)$ for all s. If the first inequality is strict at any state, the second is strict at that state too.

PROOF

Start from the hypothesis and expand q_π repeatedly, each time replacing v_π by the hypothesis applied one step further along:

$$\begin{aligned}
v_\pi(s) &\le q_\pi(s, \pi'(s)) \\
&= \mathbb{E}\left[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,|\, S_t = s,\, A_t = \pi'(s)\right] \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,|\, S_t = s\right] \\
&\le \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma\, q_\pi(S_{t+1}, \pi'(S_{t+1})) \,\middle|\, S_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma\, \mathbb{E}\left[R_{t+2} + \gamma v_\pi(S_{t+2})\right] \,|\, S_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \,\middle|\, S_t = s\right] \\
&\;\;\vdots \\
&\le \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \,\middle|\, S_t = s\right] = v_{\pi'}(s).
\end{aligned}$$

Each inequality applies the hypothesis at the newly-exposed state; each equality is the tower property. The $\gamma^k v_\pi(S_{t+k})$ remainder term vanishes as $k \to \infty$ because v_π is bounded and $\gamma < 1$ — the same geometric argument as everywhere else in this book. $$ ■

WHY THIS THEOREM IS THE LOAD-BEARING WALL

Acting greedily with respect to your current value estimate can never make things worse. That single guarantee is what licenses the entire pattern: evaluate, improve, repeat, without fear of oscillation or regress.

And when improvement produces *no* change — when $\pi' = \pi$ — the greedy equation becomes the Bellman optimality equation, so $v_\pi = v_*$ and the policy is optimal. The algorithm's own stopping condition certifies optimality. Very few algorithms in machine learning have that property, and it is worth appreciating before Part II takes it away.

5.4 Policy iteration and value iteration

Policy iteration alternates the two operations to convergence:

$$\pi_0 \xrightarrow{\text{evaluate}} v_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluate}} v_{\pi_1} \xrightarrow{\text{improve}} \cdots \longrightarrow \pi_*$$

Since each improvement strictly increases the value of at least one state unless the policy is already optimal, and there are finitely many deterministic policies, policy iteration **terminates in finitely many iterations at an exactly optimal policy**. In practice it converges in a handful — often fewer than ten, even on large problems.

The cost is that each evaluation runs to convergence, which is many sweeps of work to answer a question ("which action is best here?") that may not need that precision.

Value iteration takes the opposite extreme: truncate evaluation after a single sweep, folding the improvement into it by using a max instead of a policy average:

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

This is just repeated application of T_*, which Theorem 4.3 proved is a γ-contraction. It converges geometrically, though — unlike policy iteration — it approaches optimality asymptotically rather than hitting it exactly.

INTERACTIVE · **ch05-gpi-dashboard**

Generalized policy iteration, sweep by sweep

Value heatmap and greedy-policy arrows on Rusty's warehouse, advancing one sweep at a time, with Δ and the implied suboptimality bound streaming beside them.

Run this simulation in the web edition: `/chapters/dynamic-programming #ch05-gpi-dashboard`

Watch what happens when you press play on value iteration. Value appears first at the dock, then bleeds outward one cell per sweep. That is not an artifact of the visualization: with $\gamma = 0.95$ and a one-step backup, information about the goal can travel exactly one cell per sweep. Cells far from the dock cannot know anything about it until the wave arrives.

Then switch to policy iteration and watch a different rhythm: long stretches of evaluation where the arrows barely move, punctuated by improvement steps where large regions of the policy flip at once.

RUST **rl-tabular/src/dp.rs**

```rust
use rl_core::Mdp;

pub struct Sweep {
    pub index: usize,
    pub values: Vec<f64>,
    pub delta: f64,
    /// Bellman-error bound on ||v_pi - v*||inf implied by this Delta.
    pub suboptimality: f64,
}

/// Value iteration with in-place (Gauss-Seidel) sweeps.
pub fn value_iteration<M: Mdp>(mdp: &M, theta: f64, max_sweeps: usize) -> Vec<Sweep> {
    let gamma = mdp.gamma();
    let mut v = vec![0.0; mdp.states().len()];
    let mut history = Vec::new();

    for index in 1..=max_sweeps {
        let mut delta: f64 = 0.0;

```

```
        for (i, state) in mdp.states().iter().enumerate() {
            if mdp.is_terminal(state) { continue; }
            let old = v[i];

            // max_a Sigma p(s',r|s,a)[r + gamma v(s')] -- the Bellman optimality backup.
            v[i] = mdp
                .actions(state)
                .iter()
                .map(|&a| {
                    mdp.transitions(state, a)
                        .iter()
                        .map(|&(next, reward, prob)| prob * (reward + gamma * v[next]))
                        .sum::<f64>()
                })
                .fold(f64::NEG_INFINITY, f64::max);

            delta = delta.max((old - v[i]).abs());
        }

        history.push(Sweep {
            index,
            values: v.clone(),
            delta,
            suboptimality: 2.0 * gamma * delta / (1.0 - gamma),
        });

        if delta < theta { break; }
    }
    history
}
```

Value iteration as a sweep-by-sweep iterator. Returning snapshots rather than only the final answer is what lets the dashboard animate the algorithm — and, more importantly, lets you debug it.

5.5 Generalized policy iteration

Step back and notice that policy iteration and value iteration are the same algorithm with a dial set differently.

Both maintain a value estimate and a policy. Both push the value toward consistency with the policy (**evaluation**) and push the policy toward greediness with respect to the value (**improvement**). They differ only in how much evaluation happens between improvements: everything, one sweep, or — in *modified policy iteration* — some number in between.

Generalized policy iteration (GPI) is the name for this pattern in the abstract: any interleaving of evaluation and improvement, at any granularity, in any order, including asynchronously and on subsets of states.

> **THIS IS THE PATTERN FOR THE ENTIRE BOOK**
>
> The two processes compete and cooperate. Improvement makes the value function wrong (it was correct for the old policy). Evaluation makes the

policy non-greedy (values shifted). Each undoes some of the other's work — yet together they converge, because both are pushing toward the same joint fixed point: the value function consistent with a policy that is greedy with respect to it. That joint condition is the Bellman optimality equation. Once you see GPI you will not stop seeing it. Monte Carlo control in Chapter 6 is GPI with sampled evaluation. Q-learning is GPI with one-step sampled backups. Actor–critic in Chapter 10 is GPI with a parameterized policy and a learned critic. SAC in Chapter 11 is GPI with a soft improvement step. The rest of this book largely consists of replacing "evaluation" and "improvement" with things that work without a model, in continuous spaces, from pixels.

5.6 Asynchronous DP, and the wall

Nothing requires sweeping states in order, or sweeping all of them equally. **Asynchronous DP** updates states in any order, as long as every state is updated infinitely often in the limit. Convergence still holds.

This licenses genuinely useful strategies: back up states along a trajectory the robot actually visits, prioritize states whose values just changed a lot (Chapter 7's prioritized sweeping), or focus computation near the goal where it does most good.

But no ordering saves dynamic programming for a real robot. The problem is not the sweep order — it is that a sweep must touch every state, and the number of states is exponential in the robot's degrees of freedom.

INTERACTIVE · `ch14-dimensionality-wall`

The exponential wall

Set the degrees of freedom and bins per dimension, and read how long one exhaustive tabular sweep would take. Seven degrees of freedom at ten bins already outlives the solar system.

Run this simulation in the web edition: `/chapters/dynamic-programming#ch14-dimensionality-wall`

Set the widget to 7 degrees of freedom with 10 bins per dimension — a deliberately crude discretization of a modest research arm — and read the time for a single sweep. No faster computer rescues this; the curve is exponential in the exponent.

This is the first of Kober's four curses, and Chapter 14 develops all four properly. Its consequence for us is immediate and structural: **we must stop enumerating states and start generalizing across them.** That is Part II.

5.7 What DP still buys you

Before leaving, it is worth being clear that dynamic programming is not merely a pedagogical stepping stone.

It remains the correct tool whenever the state space is genuinely small — inventory control, discrete task planning, the high-level layer of a hierarchical robot controller. It is the ground truth against which approximate methods are measured: every algorithm in Chapters 6 through 12 is trying to approximate what DP computes exactly. And its theory transfers wholesale: the contraction argument, the improvement theorem, and GPI survive into the approximate setting, sometimes weakened but always recognizable.

Chapter 12 will bring DP back in a new guise, planning against a *learned* model rather than a given one — Bellman backups over dynamics the robot discovered for itself.

5.8 Chapter bridge

We assumed the blueprint. Rusty does not have one.

Real robots are handed no transition probabilities. Friction is unmeasured, payloads change, and the only access to the dynamics is to act and observe what happens. Chapter 6 removes the model entirely and asks whether anything survives. The answer is that a great deal does — Monte Carlo methods learn from complete episodes, temporal-difference methods learn from single transitions by bootstrapping from their own estimates, and the GPI pattern from this chapter carries over intact with sampling in place of expectation.

Exercises

1 FOUNDATION •• **Jacobi versus Gauss–Seidel**

Prove that the in-place policy-evaluation sweep still converges to v_π. Then construct a small MDP and a state ordering where in-place converges in strictly fewer sweeps, and one where the ordering makes almost no difference.

2 FOUNDATION •• **The improvement theorem, carefully**

Reproduce the proof of Theorem 5.1, justifying every equality and inequality. In particular, state precisely why the remainder term γ^k v_π(S_{t+k}) vanishes and what would break if $\gamma = 1$.

3 FOUNDATION •• **Termination in finite time**

Prove that policy iteration terminates after finitely many improvement steps. Then

give an upper bound on the number of steps in terms of $|S|$ and $|A|$, and explain why the observed number is usually far smaller.

4 FOUNDATION • • • **The stopping bound**

Derive the bound $\|v\_\pi - v^*\|\infty \leq 2\gamma\Delta/(1-\gamma)$ where π is greedy with respect to a value function whose latest sweep changed by at most Δ. Where does the factor of 2 come from?

5 CONCEPTUAL • **The propagation wave**

In the GPI dashboard, run value iteration and count how many sweeps pass before the cell farthest from the dock has a non-zero value. Relate this to the grid's diameter, and explain why one-step backups make this inevitable.

6 CONCEPTUAL •• **Two rhythms**

Compare policy iteration and value iteration on the same problem. Record total sweeps to convergence for each at $\gamma = 0.9$ and $\gamma = 0.99$. Which degrades faster as $\gamma \to 1$, and does the contraction bound predict it?

7 PRACTICAL •• **Modified policy iteration**

Implement modified policy iteration with a configurable number k of evaluation sweeps per improvement. Plot total backups to convergence as a function of k. There should be an interior optimum — find it and explain its existence.

8 PRACTICAL •• **Measure the wall**

Benchmark value iteration on grids from 10×10 up to as large as your machine tolerates, and fit the scaling. Then extrapolate to the 14-dimensional state of a 7-DoF arm and report the number honestly.

5.9 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapter 4 in full: §4.1 policy evaluation, §4.2 policy improvement, §4.3 policy iteration, §4.4 value iteration, §4.5 asynchronous DP, §4.6 generalized policy iteration.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§3.1 the curse of dimensionality — the argument §5.6 turns into arithmetic.

FURTHER READING AND MODERN SOURCES

Bellman, R. (1957). Dynamic Programming *Princeton University Press*.
The principle of optimality, and the coining of "curse of dimensionality" by the person who had the most right to complain about it.

Howard, R. A. (1960). Dynamic Programming and Markov Processes *MIT Press*.
The origin of policy iteration.

Puterman, M. L. (1994). Markov Decision Processes: Discrete Stochastic Dynamic Programming *Wiley*.
Modified policy iteration, convergence rates, and the complete theory of the methods sketched here.

Bertsekas, D. P. (2012). Dynamic Programming and Optimal Control, Vol. II *Athena Scientific, 4th edition*.
Asynchronous DP, and the approximate dynamic programming that bridges to Part II of this book.

CHAPTER 6

Learning from Experience: Monte Carlo & Temporal-Difference

> *If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference learning.*
>
> **Richard S. Sutton & Andrew G. Barto** · University of Alberta · University of Massachusetts Amherst
> *Reinforcement Learning — An Introduction, 2nd edition*

Take away the blueprint. Rusty gets no transition probabilities, no reward function he can inspect — only the ability to act and observe what happens. Remarkably, almost everything survives. Monte Carlo methods learn from complete episodes by averaging returns. Temporal-difference methods do something stranger and more powerful: they update a guess toward another guess, learning from single transitions without waiting for the outcome. This chapter derives both, proves what can be proved, and confronts the systematic optimism that makes Q-learning overestimate everything it values.

FOUNDATION

First-visit MC and its unbiasedness, importance sampling with variance analysis, the TD(0) update, SARSA/Expected SARSA/Q-learning, Watkins' convergence theorem, and the double-estimator argument.

CONCEPTUAL

Rusty learning the warehouse from scratch — value bleeding backwards from the dock as the TD error propagates, with δ and the return streaming live.

PRACTICAL

rl-tabular's model-free learners over the Chapter 4 Env trait, with the update returning δ so dashboards can display the learning signal itself.

AFTER THIS CHAPTER YOU CAN

- Explain why first-visit Monte Carlo is an unbiased estimator of v_π, and what it costs in variance
- Derive the importance-sampling correction for off-policy evaluation and explain why its variance can be infinite
- Write the TD(0) update and identify the bias–variance trade it makes against Monte Carlo
- State the difference between SARSA and Q-learning precisely, in terms of which policy the target evaluates
- Explain maximization bias, prove it occurs even with unbiased estimates, and show how Double Q-learning removes it
- Recognize GLIE conditions and their role in convergence guarantees

6.1 No model, no problem — the Monte Carlo idea

Chapter 5 needed $p(s', r \mid s, a)$ to compute expectations. Without it, one option remains: **sample**.

The value $v_\pi(s)$ is by definition an average of returns. So run episodes, record the returns that actually followed each state, and average them. That is **Monte Carlo prediction**, and its correctness follows from the law of large numbers rather than from any property of the MDP.

First-visit MC averages the returns following the first visit to s in each episode. Its estimator is unbiased: each first-visit return is an independent sample from the distribution whose mean is $v_\pi(s)$, so the average converges to $v_\pi(s)$ as the number of visits grows, with standard error falling as $1/\sqrt{n}$.

WHAT MONTE CARLO DOES NOT NEED

MC makes no use of the Markov property. It never bootstraps — never uses one estimate to update another. Each state's estimate is built from complete, actual returns.

That independence is a genuine advantage when your state representation is imperfect, which for robots it always is. If Rusty's state is only approximately Markov, MC degrades gracefully; methods that bootstrap can compound the error. The cost is that MC must wait for episodes to end, learns nothing

from a run that never terminates, and has high variance because a return accumulates every random event along the whole trajectory.

6.2 Off-policy evaluation and importance sampling

A robot often needs to evaluate one policy while behaving according to another — estimate the value of an aggressive policy while running a safe one, or reuse data collected last week. This is **off-policy** learning, and it is the setting that dominates robotics because data is expensive enough that you want to reuse all of it.

Let b be the **behaviour policy** generating the data and π the **target policy** we want to evaluate. The returns we observe come from the wrong distribution. Importance sampling reweights them by the ratio of trajectory probabilities:

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

Notice that the environment dynamics cancel: $p(s' \mid s, a)$ appears identically in numerator and denominator. Only the policy probabilities survive, which is what makes this usable without a model. Then

$$v_\pi(s) = \mathbb{E}_b \left[\rho_{t:T-1} G_t \mid S_t = s \right],$$

so **ordinary importance sampling** — averaging ρG — is unbiased.

> **⚠ UNBIASED, AND SOMETIMES USELESS**
>
> That product of ratios is the problem. Over a long episode, ρ is a product of many terms; if π and b differ even modestly, the product's variance grows exponentially in the horizon. Sutton and Barto exhibit cases where **the variance is literally infinite** — the estimator is unbiased but its sample average does not settle.
>
> **Weighted importance sampling** divides by $\sum \rho$ instead of by the count. It is biased (the bias vanishes asymptotically) but has dramatically lower — often finite — variance. In practice it wins so consistently that the biased estimator is the default.
>
> The lesson generalizes far beyond this section: unbiasedness is worth much less than practitioners assume, and the variance of an estimator usually decides whether it is usable. Chapter 10 makes exactly this trade again with GAE, and Chapter 16 makes it again with offline RL.

6.3 Temporal difference: learning a guess from a guess

Monte Carlo waits for the episode to end. TD does not, and the trick is disarmingly simple.

The return recursion from Chapter 4 says $G_t = R_{t+1} + \gamma G_{t+1}$. Monte Carlo uses the observed G_t as its target. TD substitutes the current estimate for the tail:

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{\Big[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t) \Big]}_{\text{TD error } \delta_t}.$$

That is TD(0). It updates immediately after one transition, needs no episode boundary, and works on continuing tasks.

The quantity $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the **TD error**, and it deserves a name because it recurs everywhere in this book. It measures the surprise: how much better or worse the transition turned out than the current estimate predicted. When $\delta = 0$ everywhere, the value function is self-consistent and satisfies the Bellman equation.

> **BOOTSTRAPPING IS THE BET**
>
> TD updates an estimate toward a target that is itself partly an estimate. This looks circular, and it is — productively so. The bet is that $V(S_{t+1})$, though wrong, is *less* wrong than a single noisy sampled return, because it already aggregates many past experiences.
>
> The bet usually pays. TD converges faster than MC on most problems, has far lower variance, and learns online. What it buys with that speed is **bias**: if $V(S_{t+1})$ is systematically wrong, the error propagates. And Chapter 8 will show that when function approximation enters, this same bootstrapping is one leg of the "deadly triad" that can make learning diverge outright.

6.4 Control: SARSA, Expected SARSA, Q-learning

To *control* rather than merely predict, learn action values and follow the GPI pattern from Chapter 5: estimate Q, act greedily-ish with respect to it, repeat.

Three algorithms differ only in what they put in the target.

SARSA uses the action actually taken next:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right].$$

It is **on-policy**: the target evaluates the same ε-greedy policy the agent is executing, exploration included.

Q-learning uses the best action available, whether or not it is taken:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right].$$

It is **off-policy**: the target evaluates the greedy policy while the agent behaves ε-greedily.

Expected SARSA replaces the sample with its expectation under the policy:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right],$$

eliminating the variance due to sampling A_{t+1} at the cost of a sum over actions. It usually dominates SARSA, and reduces to Q-learning when π is greedy.

> **⚠ THE CLIFF, AND WHY ROBOTS CARE**
>
> The classic demonstration is a gridworld with a cliff along the optimal path. Q-learning learns the optimal policy — walk the cliff edge — and then, because it *behaves* ε-greedily, occasionally steps off and dies. SARSA learns a safer path further from the edge, because its target accounts for the fact that the behaving policy explores.
>
> SARSA's answer is worse under the MDP's stated objective and better in the world. For a robot that can actually break, that distinction is not academic, and it is why on-policy methods retain a following in robotics long after off-policy methods became more sample-efficient. Chapter 14 returns to this as a safety question, with constrained MDPs as the principled version.

> **INTERACTIVE · `ch06-train-live`**
>
> **Learning without a model**
>
> Rusty learns the warehouse from experience alone. Value bleeds backwards from the dock as the TD error propagates; episode return and mean $|\delta|$ stream alongside.
>
> Run this simulation in the web edition: `/chapters/monte-carlo-and-td #ch06-train-live`

Press play and watch the mechanism directly. Value appears at the dock and bleeds backwards, one cell per episode-ish, because that is how far a one-step TD error can carry information. Early arrows are nonsense — every Q is zero and ties break randomly. The policy sharpens only where the TD error has actually reached.

Then set $\varepsilon = 0$ and watch the run stall. With no exploration, Rusty commits to the first corridor that ever paid off and never discovers the better one. This is the Chapter 3 dilemma, now with states.

6.5 Does it converge?

Yes, under conditions worth knowing precisely.

Theorem 6.1 — Convergence of tabular Q-learning (Watkins & Dayan, 1992). In a finite MDP, tabular Q-learning converges to q_* with probability 1, provided:

1. every state–action pair is visited infinitely often;
2. the learning rates satisfy the Robbins–Monro conditions, $\sum_t \alpha_t(s,a) = \infty$ and $\sum_t \alpha_t^2(s,a) < \infty$ for every pair;
3. rewards are bounded.

The proof combines the two things Chapter 2 built. The Bellman optimality operator is a γ-contraction, so the *expected* update moves Q toward q_*; stochastic approximation theory then shows the noisy sampled updates track the expected ones almost surely, given Robbins–Monro step sizes. The full argument is in Watkins and Dayan and in Bertsekas & Tsitsiklis.

Condition 1 is the one that bites. "Visited infinitely often" requires persistent exploration, which is why ε must decay slowly — the **GLIE** conditions (Greedy in the Limit with Infinite Exploration): explore forever, but let exploration vanish so the policy becomes greedy in the limit. A schedule like $\varepsilon_t = 1/t$ satisfies both.

⚠ WHAT THIS THEOREM DOES NOT SAY

It says nothing about *how fast*, and every hypothesis fails on a real robot. A physical arm cannot visit every state–action pair even once, let alone infinitely often. The state space is continuous, so "tabular" does not apply. And in Part II, when tables become function approximators, this theorem's guarantee evaporates entirely — Chapter 8 shows it can be replaced by divergence.

Know what you have proved. The theorem justifies the algorithm's form and tells you which knobs matter. It does not promise your robot will learn to walk.

6.6 Maximization bias and the double estimator

One more phenomenon, because it explains a family of algorithms in Chapter 9.

Q-learning's target contains $\max_a Q(S_{t+1}, a)$. Suppose all the Q estimates are unbiased but noisy. Is the max unbiased?

No. By Jensen's inequality applied to the convex max function,

$$\mathbb{E}\left[\max_a Q(s,a)\right] \geq \max_a \mathbb{E}\left[Q(s,a)\right] = \max_a q(s,a).$$

Taking the maximum of noisy estimates systematically overestimates the true maximum. The max operator preferentially selects whichever action got lucky, and luck does not persist — but the inflated estimate does, and it propagates through every subsequent backup.

This is **maximization bias**, and it is not a small effect. With many actions and noisy returns, the overestimation compounds through bootstrapping until the value function is meaningfully optimistic everywhere.

Proposition 6.2 — Double estimation removes the bias. Maintain two independent estimates Q_1 and Q_2. Use one to *select* the maximizing action and the other to *evaluate* it:

$$A^* = \arg\max_a Q_1(s,a), \qquad \text{target uses } Q_2(s, A^*).$$

Since Q_2 is independent of the selection, $\mathbb{E}[Q_2(s,A^*)] = q(s,A^*) \leq \max_a q(s,a)$ — the estimate is no longer biased upward.

Double Q-learning implements this by keeping two tables and flipping a coin each step to decide which is updated and which evaluates. It costs twice the memory and nothing in computation, and the fix carries directly into deep RL: Double DQN in Chapter 9 is this exact idea with the target network playing the role of the second estimator, and TD3's clipped double-Q in Chapter 11 is the continuous-control version.

RUST **rl-tabular/src/td.rs**

```rust
use rl_core::{Env, Transition};

pub struct QLearning {
    q: Vec<f64>,          // flat: state * n_actions + action
    n_actions: usize,
    alpha: f64,
    gamma: f64,
}

impl QLearning {
    #[inline]
    fn idx(&self, s: usize, a: usize) -> usize { s * self.n_actions + a }

    fn max_q(&self, s: usize) -> f64 {
        (0..self.n_actions)
            .map(|a| self.q[self.idx(s, a)])
            .fold(f64::NEG_INFINITY, f64::max)
```

```
    }

    /// One off-policy update. Returns the TD error delta -- the learning signal.
    pub fn update(&mut self, s: usize, a: usize, t: &Transition<usize>) -> f64 {
        // The target bootstraps from the GREEDY action, whatever we actually do next.
        let target = if t.done {
            t.reward
        } else {
            t.reward + self.gamma * self.max_q(t.next_state)
        };

        let i = self.idx(s, a);
        let delta = target - self.q[i];
        self.q[i] += self.alpha * delta;
        delta
    }
}
```

The Q-learning update. Returning δ rather than discarding it is a deliberate API choice: the TD error is the learning signal, and every dashboard in this book plots it.

6.7 Chapter bridge

Rusty can now learn without a map. He samples, bootstraps, and improves — the GPI pattern of Chapter 5 with expectations replaced by samples.

But we have accepted a stark choice. Monte Carlo waits for the whole return: unbiased, high variance, no bootstrapping. TD(0) uses one step: low variance, biased, fast. Nothing in the mathematics says those are the only options, and they are not.

Chapter 7 puts a dial between them. The λ-return blends every n-step return at once; eligibility traces make the blend cheap enough to compute online. And then a second idea: if we can learn a *model* from experience, we can plan with it as in Chapter 5 while continuing to learn as in this chapter — Dyna, the architecture that unifies both halves of Part I.

Exercises

1 FOUNDATION •• **First-visit MC is unbiased**

Prove that the first-visit MC estimator of v_π(s) is unbiased. Then explain precisely why the every-visit estimator is biased, and why the bias nonetheless vanishes asymptotically.

2 FOUNDATION ••• **Infinite variance**

Construct a two-state MDP with a behaviour policy b and target π for which ordinary importance sampling has infinite variance. Show the variance integral diverges, then

verify empirically that weighted importance sampling behaves acceptably on the same problem.

3 FOUNDATION •• **MC error as a sum of TD errors**

Prove the identity G_t − V(S_t) = Σ_{k≥t} γ^{k−t} δ_k, assuming V does not change during the episode. This identity is the bridge to Chapter 7 — make sure you can produce it from memory.

4 FOUNDATION •• **Maximization bias by construction**

Build a single-state MDP with several actions whose true values are all zero but whose sampled returns are noisy. Show analytically that Q-learning's estimate is positive in expectation, and compute how it grows with the number of actions.

5 CONCEPTUAL • **The backward wave**

In the live training dashboard, note the episode at which the cell farthest from the dock first acquires a non-zero value under Q-learning. Increase α and repeat. Explain the relationship you observe and its limit.

6 CONCEPTUAL •• **SARSA versus Q-learning under risk**

Set the slip probability high and compare the greedy policies SARSA and Q-learning converge to near the shelves. Which hugs the obstacles? Explain in terms of what each algorithm's target evaluates.

7 PRACTICAL •• **Expected SARSA**

Implement Expected SARSA and compare it against SARSA across learning rates from 0.05 to 0.6. Expected SARSA should tolerate larger α — explain why in terms of the variance of the target.

8 PRACTICAL ••• **Measure the overestimation**

Instrument Q-learning and Double Q-learning to log max_a Q(s,a) against the true q*(s) computed by Chapter 5's value iteration. Plot the gap over training. The single-estimator version should sit visibly above the truth for a long time.

6.8 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapter 5 (Monte Carlo methods, importance sampling) and Chapter 6 (TD learning, SARSA, Q-learning, Expected SARSA, maximization bias).

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§2.2.1 value-function approaches, and §3.2 on why sample cost dominates algorithm choice in robotics.

FURTHER READING AND MODERN SOURCES

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences *Machine Learning 3(1)*.
The paper that introduced TD learning.

Watkins, C. J. C. H. & Dayan, P. (1992). Q-learning *Machine Learning 8*.
The convergence proof of Theorem 6.1.

van Hasselt, H. (2010). Double Q-learning *NeurIPS 23*.
Maximization bias identified and fixed — the ancestor of Double DQN (Chapter 9) and TD3 (Chapter 11).

Rummery, G. A. & Niranjan, M. (1994). On-line Q-learning using connectionist systems *Cambridge University Engineering Department, Technical Report*.
The origin of SARSA, under its original name.

Bertsekas, D. P. & Tsitsiklis, J. N. (1996). Neuro-Dynamic Programming *Athena Scientific*.
The rigorous stochastic-approximation treatment underlying every convergence claim in this chapter.

CHAPTER 7

Unifying Learning & Planning: n-step, Traces, Dyna & MCTS

> *Planning and learning are deeply related. Both involve estimating value functions by backing up updates; the difference is only whether the experience is real or simulated.*
>
> **Richard S. Sutton & Andrew G. Barto** · On the Dyna architecture
> *Reinforcement Learning — An Introduction, 2nd edition*

Chapter 6 left a false dichotomy: Monte Carlo waits for the whole return, TD(0) uses one step. This chapter puts a continuous dial between them — the λ-return blends every n-step return at once, and eligibility traces make that blend computable online with one update per step. Then a second unification: if a robot can learn a model from experience, it can plan with it exactly as in Chapter 5 while still learning as in Chapter 6. Dyna is that architecture, and it is the intellectual ancestor of every model-based method in Chapter 12.

FOUNDATION

n-step returns and the error-reduction property, the λ-return, forward/backward equivalence proved, accumulating traces, Dyna-Q and Dyna-Q+, prioritized sweeping, and UCT.

CONCEPTUAL

The λ dial: drag it from 0 to 1 and watch weight slide from the one-step return onto the full return, with the trace decaying along Rusty's corridor beside it.

PRACTICAL

Sarsa(λ) with traces, Dyna-Q(+) composed from Chapter 6's learner, and prioritized

sweeping on a binary heap.

◎ AFTER THIS CHAPTER YOU CAN

- Write the n-step return and state the error-reduction property that justifies intermediate n
- Define the λ-return and prove the forward and backward views are equivalent offline
- Implement eligibility traces and explain what the trace vector physically represents
- Build Dyna-Q and explain why planning steps buy sample efficiency at the cost of model bias
- Explain how Dyna-Q+ detects a world that has changed underneath a stale model
- Describe MCTS as decision-time planning, and name the two substitutions AlphaZero makes

7.1 The space between one step and all of them

Rusty delivers a package and receives +25. Which of the forty decisions that got him there deserves credit?

TD(0) credits exactly one: the state immediately before the reward. The state two steps back learns nothing this episode — it must wait until its successor's value improves, then inherit a little. Monte Carlo credits all forty at once, but every one of them absorbs the noise of the entire trajectory.

The ***n*-step return** interpolates by looking ahead n steps before bootstrapping:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}).$$

At $n = 1$ this is the TD target; at $n = \infty$ (or the episode end) it is the Monte Carlo return. The update is the usual one, $V(S_t) \leftarrow V(S_t) + \alpha[G_{t:t+n} - V(S_t)]$.

Intermediate n is not a compromise — it is usually strictly better than either extreme, and there is a theorem saying why.

Theorem 7.1 — Error-reduction property of n-step returns. For any value function V,

$$\max_s \left|\mathbb{E}_\pi[G_{t:t+n} \mid S_t = s] - v_\pi(s)\right| \;\leq\; \gamma^n \max_s \left|V(s) - v_\pi(s)\right|.$$

The expected n-step return is a better estimate of v_π than V itself, by a factor γ^n. Larger n contracts the bias faster — while adding the variance of n extra sampled rewards. That is the whole trade, and it has no universal optimum.

7.2 The λ-return: averaging all of them

Rather than choosing one n, average them all, with geometrically decaying weights:

$$G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}.$$

The factor $(1-\lambda)$ normalizes the weights to sum to one. At $\lambda = 0$ only the $n = 1$ term survives — TD(0). At $\lambda = 1$ all the weight slides onto the full return — Monte Carlo. In between, a smooth blend.

> **INTERACTIVE · ch07-lambda-dial**
>
> **The λ dial: from TD(0) to Monte Carlo**
>
> Two views of λ: the weights $(1-\lambda)\lambda^{n-1}$ placed on each n-step return, and the eligibility trace decaying along a corridor Rusty has just driven.
>
> Run this simulation in the web edition: `/chapters/traces-planning-mcts #ch07-lambda-dial`

Drag the dial and watch both panels. On the left, the weight distribution over n-step returns; on the right, the eligibility trace decaying backwards along the corridor Rusty just drove. Those are two views of the same number.

7.3 Forward and backward views

The definition above is the **forward view**: to update $V(S_t)$ you look ahead over future returns. It is conceptually clean and computationally useless online — you would have to wait for the episode to finish.

The **backward view** achieves the same thing causally. Maintain an **eligibility trace** $E_t(s)$ for every state, decaying by $\gamma\lambda$ each step and incremented when the state is visited:

$$E_t(s) = \gamma\lambda\, E_{t-1}(s) + \mathbb{1}[S_t = s].$$

Then at every step, compute the one-step TD error δ_t and apply it to **every** state in proportion to its trace:

$$V(s) \leftarrow V(s) + \alpha\, \delta_t\, E_t(s) \quad \text{for all } s.$$

The trace is a short-term memory of *who is responsible* for what happens now. A cell visited recently has a large trace and receives most of the credit; a cell visited long ago has a small one.

Theorem 7.2 — Forward–backward equivalence (offline). If updates are accumulated and applied at the end of an episode, the total update made by the backward view with accumulating traces equals the total update prescribed by the forward view using λ-returns.

PROOF

Unroll the trace at time t for a state s first visited at time $k \le t$: $E_t(s) = (\gamma\lambda)^{t-k}$. The total backward update to s over the episode is therefore

$$\alpha \sum_{t \ge k} (\gamma\lambda)^{t-k}\, \delta_t .$$

Now take the forward view. Chapter 6's exercise established that the MC error decomposes as a discounted sum of TD errors; the same telescoping applied to the λ-return gives

$$G_k^\lambda - V(S_k) = \sum_{t \ge k} (\gamma\lambda)^{t-k} \delta_t .$$

The two expressions are identical, so the accumulated updates agree. $$

The equivalence is exact only offline, because online updates change V mid-episode and the two views then diverge slightly. *True online* TD(λ) repairs this at modest extra cost. ■

WHY TRACES MATTER FOR ROBOTS SPECIFICALLY

Robot tasks have long horizons and sparse, delayed rewards. A quadruped that falls at step 800 did something wrong around step 780, not at step 799. One-step methods must propagate that information backwards one step per update — hundreds of updates to assign credit correctly.
Traces do it in one sweep, at the cost of one vector the size of the state (or the parameters). This is why $\lambda \approx 0.9$–0.97 is nearly universal in practice, and why the GAE parameter in Chapter 10's PPO — which is precisely this idea applied to advantages — is one of the few hyperparameters practitioners rarely change.

7.4 Dyna: planning with a learned model

Now the second unification. Chapter 5 planned with a *given* model; Chapter 6 learned with *no* model. Dyna does both: learn a model from real experience, then use it to generate simulated experience and learn from that too.

The architecture is small enough to state completely. After each real transition (S, A, R, S'):

1. **Direct RL** — apply a Q-learning update from the real transition.
2. **Model learning** — store Model$(S, A) \leftarrow (R, S')$.
3. **Planning** — repeat n times: sample a previously-seen (S, A), look up the model's prediction, and apply the *same* Q-learning update to that imagined transition.

The planning updates are indistinguishable from real ones as far as the learner is concerned. That is the insight in the epigraph: **planning and learning differ only in where the experience comes from.**

The payoff is sample efficiency, which for a robot is the currency that matters. With $n = 50$ planning steps per real step, Dyna-Q typically reaches good policies in an order of magnitude fewer *real* interactions than Q-learning — because each expensive real transition is mined fifty times.

> **⚠ THE MODEL IS WRONG, AND THE WORLD MOVES**
>
> Dyna trusts its model completely. When the model is wrong, the agent confidently learns nonsense — this is **model bias**, and Chapter 12 shows it compounding with the planning horizon.
>
> Worse is a model that *was* right. Suppose the warehouse is re-shelved overnight: a corridor that used to be open is now blocked. Dyna-Q keeps planning through the remembered corridor and may never re-explore it, because its policy sees no reason to go there.
>
> **Dyna-Q+** fixes this by adding an exploration bonus $\kappa\sqrt{\tau}$ to simulated rewards, where τ counts the steps since that state–action pair was tried *for real*. Long-untested transitions become attractive again, so the agent periodically re-verifies its beliefs. The idea — curiosity about the staleness of your own knowledge — recurs in modern exploration research.

Prioritized sweeping improves on uniform sampling of imagined transitions. Instead of picking (S, A) at random, maintain a priority queue ordered by the magnitude of the change a backup would produce, and always work on the largest. When a state's value changes, all its predecessors are enqueued with priority proportional to the change. Computation flows to where it matters, and the speedup over uniform Dyna is often an order of magnitude.

RUST `rl-tabular/src/dyna.rs`

```rust
use std::collections::HashMap;
use rl_core::Transition;
use crate::td::QLearning;

pub struct DynaQPlus {
    learner: QLearning,
    model: HashMap<(usize, usize), (f64, usize)>,
    /// Steps since each (s, a) was tried for real -- drives the bonus.
    last_tried: HashMap<(usize, usize), u64>,
    planning_steps: usize,
    kappa: f64,
    step: u64,
}

impl DynaQPlus {
    pub fn observe<R: rand::Rng>(&mut self, s: usize, a: usize, t: &Transition<usize>,
   rng: &mut R) {
        self.step += 1;

        // 1. Direct RL from real experience.
        self.learner.update(s, a, t);

        // 2. Model learning.
        self.model.insert((s, a), (t.reward, t.next_state));
        self.last_tried.insert((s, a), self.step);

        // 3. Planning -- the same update, on remembered transitions.
        let keys: Vec<_> = self.model.keys().copied().collect();
        for _ in 0..self.planning_steps {
            let &(ps, pa) = &keys[rng.gen_range(0..keys.len())];
            let (reward, next) = self.model[&(ps, pa)];

            // Dyna-Q+ bonus: reward staleness, so the agent re-checks old beliefs.
            let tau = (self.step - self.last_tried[&(ps, pa)]) as f64;
            let imagined = Transition {
                next_state: next,
                reward: reward + self.kappa * tau.sqrt(),
                done: false,
            };
            self.learner.update(ps, pa, &imagined);
        }
    }
}
```

Dyna-Q+ composed from Chapter 6's QLearning. The planning loop calls exactly the same update as real experience — that identity is the entire point of the architecture.

7.5 Decision-time planning and MCTS

Everything so far uses planning to improve a *stored* policy — Sutton and Barto call this **background planning**. There is another use: plan at the moment of decision, for the state you are actually in, then throw the computation away.

Monte Carlo tree search is the canonical method. From the current state, repeat four steps:

1. **Selection** — descend the tree using a rule that balances exploitation and

exploration. The standard is UCT, which is Chapter 3's UCB applied at each node: $\arg\max_a \left[Q(s,a) + c\sqrt{\ln N(s)/N(s,a)} \right]$.

2. **Expansion** — add a child node for an untried action.
3. **Simulation** — roll out to a terminal state with a cheap default policy.
4. **Backpropagation** — propagate the outcome up the path, updating N and Q.

Given enough iterations, UCT converges to the optimal action at the root. The tree grows asymmetrically, concentrating depth along promising lines — which is exactly how you would want a finite computation budget spent.

> **WHAT ALPHAZERO CHANGED**
>
> AlphaZero keeps the MCTS skeleton and makes two substitutions. First, the random rollout is replaced by a **learned value network** evaluating the leaf directly — no simulation to the end at all. Second, the UCT exploration term is replaced by **PUCT**, which weights exploration by a learned policy network's prior, so the search spends its budget on moves that look plausible.
>
> The result is a loop in which search improves the policy and the improved policy makes search better — generalized policy iteration again, with search as the improvement operator. Chapter 12 builds the robotics version, where the "search" is trajectory optimization through a learned dynamics model and the horizon is a few hundred milliseconds rather than a whole game.

For robots, decision-time planning is most familiar as **model predictive control**: at each control step, optimize a short action sequence against a model, execute the first action, discard the rest, and re-plan. Chapter 12 derives it properly and Chapter 13 gives it its classical-control context.

7.6 Chapter bridge

Part I is complete, and it is worth naming what has been assembled.

We have a formalism (Chapter 4), an exact planning method for when the model is known (Chapter 5), sampling methods for when it is not (Chapter 6), a continuous dial between one-step and full-return credit assignment (this chapter), and an architecture that learns a model and plans with it while continuing to learn (also this chapter). All of it is unified by generalized policy iteration and underwritten by the contraction mathematics of Chapter 2.

All of it also assumes **tables**. Every method here stores one number per state or per state–action pair, and Chapter 5's arithmetic already showed that a modest robot arm has more states than a sweep can touch before the sun burns out.

Part II abandons tables. We will represent value functions and policies as

parameterized functions — linear in features, then deep networks — and discover that the guarantees do not survive the transition intact. Chapter 8 begins with what breaks, because knowing precisely how approximation can diverge is what makes the engineering of Chapters 9 through 12 comprehensible rather than superstitious.

Exercises

1 FOUNDATION • • • **Prove the error-reduction property**

Prove Theorem 7.1. Then explain why it does NOT imply that larger n is always better in practice — what does the theorem measure, and what does it ignore?

2 FOUNDATION • **λ-weights sum to one**

Verify that $(1-\lambda)\Sigma\_\{n\geq 1\}\ \lambda\^\{n-1\} = 1$, and work out what the weights become when an episode terminates at step T. Why must the remaining weight go onto the full return?

3 FOUNDATION • • • **The equivalence, in full**

Reproduce the proof of Theorem 7.2, justifying the telescoping step explicitly. Then construct a small example where online updating makes the forward and backward views differ, and quantify the discrepancy.

4 FOUNDATION • • **Effective lookahead**

Show that the mean n under the λ-weighting is $1/(1-\lambda)$, and interpret this as an effective lookahead horizon. Compare with the effective horizon $1/(1-\gamma)$ from Chapter 4 — what does the product $\gamma\lambda$ mean?

5 CONCEPTUAL • **The two extremes**

In the λ dial, set $\lambda = 0$ and $\lambda = 1$ and describe precisely what each does to the trace along the corridor. Then find the λ at which the trace half-life is about five steps, at $\gamma = 0.95$.

6 PRACTICAL • • **Sarsa(λ) with traces**

Implement Sarsa(λ) with accumulating traces and sweep $\lambda \in \{0, 0.5, 0.9, 0.95, 0.99\}$ on Rusty's warehouse. Plot episodes-to-threshold against λ. The curve should be U-shaped — explain both arms.

7 PRACTICAL • • • **The blocking maze**

Build a maze whose optimal corridor is blocked halfway through training. Run Dyna-Q and Dyna-Q+ across the change. Measure how many episodes each needs to recover, and tune κ to trade recovery speed against steady-state performance.

8 PRACTICAL • • • **Prioritized sweeping**

Implement prioritized sweeping with a binary heap and compare total backups to convergence against uniform Dyna-Q at equal planning budgets. Report the speedup, and identify what problem structure makes the advantage largest.

7.7 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapter 7 (n-step bootstrapping), Chapter 12 (eligibility traces, forward and backward views, true online TD(λ)), and Chapter 8 (Dyna, prioritized sweeping, MCTS).

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§6 mental rehearsal — the robotics case for planning with learned models, developed fully in Chapter 12.

FURTHER READING AND MODERN SOURCES

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming *ICML 1990*.
The Dyna architecture, in the original.

Moore, A. W. & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time *Machine Learning 13*.
The priority-queue idea of §7.4.

van Seijen, H. & Sutton, R. S. (2014). True Online TD(λ) *ICML 2014*.
Repairs the online gap in the forward–backward equivalence exactly, rather than approximately.

Kocsis, L. & Szepesvári, C. (2006). Bandit based Monte-Carlo Planning *ECML 2006*.
UCT — Chapter 3's UCB applied inside a search tree, with the consistency proof.

Silver, D. et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play *Science 362(6419)*.
AlphaZero — the two substitutions described in §7.5, and search as a policy improvement operator.

Part II

Scaling Up: Function Approximation and Deep RL

CHAPTER 8

Function Approximation & the Deadly Triad

> *In robotics, the state and action spaces are continuous and high-dimensional. Tabular representations are hopeless; the question is not whether to approximate but how.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On why robot RL cannot use tables
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Part I built a complete theory that assumes one number per state. Chapter 5's arithmetic already showed a modest robot arm has more states than a sweep can touch before the sun burns out. So we replace the table with a parameterized function and generalize across states — and discover that the guarantees do not survive intact. This chapter is about what breaks: the objective becomes a projection rather than an equality, the update stops being a gradient of anything, and three individually-harmless ingredients combine into a mechanism that makes learning diverge outright.

FOUNDATION

The VE objective, semi-gradient TD and the dropped term, the linear TD fixed point, tile coding, and the deadly-triad divergence analysis.

CONCEPTUAL

Baird's counterexample running live, with each triad ingredient switchable — removing any one restores stability.

PRACTICAL

Tile coding in pure ndarray and the first burn network: linear methods on Pendle, then

a nonlinear value function.

◎ AFTER THIS CHAPTER YOU CAN

- State the value-error objective and explain why the on-policy distribution appears in it
- Derive the semi-gradient TD update and explain precisely which term is dropped and why
- Construct tile-coded features and reason about generalization versus resolution
- Name the three members of the deadly triad and explain the divergence mechanism
- Explain why Baird's counterexample diverges even though the true value function is representable
- Choose sensibly between on-policy stability and off-policy sample efficiency for a robot problem

8.1 The end of tables

Pendle's state is $(\theta, \dot{\theta})$ — two real numbers. There are uncountably many of them. A table has no entry for $\theta = 0.7231847\ldots$, and it never will.

Discretizing does not rescue us. Chapter 5's widget made the arithmetic concrete: even a coarse 10-bin discretization of a 7-degree-of-freedom arm produces more states than could be swept in the lifetime of the universe. And a discretization fine enough to control well is far worse than coarse.

The deeper problem is that tables **cannot generalize**. Rusty learning that a cell three metres from a shelf is safe tells a table nothing about the cell one centimetre away. Every state must be visited to be learned. For a robot that is not a slow path to success; it is no path at all.

So we replace the table with a parameterized function

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s), \qquad \mathbf{w} \in \mathbb{R}^d, \quad d \ll |\mathcal{S}|,$$

and accept the consequence: **changing one weight changes the value of many states at once.** That is the entire point — it is how generalization happens — and it is also the source of every difficulty in this chapter.

8.2 What are we even optimizing?

With a table, we could hit the Bellman equation exactly. With $d \ll |\mathcal{S}|$ parameters we generally cannot, so we must decide which errors matter.

The standard choice is the **mean squared value error**, weighted by how often each state is actually visited:

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_s \mu(s)\,[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2\,,$$

where μ is the on-policy state distribution.

> **WHY THE DISTRIBUTION BELONGS IN THE OBJECTIVE**
>
> Accuracy in states the robot never visits is worthless. If Ferris spends his life trotting on flat ground, the value function's error while inverted mid-backflip does not matter.
> That weighting is not a convenience — it turns out to be the technical condition that makes on-policy learning stable. Learn under a different distribution than μ, and §8.5 shows the whole thing can come apart. The distribution is load-bearing.

8.3 Semi-gradient TD, and the term we drop

Gradient descent on $\overline{\mathrm{VE}}$ would use targets $v_\pi(s)$ — which we do not have. Substituting the TD target gives the update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[R_{t+1} + \gamma\,\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})]\,\nabla\hat{v}(S_t,\mathbf{w}).$$

Look carefully at what happened. The target $R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$ **depends on** $\mathbf{w}$, yet we did not differentiate through it. That is deliberate, and it is why this is called a **semi-gradient** method: it is the gradient of the error with respect to the prediction only, treating the target as a constant.

> **THIS IS NOT THE GRADIENT OF ANYTHING**
>
> Semi-gradient TD is not performing gradient descent on $\overline{\mathrm{VE}}$, or on any other objective function. There is no scalar $J(\mathbf{w})$ whose gradient this is.
> That matters because every reassuring intuition about gradient descent — that it goes downhill, that it converges to a local minimum, that a small enough step size makes it safe — is unavailable. We are iterating a fixed-point operator and hoping it contracts, which §8.5 shows it may not.
> Why not take the true gradient? Because differentiating through the target gives the **residual gradient** method, which requires two independent samples of the next state from the same state — a *double sampling* problem. In a simulator you can reset and re-sample; on a robot you cannot rewind the world. Semi-gradient is what is actually available.

For the **linear** case $\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$ the gradient is just the feature vector, and there is real theory. On-policy linear semi-gradient TD(0) converges to the **TD fixed point** $\mathbf{w}_{\text{TD}}$, which satisfies

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \le \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}).$$

So TD's answer is within a factor $1/(1-\gamma)$ of the best the function class can do. At $\gamma = 0.99$ that factor is 100 — a weak guarantee, but a guarantee, and it is more than Part II will offer once networks arrive.

8.4 Features: tile coding and what came after

Linear methods are only as good as their features. The classic robotics choice is **tile coding**: overlay several offset grids ("tilings") on the state space; a state activates exactly one tile per tiling, giving a sparse binary feature vector.

Tile coding gets three things right at once. Generalization is controlled by tile *width* while resolution is controlled by the *number of tilings* — you tune them independently. Lookup is $O(\text{tilings})$ regardless of dimension. And with exactly n active features, the effective learning rate is α/n, so step sizes behave predictably.

RUST `rl-deep/src/features/tile.rs`

```rust
/// Tile coding over a bounded continuous state space.
/// Each tiling is offset so that a state activates a different tile in each,
/// giving resolution finer than any single tiling's width.
pub struct TileCoder {
    n_tilings: usize,
    tiles_per_dim: usize,
    bounds: Vec<(f64, f64)>,
    offsets: Vec<Vec<f64>>,
}

impl TileCoder {
    /// Indices of the active features -- exactly `n_tilings` of them.
    pub fn active(&self, state: &[f64]) -> Vec<usize> {
        let mut idx = Vec::with_capacity(self.n_tilings);

        for (t, offset) in self.offsets.iter().enumerate() {
            let mut coords = Vec::with_capacity(state.len());

            for (d, &x) in state.iter().enumerate() {
                let (lo, hi) = self.bounds[d];
                let scaled = (x - lo) / (hi - lo) * self.tiles_per_dim as f64;
                let c = (scaled + offset[d]).floor() as isize;
                coords.push(c.clamp(0, self.tiles_per_dim as isize - 1) as usize);
            }

```

```
            // Flatten (tiling, coords) into one feature index.
            let mut flat = t;
            for c in coords {
                flat = flat * self.tiles_per_dim + c;
            }
            idx.push(flat);
        }
        idx
    }
}

/// Semi-gradient SARSA with binary features: the update touches only the
/// active tiles, so it costs O(n_tilings) regardless of the table's size.
pub fn sarsa_update(w: &mut [f64], active: &[usize], delta: f64, alpha: f64) {
    let step = alpha / active.len() as f64;
    for &i in active {
        w[i] += step * delta;
    }
}
```

Tile coding with hashing, in pure ndarray. This is the workhorse that solved cart-pole and mountain-car for two decades before deep networks — and it still trains in milliseconds where a network takes minutes.

Neural networks replace hand-designed features with learned ones. The trade is stark: networks handle raw pixels and high dimensions where tile coding cannot, at the cost of every theoretical guarantee in this section and orders of magnitude more compute. For a 2-D state like Pendle's, **tile coding is still the better engineering choice** — a fact worth knowing before reaching for a GPU.

8.5 The deadly triad

Now the central result of the chapter, and the reason Part II is engineered the way it is.

Three ingredients, each individually fine:

1. **Function approximation** — required, since tables do not scale.
2. **Bootstrapping** — targets built from your own estimates, as in TD. Gives efficiency and low variance.
3. **Off-policy training** — learning about one policy from data generated by another. Essential for sample efficiency, which robots cannot do without.

Any two are safe. All three together can make the parameters diverge to infinity.

> **INTERACTIVE · ch08-deadly-triad**
>
> **The deadly triad, one ingredient at a time**
>
> Baird's counterexample running live. All rewards are zero and the true value function is exactly representable, yet the parameters diverge. Switch off function approximation,

bootstrapping or off-policy training and stability returns.

Run this simulation in the web edition: `/chapters/function-approximation #ch08-deadly-triad`

Baird's counterexample is the cleanest demonstration. Every reward is zero, so the true value function is $v = 0$ everywhere — and it is **exactly representable** by the feature set. There is nothing to approximate badly. Yet the weights grow without bound.

⚠ THE MECHANISM, STATED PLAINLY

Under off-policy training, the states you *update* are distributed differently from the states the target policy *visits*. Semi-gradient updates at state s change the value of s' through shared features, but nothing corrects the resulting error at s' — because the update distribution does not weight s' appropriately.

Bootstrapping then feeds that uncorrected error back into the next target, which inflates the next update. The loop has positive gain and no stabilizing term. Since the update is not the gradient of any objective, there is no potential function decreasing along the way to stop it.

Switch off any single ingredient in the widget and the loop breaks. Without bootstrapping, targets are real returns and carry no inherited error. Without off-policy, the update distribution matches the target policy's and the errors are corrected where they occur. Without approximation, changing one state's value cannot corrupt another's.

Three families of response exist, and Part II uses all of them:

Live with it, carefully. DQN (Chapter 9) is squarely in the triad and works anyway — because target networks slow the feedback loop and replay buffers keep the update distribution broad. It is engineering, not theory, and it is honest to say so.

Stay on-policy. PPO (Chapter 10) avoids the third ingredient entirely, which is a large part of why it is the workhorse of real-world robot RL despite being less sample-efficient. Tang's survey finds on-policy methods dominating the successful sim-to-real results, and this is the reason.

Fix the objective. Gradient-TD methods (GTD2, TDC) perform true stochastic gradient descent on a *projected* Bellman error, and converge under all three conditions. They are theoretically satisfying, somewhat more complex, and less used in practice than they deserve.

8.6 Control with approximation

For control, parameterize $\hat{q}(s, a, \mathbf{w})$ and run the same GPI loop from Chapter 5, with semi-gradient SARSA updates and ε-greedy action selection.

Two new difficulties appear immediately. **The policy is no longer stable:** a small weight change can flip the argmax at many states at once, so the policy can oscillate even when the values are nearly converged. And **the state distribution shifts as the policy changes**, so the μ that made $\overline{\text{VE}}$ meaningful is a moving target — a control problem is a non-stationary prediction problem wearing a disguise.

Despite this, semi-gradient SARSA with tile coding is remarkably robust in practice and remains a strong baseline. Before assuming a robot problem needs deep RL, it is worth trying: it trains in seconds, has few hyperparameters, and when it works you are done.

8.7 Chapter bridge

We gave up tables and got generalization. We also gave up exact convergence guarantees, learned that our update is not a gradient of anything, and met a mechanism that can make training diverge on a problem with zero rewards and a representable answer.

Chapter 9 takes the triad on anyway. Deep Q-networks put a neural network in the value function, train off-policy from a replay buffer, and bootstrap — all three ingredients at maximum strength. It works, and understanding *why* it works is understanding the two pieces of engineering that make it work: replay buffers that decorrelate the update distribution, and target networks that slow the feedback loop enough that the loop's gain drops below one. Both are direct responses to what this chapter diagnosed.

Exercises

1 FOUNDATION •• **The dropped term**

Write out the true gradient of the squared TD error, including the term that differentiates through the target. Show that estimating it from a single sample gives a biased estimate, and explain why two independent successor samples would be needed.

2 FOUNDATION ••• **The TD fixed point bound**

State the linear TD fixed-point bound precisely and evaluate the factor $1/(1-\gamma)$ at $\gamma = 0.9, 0.99, 0.999$. At what point does the guarantee stop being informative for a robot task?

3 FOUNDATION • • • **Why Baird diverges**

Analyze Baird's counterexample directly: write the expected update as a linear map on w, and show that its iteration matrix has an eigenvalue with magnitude greater than one. Then verify that restricting to the on-policy distribution makes all eigenvalues sub-unit.

4 FOUNDATION • • **Tile coding arithmetic**

For a 2-D state with 8 tilings of 8×8 tiles each, compute the total feature count, the number active at any state, and the effective per-feature learning rate for a nominal α. Then redo it for a 6-D state and say what breaks.

5 CONCEPTUAL • **Break the triad three ways**

In the deadly-triad widget, disable each ingredient in turn and record the final parameter norm. Confirm that removing ANY one stabilizes learning, then explain each stabilization in one sentence.

6 CONCEPTUAL • • **Does a smaller step size save you?**

With all three ingredients active, reduce α as far as the slider allows. Does divergence stop, or merely slow? Explain what this tells you about the difference between instability and slow convergence.

7 PRACTICAL • • **Tile-coded cart-pole**

Implement semi-gradient SARSA with tile coding on Pendle's swing-up. Sweep the number of tilings and tile width, and plot the resolution/generalization frontier. Note the wall-clock time — it should embarrass a neural network.

8 PRACTICAL • • • **Linear versus nonlinear**

Replace the tile-coded value function with a small burn MLP on the same task, holding everything else fixed. Compare sample efficiency, wall-clock time, and stability across five seeds. Report honestly which you would ship.

8.8 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapters 9–11: on-policy prediction with approximation, on-policy control, and off-policy methods with approximation — where the deadly triad is named and analyzed.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§4.2 value-function approximation in robotics, and §2.4 on why representation choice dominates algorithm choice.

FURTHER READING AND MODERN SOURCES

Baird, L. (1995). Residual Algorithms: Reinforcement Learning with Function Approximation *ICML 1995*.
The counterexample, and the residual-gradient alternative that avoids it at the cost of double sampling.

Tsitsiklis, J. N. & Van Roy, B. (1997). An Analysis of Temporal-Difference Learning with Function Approximation *IEEE Transactions on Automatic Control 42(5)*.
The convergence proof for on-policy linear TD, and the $1/(1-\gamma)$ error bound quoted in §8.3.

Sutton, R. S. (1996). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding *NeurIPS 8*.
Tile coding as it is actually used.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C. & Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation *ICML 2009*.
GTD2 and TDC — true gradient methods that converge under all three triad conditions.

van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N. & Modayil, J. (2018). Deep Reinforcement Learning and the Deadly Triad *arXiv:1812.02648*.
An empirical study of when the triad actually bites in deep RL — essential reading before Chapter 9.

CHAPTER 9

Deep Value-Based Methods: DQN & Descendants

> *We demonstrate that a single architecture can successfully learn control policies in a range of different environments with only very minimal prior knowledge, receiving only the pixels and the game score as inputs.*
>
> **Volodymyr Mnih and colleagues** · DeepMind
> *Human-level control through deep reinforcement learning, Nature 2015*

Chapter 8 diagnosed the deadly triad and showed it can make learning diverge. Deep Q-networks charge straight into all three ingredients — a neural network, off-policy data, and bootstrapped targets — and work anyway. This chapter is about the two pieces of engineering that make that possible, why each is a direct response to a specific failure mode, and how the family that grew from DQN systematically removed its remaining flaws. It closes on where value-based methods belong in robotics, which is narrower than their fame suggests.

FOUNDATION

The DQN objective, replay and target networks analyzed as variance and stability surgery, Double DQN, dueling decomposition, prioritized replay with bias correction, and the distributional Bellman operator.

CONCEPTUAL

Replay and target networks switchable live, each producing its own characteristic failure signature in the training curve.

PRACTICAL

A full DQN in burn with a sum-tree prioritized replay buffer, trained on visual-

gridworld Rusty.

◎ AFTER THIS CHAPTER YOU CAN

- Explain experience replay as a fix for correlated updates, and target networks as a fix for the moving-target problem
- Derive the Double DQN target and connect it to Chapter 6's maximization bias
- Explain the dueling architecture's decomposition and when it helps
- Implement prioritized replay and state why importance-sampling weights are required
- State the distributional Bellman operator and explain what C51 learns that DQN does not
- Say precisely where value-based methods fit in robotics, and where they do not

9.1 From Q-table to Q-network

The move is small to write and large in consequence. Replace $Q(s, a)$ — a table — with $Q(s, a; \theta)$, a neural network, and minimize the squared TD error:

$$L(\theta) = \mathbb{E}\left[\left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta) - Q(S_t, A_t; \theta)\right)^2\right].$$

Then run semi-gradient descent, exactly as in Chapter 8: differentiate the prediction, not the target.

Done naively, this fails reliably. It has all three triad ingredients and two additional problems that Chapter 8's tabular-adjacent analysis did not surface.

Correlated samples. Gradient methods assume roughly independent samples. Consecutive transitions from a robot are anything but: at 100 Hz, successive states differ by a centimetre. A batch of 32 consecutive transitions carries perhaps two transitions' worth of information, and the gradient is systematically biased toward whatever the robot happens to be doing right now.

A target that runs away. The target $R + \gamma \max_{a'} Q(S', a'; \theta)$ depends on θ — the very parameters being updated. Every gradient step moves the target. It is regression where the labels change each time you fit them, and the resulting dynamics oscillate or diverge.

9.2 The two pieces of surgery

Experience replay. Store transitions (s, a, r, s') in a large circular buffer and train on uniformly sampled minibatches. Three benefits follow at once: samples within a batch are decorrelated, each expensive transition is reused many times (sample efficiency, which robots care about above all else), and the update distribution is smoothed across the whole recent history rather than concentrated wherever the policy currently is — directly attacking the distribution mismatch that Chapter 8 identified as the triad's fuel.

Target networks. Keep a second, frozen copy θ^- of the parameters, used only to compute targets, and sync it every C steps:

$$L(\theta) = \mathbb{E}\left[\left(R + \gamma \max_{a'} Q(S', a'; \theta^-) - Q(S, A; \theta)\right)^2\right].$$

Between syncs the target is *constant*, so each interval is honest supervised regression toward a fixed objective. This is fitted Q-iteration in disguise, and it is where the stability comes from: the feedback loop that Chapter 8 diagnosed still exists, but its gain is now throttled by C.

> INTERACTIVE · `ch09-replay-target`
>
> **Replay and target networks as variance surgery**
>
> Disable each and watch its characteristic failure appear: correlated batches add noise, a moving target adds oscillation.
>
> Run this simulation in the web edition: `/chapters/deep-value-methods #ch09-replay-target`

Switch each off in turn and the failure signatures are distinct. Without replay the estimate rattles — correlated batches. Without a target network it oscillates and can run away — the moving target. That the two failures look different is useful diagnostic knowledge when your own training goes wrong.

> ENGINEERING, NOT THEORY
>
> Neither fix comes with a convergence proof. They do not remove any triad ingredient — DQN still approximates, bootstraps, and learns off-policy. What they do is reduce the gain of the divergence loop below one for problems people actually care about.
> It is worth being honest about this, because it sets expectations correctly for your own robot: DQN can still diverge, and when it does, the first two things to check are buffer size and target sync interval.

9.3 The family that fixed DQN's flaws

Double DQN. Chapter 6 proved that max over noisy estimates overestimates. DQN's target has exactly that form, so DQN systematically overestimates action values. The fix is Chapter 6's double estimator, and it is nearly free because a second network already exists:

$$Y^{\text{DoubleDQN}} = R + \gamma\, Q\big(S', \arg\max_{a'} Q(S', a'; \theta);\, \theta^{-}\big).$$

The online network *selects*; the target network *evaluates*. One line of code, and it measurably improves both value accuracy and final policy quality.

Dueling networks. Split the head into a state-value stream and an advantage stream:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right).$$

The subtraction makes the decomposition identifiable — without it, V and A could shift by any constant. The benefit is that $V(s)$ is learned from every transition regardless of which action was taken, which matters greatly in states where the action barely matters. For a robot, that is most states: while driving down an empty corridor, the choice of small steering correction is nearly irrelevant, but the corridor's value is not.

Prioritized replay. Sample transitions in proportion to their TD error, $P(i) \propto |\delta_i|^{\omega}$, so surprising transitions are revisited more. This is Chapter 7's prioritized sweeping applied to a replay buffer. Because it changes the sampling distribution, the estimate becomes biased, and the bias must be corrected with importance-sampling weights $w_i \propto (1/(N \cdot P(i)))^{\beta}$, with β annealed to 1 over training.

Rainbow combines Double, dueling, prioritized replay, multi-step returns, distributional values and noisy exploration nets. The ablation is the interesting part: multi-step returns and prioritized replay contribute most, and the components are largely complementary rather than redundant.

9.4 Distributional RL: learning the whole distribution

A genuinely different idea, and the most robotics-relevant of the extensions.

Instead of learning $\mathbb{E}[G_t]$, learn the **distribution** of returns $Z(s,a)$. It satisfies its own recursion — the distributional Bellman equation, where $\overset{D}{=}$ denotes equality in distribution:

$$Z(s,a) \overset{D}{=} R + \gamma Z(S', A').$$

C51 represents Z as a categorical distribution over 51 fixed atoms, applies the Bellman operator (which shifts and scales the support), projects back onto the fixed atoms, and minimizes cross-entropy.

WHY A ROBOT MIGHT WANT THE WHOLE DISTRIBUTION

Two policies with identical expected return can be wildly different in practice. One reaches the goal in 10 seconds every time; the other reaches it in 5 seconds usually and falls down the stairs occasionally. The mean cannot tell them apart. The distribution can.

That opens the door to **risk-sensitive control**: optimize a conditional value at risk rather than the mean, and the policy becomes conservative in precisely the situations where variance is dangerous. For a robot that can break itself or its surroundings, this is not a refinement — it is often the right objective, and Chapter 14 returns to it under constrained MDPs.

Distributional methods also just learn better, even when only the mean is used. The leading explanation is that predicting a richer target is a better auxiliary learning signal for the representation.

RUST `rl-deep/src/dqn.rs`

```rust
use burn::prelude::*;
use burn::optim::{GradientsParams, Optimizer};

pub struct Dqn<B: Backend> {
    online: QNetwork<B>,
    target: QNetwork<B>,
    gamma: f32,
    sync_every: usize,
    steps: usize,
}

impl<B: AutodiffBackend> Dqn<B> {
    pub fn train_step<O: Optimizer<QNetwork<B>, B>>(
        &mut self,
        batch: &Batch<B>,
        optim: &mut O,
        lr: f64,
    ) -> f32 {
        // Q(s, a) for the actions actually taken.
        let q_pred = self
            .online
            .forward(batch.states.clone())
            .gather(1, batch.actions.clone().unsqueeze_dim(1))
            .squeeze(1);

        // Double DQN: the ONLINE net picks the argmax, the TARGET net scores it.
        // This is Chapter 6's double estimator, reusing a network we already have.
        let next_actions = self
            .online
            .forward(batch.next_states.clone())
            .argmax(1);

        let next_q = self
            .target
            .forward(batch.next_states.clone())
            .gather(1, next_actions)
            .squeeze(1);
```

```

        // detach(): no gradient through the target -- the semi-gradient of Ch 8.
        let target = batch
            .rewards
            .clone()
            .add(next_q.mul_scalar(self.gamma).mul(batch.not_done.clone()))
            .detach();

        // Huber loss: bounded gradients keep one outlier transition from
        // wrecking the network, which matters when rewards are unnormalized.
        let loss = huber(q_pred - target, 1.0).mean();

        let grads = GradientsParams::from_grads(loss.backward(), &self.online);
        self.online = optim.step(lr, self.online.clone(), grads);

        self.steps += 1;
        if self.steps % self.sync_every == 0 {
            self.target = self.online.clone(); // freeze a fresh copy
        }
        loss.into_scalar().elem()
    }
}
```

A Double DQN training step in burn. Note detach() on the target — that is the semi-gradient of Chapter 8, expressed in code: gradients flow through the prediction only.

9.5 Where value-based methods belong in robotics

Honesty is due here, because DQN's fame does not match its robotics footprint.

The hard constraint is discrete actions. Computing $\max_{a'} Q(s', a')$ requires enumerating actions. A 7-DoF arm's torque vector is continuous; discretizing each joint into even 5 levels gives $5^7 = 78{,}125$ actions per step, and the max is now the bottleneck. This is why Chapter 11 exists.

Where value-based methods genuinely fit is **mid- and high-level discrete decisions**: which grasp to attempt from a set of candidates, which skill to invoke next, which waypoint to pursue. Tang's survey finds exactly this pattern — off-policy value methods appear in manipulation papers with discrete or discretized action spaces, while continuous control belongs to policy-gradient and actor–critic methods.

The grasp-selection case is the clearest, and Chapter 20 builds it: a network scores candidate grasps from a depth image, and the argmax over a few hundred candidates is both tractable and exactly the right computation.

9.6 Chapter bridge

We put a network in the value function and survived the deadly triad through two pieces of engineering — replay to decorrelate and broaden the update distribution, target networks to throttle the feedback loop — then watched a decade of refinements remove the remaining flaws one at a time.

But the max over actions is a wall. Robots command continuous torques and velocities, and no amount of discretization makes that comfortable.

Chapter 10 changes the object being learned. Instead of a value function from which a policy is extracted, parameterize the **policy directly** and ascend the gradient of expected return. The policy gradient theorem makes that possible, continuous actions become natural rather than awkward, and the algorithm that results — PPO — is the one behind more real-world robot RL successes than every method in this chapter combined.

Exercises

1 FOUNDATION •• **Replay as decorrelation**

Model consecutive states as an AR(1) process with correlation ρ. Compute the effective sample size of a batch of B consecutive transitions versus B uniformly sampled from a buffer of size N. At $\rho = 0.99$, how large must N be for the batch to be worth its nominal size?

2 FOUNDATION •• **Target networks as fitted Q-iteration**

Show that DQN with sync interval C is approximately fitted Q-iteration with C gradient steps per iteration. What does the contraction argument from Chapter 4 say about this, and where exactly does the argument break for a neural network?

3 FOUNDATION •• **Double DQN target**

Write both the DQN and Double DQN targets and prove that they coincide when the two networks are identical. Then explain why they diverge in exactly the direction that removes overestimation.

4 FOUNDATION •• **The dueling identifiability problem**

Show that $Q = V + A$ is unidentifiable without a constraint. Compare subtracting the mean advantage against subtracting the max, and say which gives more stable optimization and why.

5 CONCEPTUAL • **Two distinct failures**

Using the replay/target widget, produce and describe the characteristic curve for (a) no replay and (b) no target network. Write a two-sentence diagnostic guide you could use on your own training runs.

6 CONCEPTUAL •• **Sync interval sweet spot**

Sweep the target sync interval from 1 to 500. Both extremes are bad. Explain each failure mode and identify roughly where the interior optimum sits.

7 PRACTICAL ••• **Sum-tree prioritized replay**

Implement proportional prioritized replay with a sum-tree for O(log N) sampling and updates. Verify the importance-sampling weights correct the bias by comparing final value accuracy against uniform replay on a task with rare high-error transitions.

8 PRACTICAL ••• **Measure the overestimation**

Train DQN and Double DQN on visual-gridworld Rusty, logging max_a Q(s,a) against the true value computed by Chapter 5's value iteration on the underlying MDP. Plot both. The single-estimator version should sit visibly above the truth throughout training.

9.7 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapters 9–11 supply the approximation theory this chapter engineers around; §16.5 discusses DQN as a case study.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.

§3.3 policy-optimization axis — the empirical evidence for where off-policy value methods actually appear in successful robot systems.

FURTHER READING AND MODERN SOURCES

Mnih, V. et al. (2015). Human-level control through deep reinforcement learning *Nature 518*.

The DQN paper: replay, target networks, and the Atari results that started the deep RL era.

van Hasselt, H., Guez, A. & Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning *AAAI 2016*.

Chapter 6's double estimator, applied to DQN at almost no cost.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. & de Freitas, N. (2016). Dueling Network Architectures for Deep Reinforcement Learning *ICML 2016*.
The V/A decomposition and its identifiability constraint.

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2016). Prioritized Experience Replay *ICLR 2016*.
Prioritized sampling with importance-sampling correction, and the sum-tree implementation.

Hessel, M. et al. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning *AAAI 2018*.
The combination, and — more useful — the ablation showing which components actually carry the weight.

Bellemare, M. G., Dabney, W. & Munos, R. (2017). A Distributional Perspective on Reinforcement Learning *ICML 2017*.
C51 and the distributional Bellman operator, including its contraction in the Wasserstein metric.

CHAPTER 10

Policy Gradients: REINFORCE → PPO

> *Policy search methods are often the preferred approach in robotics, as they scale to high-dimensional continuous action spaces and allow the incorporation of task-appropriate prior structure into the policy.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On why robotics chose policy search
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Chapter 9 hit a wall: computing max over actions requires enumerating them, and robots command continuous torques. So stop learning a value function and extracting a policy from it — parameterize the policy directly and ascend the gradient of expected return. The policy gradient theorem makes that computable despite the fact that the distribution you are averaging over depends on the parameters you are differentiating. The result is PPO: the algorithm behind more real-world robot RL successes than every method in Chapter 9 combined.

FOUNDATION

The policy gradient theorem with full derivation, score-function estimators and their variance, baselines and advantage, GAE, the performance-difference lemma, TRPO's trust region, and PPO's clipped surrogate.

CONCEPTUAL

Three unbiased gradient estimators sampled hundreds of times, so the variance collapse from baselines is something you see rather than something you are told.

PRACTICAL

REINFORCE → A2C → PPO in burn with rayon-vectorized environments, solving

Pendle's swing-up with a Gaussian policy.

◎ AFTER THIS CHAPTER YOU CAN

- State three concrete reasons robotics prefers policy search over value-based methods
- Derive the policy gradient theorem line by line, including why the state-distribution term vanishes
- Explain why baselines reduce variance without introducing bias, and derive the advantage form
- Derive GAE and identify it as Chapter 7's λ-return applied to advantages
- Explain the trust-region argument and how PPO's clipped surrogate approximates it cheaply
- List the implementation details that actually decide whether PPO works, and why each matters

10.1 Why robotics chose policy search

Three reasons, all of which matter more on hardware than on benchmarks.

Continuous actions are natural. A policy that outputs the mean and standard deviation of a Gaussian over joint torques needs no argmax. Chapter 9's wall simply is not there.

Updates are smooth. A small parameter change produces a small change in behaviour. Value-based methods can flip the argmax at many states from one gradient step to the next, and a robot whose policy changes discontinuously mid-training is a robot that breaks something. This alone is worth a great deal of sample efficiency.

Prior structure fits naturally. You can constrain the policy class to whatever you know: a movement primitive with learnable weights (Chapter 17), a residual on top of a hand-tuned controller, an action space that respects joint limits by construction. Kober and colleagues make this argument at length, and it remains correct.

The cost is that policy gradient methods are usually **on-policy** — data must come from the current policy, so it is discarded after each update. That is expensive. It is also, per Chapter 8, exactly why they are stable: dropping the off-policy ingredient breaks the deadly triad.

10.2 The policy gradient theorem

Parameterize the policy as $\pi_\theta(a \mid s)$ and define the objective as expected return from the start-state distribution:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}\right].$$

We want $\nabla_\theta J$. The difficulty is immediate: J averages over trajectories whose distribution depends on θ. Differentiating an expectation whose measure moves is not a routine operation.

The **score-function identity** resolves it. For any distribution p_θ,

$$\nabla_\theta p_\theta(x) = p_\theta(x)\, \nabla_\theta \log p_\theta(x),$$

which is just the chain rule on log, rearranged. It converts a gradient of a probability into an expectation of a gradient — and expectations we can sample.

Theorem 10.1 — Policy gradient theorem. For the episodic objective,

$$\nabla_\theta J(\theta) \;\propto\; \sum_s \mu(s) \sum_a q_\pi(s,a)\, \nabla_\theta \pi_\theta(a \mid s) \;=\; \mathbb{E}_{\pi_\theta}\left[q_\pi(S_t, A_t)\, \nabla_\theta \log \pi_\theta(A_t \mid S_t)\right],$$

where μ is the on-policy state distribution.

PROOF

Start with the gradient of the state-value function and expand using $v_\pi(s) = \sum_a \pi_\theta(a \mid s) q_\pi(s,a)$ and the product rule:

$$\begin{aligned}\nabla v_\pi(s) &= \nabla\left[\sum_a \pi_\theta(a \mid s)\, q_\pi(s,a)\right] \\ &= \sum_a \Big[\nabla \pi_\theta(a \mid s)\, q_\pi(s,a) + \pi_\theta(a \mid s)\, \nabla q_\pi(s,a)\Big].\end{aligned}$$

Now expand $q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_\pi(s')]$. The reward and the dynamics do **not** depend on θ, so

$$\nabla q_\pi(s,a) = \gamma \sum_{s'} p(s' \mid s, a)\, \nabla v_\pi(s').$$

Substituting gives a recursion in ∇v_π:

$$\nabla v_\pi(s) = \sum_a \Big[\nabla \pi_\theta(a \mid s)\, q_\pi(s,a) + \gamma\, \pi_\theta(a \mid s) \sum_{s'} p(s' \mid s, a)\, \nabla v_\pi(s')\Big].$$

Unrolling this repeatedly, each step pushes the ∇v_π term one transition further into the future while accumulating a γ. Collecting terms by the number of steps k needed to reach each state gives

$$\nabla v_\pi(s_0) = \sum_s \left(\sum_{k=0}^{\infty} \gamma^k \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi_\theta(a \mid s)\, q_\pi(s, a),$$

and the bracketed quantity is precisely the (discounted) on-policy state distribution $\mu(s)$. Finally, apply the score-function identity $\nabla \pi_\theta = \pi_\theta \nabla \log \pi_\theta$ to turn the inner sum into an expectation over actions drawn from π_θ. $$ ■

WHAT THE THEOREM BUYS, AND WHY IT IS REMARKABLE

Look at what is **absent** from the final expression: the environment dynamics $p(s' \mid s, a)$. Nowhere does the gradient require knowing how the world responds. It requires only $\nabla \log \pi_\theta$ — the gradient of your own policy, which you wrote and can differentiate exactly.

That is why the theorem matters. The distribution over states shifts as θ changes, and you might expect a term accounting for that shift. There is none, because the derivation shows that shift is already absorbed into μ. You can estimate the gradient purely from sampled experience, with no model at all.

You have seen this before. Chapter 3's gradient bandit was this theorem on a one-state MDP, baseline and all.

REINFORCE turns the theorem into an algorithm: sample an episode, and for each step update

$$\theta \leftarrow \theta + \alpha\, \gamma^t\, G_t\, \nabla_\theta \log \pi_\theta(A_t \mid S_t).$$

It is unbiased and it is nearly unusable, because the variance of G_t is enormous — the return accumulates every random event in the trajectory, and multiplying that by the score function does not help.

10.3 Baselines and advantage

Chapter 3 already gave the fix. Subtract any function $b(s)$ that does not depend on the action:

$$\nabla_\theta J = \mathbb{E}\left[\big(q_\pi(S_t, A_t) - b(S_t)\big) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right].$$

WHY THE BASELINE IS FREE

The added term has expectation zero:

$$\mathbb{E}\left[b(S_t)\nabla \log \pi_\theta(A_t \mid S_t)\right] = \sum_s \mu(s)\, b(s) \sum_a \pi_\theta(a \mid s)\nabla \log \pi_\theta(a \mid s) = \sum_s \mu(s)\, b(s) \sum_a \nabla \pi_\theta(a \mid s).$$

Since $\sum_a \pi_\theta(a \mid s) = 1$ for every s, its gradient is zero, so the whole term vanishes. The estimator's mean is unchanged; only its variance moves. $$ ■

The best practical baseline is $v_\pi(s)$, which makes the multiplier the **advantage**:

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s).$$

The interpretation is exactly right. Advantage asks *"was this action better than what I typically do here?"* rather than *"was this outcome good?"* — and only the first question carries information about which action to prefer. A state where everything goes well produces large returns for every action, and without a baseline all of them get reinforced.

> **INTERACTIVE · ch10-gradient-variance**
>
> **Three unbiased estimators, three variances**
>
> The same policy gradient estimated hundreds of times under REINFORCE, REINFORCE with a baseline, and GAE. All share a mean; the spread differs by decades.
>
> Run this simulation in the web edition: `/chapters/policy-gradients #ch10-gradient-variance`

Turn up the reward offset. REINFORCE's histogram spreads badly even though a constant added to every return says nothing about which action was good — the score function multiplies it regardless. The baseline subtracts it straight back out. All three estimators have the same mean throughout; only the spread differs, and spread is what you pay for in samples.

10.4 Actor–critic and GAE

We do not know v_π, so learn it: a **critic** $V_\phi(s)$ trained by the TD methods of Chapter 6, and an **actor** π_θ updated by the policy gradient using the critic's advantage estimate. This is generalized policy iteration again — Chapter 5's pattern, with a parameterized policy and a learned value.

How should the advantage be estimated? The one-step form $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ has low variance and inherits the critic's bias. The Monte Carlo form $G_t - V(S_t)$ is unbiased and noisy. This is Chapter 7's dilemma verbatim, so it takes Chapter 7's answer.

Generalized advantage estimation is the λ-return applied to advantages:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}.$$

At $\lambda = 0$ it is the one-step TD error; at $\lambda = 1$ it is the Monte Carlo advantage. In practice $\lambda \approx 0.95$ works across an unusually wide range of tasks, which is why it is one of the few hyperparameters practitioners rarely touch.

10.5 Trust regions and PPO

One problem remains, and it is the one that decides whether training survives.

Policy gradient tells you a direction, not a step size. Take too large a step and the policy changes drastically; the data you collected is now from a policy that no longer exists, the next batch is collected under a worse policy, and performance collapses without recovering. Unlike supervised learning, **a bad update poisons all future data**.

The **performance-difference lemma** quantifies the risk: the improvement from π_{old} to π is exactly

$$J(\pi) - J(\pi_{\text{old}}) = \mathbb{E}_{s\sim d_\pi, a\sim\pi}\left[A_{\pi_{\text{old}}}(s,a)\right],$$

where the states are drawn from the **new** policy's distribution — which we cannot sample without running it. Approximating d_π by $d_{\pi_{\text{old}}}$ gives a surrogate objective that is accurate only while the policies remain close. TRPO enforces closeness with a hard KL constraint and solves the resulting problem with conjugate gradients — principled, and heavy.

PPO achieves nearly the same effect with a clipped objective. With the probability ratio $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\text{old}}}(A_t|S_t)}$:

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t,\ \ \mathrm{clip}\big(r_t(\theta),\,1-\epsilon,\,1+\epsilon\big)\hat{A}_t\right)\right].$$

WHAT THE CLIP ACTUALLY DOES

Read the two branches separately. When $\hat{A}_t > 0$ the action was better than average and we want to increase its probability — but the clip caps the objective at $r = 1 + \epsilon$, so once the probability has risen enough, further increase yields no additional reward. When $\hat{A}_t < 0$ the clip floors it at $1 - \epsilon$. The `min` is what makes this a pessimistic bound rather than merely a clamp: it takes the *worse* of the clipped and unclipped objectives, so the update never benefits from moving further than the trust region allows, in either direction.

The result is a first-order method — plain Adam, no conjugate gradients —

that empirically stays inside a trust region. That simplicity is why PPO, not TRPO, is what Tang's survey finds underneath most successful sim-to-real robot systems.

⚠ THE IMPLEMENTATION DETAILS ARE NOT DETAILS

Published ablations have repeatedly found that PPO's performance depends as much on its surrounding machinery as on the clipped objective itself. The ones that matter:
Advantage normalization per batch — without it the effective learning rate drifts with reward scale. **Observation normalization** with running statistics — a robot's joint angles and velocities differ by orders of magnitude. **Value-loss clipping** and a tuned value-loss coefficient. **Entropy bonus** to prevent premature collapse to a deterministic policy. **Gradient clipping** by global norm, typically 0.5. **Multiple epochs over each batch** — usually 3–10; more and the ratio drifts too far. **Approximate KL as a watchdog**: monitor it, and early-stop the epoch loop if it exceeds a threshold.
If your PPO does not work, the fault is more often in this list than in the algorithm.

RUST `rl-deep/src/ppo.rs`

```rust
use burn::prelude::*;

pub struct PpoConfig {
    pub clip_eps: f32,
    pub entropy_coef: f32,
    pub value_coef: f32,
    pub target_kl: f32,
}

pub struct PpoDiagnostics {
    pub policy_loss: f32,
    pub value_loss: f32,
    pub entropy: f32,
    pub approx_kl: f32,
    pub clip_fraction: f32,
}

impl<B: AutodiffBackend> Ppo<B> {
    pub fn surrogate(&self, batch: &RolloutBatch<B>, cfg: &PpoConfig)
        -> (Tensor<B, 1>, PpoDiagnostics)
    {
        let (mean, log_std) = self.actor.forward(batch.states.clone());
        let log_probs = gaussian_log_prob(&mean, &log_std, &batch.actions);

        // r_t(theta) = pi_theta(a|s) / pi_old(a|s), computed in log space for stability.
        let ratio = (log_probs.clone() - batch.old_log_probs.clone()).exp();

        // Advantage normalization -- not optional in practice.
```

```
        let adv = normalize(batch.advantages.clone());

        let unclipped = ratio.clone() * adv.clone();
        let clipped = ratio.clone().clamp(1.0 - cfg.clip_eps, 1.0 + cfg.clip_eps) * adv;

        // min() takes the pessimistic branch: the update never profits from
        // stepping outside the trust region in either direction.
        let policy_loss = -unclipped.min_pair(clipped).mean();

        let values = self.critic.forward(batch.states.clone()).squeeze(1);
        let value_loss = (values - batch.returns.clone()).powf_scalar(2.0).mean();

        let entropy = gaussian_entropy(&log_std).mean();

        let loss = policy_loss.clone()
            + value_loss.clone() * cfg.value_coef
            - entropy.clone() * cfg.entropy_coef;

        // Schulman's low-variance KL estimator; early-stop the epoch if it spikes.
        let log_ratio = log_probs - batch.old_log_probs.clone();
        let approx_kl = ((log_ratio.clone().exp() - 1.0) - log_ratio).mean();

        let clip_fraction = ratio
            .sub_scalar(1.0)
            .abs()
            .greater_elem(cfg.clip_eps)
            .float()
            .mean();

        (loss, PpoDiagnostics {
            policy_loss: policy_loss.into_scalar().elem(),
            value_loss: value_loss.into_scalar().elem(),
            entropy: entropy.into_scalar().elem(),
            approx_kl: approx_kl.into_scalar().elem(),
            clip_fraction: clip_fraction.into_scalar().elem(),
        })
    }
}
```

The PPO surrogate loss in burn, with the diagnostics that make training debuggable. Approximate KL and clip fraction are the two numbers to watch: KL spiking means the step was too large, clip fraction near zero means it was too small.

10.6 Chapter bridge

We changed what is learned. Instead of a value function with a policy extracted from it, the policy is the parameter vector, and the policy gradient theorem makes its gradient estimable from experience alone — with no model, and with the state-distribution shift already absorbed. Baselines and GAE tamed the variance; trust regions tamed the step size; PPO made the whole thing simple enough to run with Adam.

What we did not fix is sample efficiency. PPO is on-policy: every batch is used for a few epochs and thrown away. For a simulator with thousands of parallel environments that is acceptable, and Chapter 18 shows it is how locomotion is

actually trained. For a robot learning on hardware, discarding data is a luxury nobody can afford.

Chapter 11 returns to off-policy learning — this time with continuous actions and a replay buffer, accepting the deadly triad's risks in exchange for reusing every transition many times. Reacher gets his proper debut as a continuous-control task, and maximum-entropy RL turns exploration from a schedule you tune into a quantity the objective optimizes.

Exercises

1 FOUNDATION • • • **The theorem, unrolled**

Reproduce the proof of Theorem 10.1, being explicit about the unrolling step and how the discounted state distribution μ emerges. State exactly where the assumption that p and r do not depend on θ is used.

2 FOUNDATION • • **Baselines are free**

Prove that any action-independent baseline leaves the policy gradient unbiased. Then derive the variance-minimizing baseline and explain why v_π(s) is used instead despite not being optimal.

3 FOUNDATION • • **GAE telescopes**

Show that GAE with $\lambda = 1$ reduces to the Monte Carlo advantage and with $\lambda = 0$ to the one-step TD error. Then write GAE as an exponentially weighted average of n-step advantage estimators, mirroring Chapter 7's λ-return.

4 FOUNDATION • • **Reading the clip**

Sketch L^CLIP as a function of the ratio r, separately for positive and negative advantage. Mark where the gradient is zero. Then explain what the min() adds over simply clipping the ratio.

5 CONCEPTUAL • **Variance versus offset**

In the gradient lab, record the variance of all three estimators at reward offsets of 0, 10, 20, 40. REINFORCE should scale with the offset while the baseline versions do not. Explain the mechanism.

6 CONCEPTUAL • • **The GAE frontier**

Sweep GAE λ from 0 to 0.99 and record estimator variance. Then argue why the lowest-variance setting is not necessarily the best choice for learning.

7 PRACTICAL • • • **Build the ladder**

Implement REINFORCE, then add a baseline, then GAE, then clipping — four algorithms, each a small edit of the last. Run all four on Pendle's swing-up across five seeds and plot the learning curves together.

8 PRACTICAL • • • **Ablate the details**

Take working PPO and disable, one at a time: advantage normalization, observation normalization, gradient clipping, the entropy bonus. Quantify the damage from each. At least one should hurt more than you expect.

10.7 References

BASELINE REFERENCES

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapter 13: the policy gradient theorem, REINFORCE, baselines, and actor–critic methods.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§2.2.2 policy search, and §2.3 value-function approaches versus policy search — the argument reproduced in §10.1.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§5 — the finding that stable on-policy methods dominate the mature zero-shot sim-to-real results.

FURTHER READING AND MODERN SOURCES

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning *Machine Learning 8*.
REINFORCE, in the original.

Sutton, R. S., McAllester, D., Singh, S. & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation *NeurIPS 12*.
The policy gradient theorem, with the compatible-function-approximation condition.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. (2015). Trust Region Policy Optimization *ICML 2015*.
The performance-difference lemma, the monotonic improvement bound, and the KL-constrained update.

Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. (2016). High-Dimensional Continuous Control Using Generalized Advantage Estimation *ICLR 2016*.
GAE — Chapter 7's λ-return, applied to advantages.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal Policy Optimization Algorithms *arXiv:1707.06347*.
The clipped surrogate objective.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L. & Madry, A. (2020). Implementation Matters in Deep RL: A Case Study on PPO and TRPO *ICLR 2020*.
The evidence behind §10.5's warning: the surrounding machinery contributes as much as the objective.

CHAPTER 11

Off-Policy Continuous Control: DDPG, TD3 & SAC

> *Every real-world trial costs time, causes wear, and risks damage. Sample efficiency is not a convenience in robot learning — it is the constraint that shapes the entire problem.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On the curse of real-world samples
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

PPO throws its data away after a few epochs. In a simulator with thousands of parallel environments that is fine; on hardware it is unaffordable. This chapter returns to off-policy learning — replay buffers, continuous actions, every transition reused hundreds of times — and accepts the deadly triad's risks in exchange. We derive the deterministic policy gradient, diagnose the overestimation that made DDPG notoriously fragile, and arrive at maximum-entropy RL, where exploration stops being a schedule you tune and becomes a quantity the objective optimizes. Reacher gets his proper debut.

FOUNDATION

The DPG theorem, overestimation-bias analysis, TD3's clipped double-Q, the maximum-entropy objective, soft policy iteration convergence, and the reparameterization gradient with automatic temperature tuning.

CONCEPTUAL

The entropy temperature as a dial: watch a policy morph from a deterministic spike into a distribution that hedges across every action worth considering.

PRACTICAL

DDPG, TD3 and SAC in burn sharing Chapter 9's replay infrastructure, with Reacher built on rapier2d.

AFTER THIS CHAPTER YOU CAN

- Derive the deterministic policy gradient and explain why it eliminates the action integral
- Explain how overestimation bias accumulates through bootstrapping in continuous control
- State TD3's three fixes and say which failure mode each addresses
- Derive the soft Bellman operator and the maximum-entropy objective
- Apply the reparameterization trick and explain why it beats the score-function estimator here
- Choose between PPO and SAC for a given robot problem, with reasons

11.1 The sample-efficiency imperative

Chapter 10's PPO is stable, simple, and wasteful. Each batch is used for a handful of epochs and discarded, because the moment θ changes the data is off-policy and the surrogate objective stops being valid.

In simulation that is a fine trade — spin up 4096 parallel environments and collect a million transitions per minute. On a physical arm, a million transitions is months of continuous operation, several gearbox replacements, and a human supervising throughout.

Off-policy methods keep every transition in a replay buffer and reuse it indefinitely. The cost is that we re-enter the deadly triad: function approximation, bootstrapping, and off-policy data, all three at once. Chapter 9's engineering — replay to broaden the update distribution, target networks to throttle the feedback loop — carries over, and this chapter adds the fixes that continuous actions specifically require.

11.2 The deterministic policy gradient

Chapter 9's obstacle was $\max_a Q(s, a)$ over a continuous action space. The trick is to stop computing the max and *learn* it: maintain a deterministic policy $\mu_\theta(s)$ trained to output the maximizing action.

Theorem 11.1 — Deterministic policy gradient (Silver et al., 2014). For a deterministic policy $\mu_\theta : \mathcal{S} \to \mathcal{A}$,

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)}\right],$$

where ρ^β is the state distribution of an arbitrary behaviour policy β.

Two features make this the right tool. **The expectation is over states only** — there is no integral over actions, because the policy is deterministic, so the estimator's variance does not grow with action dimension the way the stochastic policy gradient's does. And **the state distribution belongs to the behaviour policy**, which is precisely what licenses off-policy training from a replay buffer.

The gradient is a chain rule through the critic: differentiate Q with respect to the action, then push that back through the actor. In practice the actor's loss is simply $-Q_\phi(s, \mu_\theta(s))$, and autodiff handles the rest.

DDPG puts this together with Chapter 9's machinery: replay buffer, target networks (soft-updated by Polyak averaging $\theta^- \leftarrow \eta\theta + (1-\eta)\theta^-$ rather than periodic copying), and exploration by adding noise to the deterministic action.

⚠ DDPG'S REPUTATION IS DESERVED

DDPG is famously brittle. It is hyperparameter-sensitive, seed-sensitive, and prone to collapsing after appearing to work. Reproducibility studies have found performance varying wildly across seeds with identical configurations.
The root cause is not exotic. It is Chapter 6's maximization bias, amplified by continuous actions and compounded by bootstrapping — and the next section is about why it is so much worse here than it was for tables.

11.3 Overestimation, and why continuous control makes it worse

Chapter 6 proved that $\mathbb{E}[\max_a Q] \geq \max_a \mathbb{E}[Q]$: taking the max of noisy estimates overestimates. DDPG does not compute an explicit max, but its actor is *trained to find* the maximizing action, which is the same operation performed by gradient ascent.

Worse, the actor is optimizing against the critic's errors. Wherever the critic has an erroneously high value — a spurious bump in a region of action space with little data — the actor will find it and exploit it, because that is exactly what

"maximize Q" instructs it to do. The critic then bootstraps from that inflated value, and the error propagates into the next target.

The result is a feedback loop that inflates values throughout the state space. It is not a small bias; measured overestimation in DDPG can be an order of magnitude.

TD3 applies three fixes, each targeting a distinct part of the loop.

Clipped double-Q. Maintain two critics and use the *minimum* for the target:

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i^-}\left(s', \tilde{a}'\right).$$

Taking the minimum of two independent estimates is deliberately pessimistic — it induces underestimation, which is safe because underestimation does not get amplified by the actor's maximization. This is Chapter 6's double estimator, adapted: rather than decoupling selection from evaluation, it simply refuses to believe the more optimistic critic.

Delayed policy updates. Update the actor once per two critic updates. A policy chasing a critic that is itself still moving amplifies error; letting the critic settle first breaks the resonance.

Target policy smoothing. Add clipped noise to the target action, $\tilde{a}' = \mu_{\theta^-}(s') + \text{clip}(\epsilon, -c, c)$. This regularizes the critic by enforcing that similar actions have similar values, which flattens the narrow spurious peaks the actor would otherwise exploit.

11.4 Maximum entropy: exploration as an objective

Now a genuinely different idea, and the one that produced the most robust algorithm in this chapter.

Standard RL maximizes expected return. **Maximum-entropy RL** maximizes return *plus* the entropy of the policy:

$$J(\pi) = \mathbb{E}\left[\sum_t \gamma^t \Big(R_{t+1} + \alpha\, \mathcal{H}\big(\pi(\cdot \mid S_t)\big)\Big)\right],$$

where α is a **temperature** trading reward against randomness.

The Bellman equation becomes *soft*, with the hard max replaced by a log-sum-exp:

$$Q^{\text{soft}}(s,a) = r + \gamma\, \mathbb{E}_{s'}\left[\alpha \log \int \exp\left(Q^{\text{soft}}(s',a')/\alpha\right) da'\right],$$

and the optimal policy is the Boltzmann distribution $\pi^*(a \mid s) \propto \exp\left(Q^{\text{soft}}(s,a)/\alpha\right)$. As $\alpha \to 0$ the log-sum-exp becomes a max and standard RL is recovered.

INTERACTIVE · ch11-entropy-dial

The entropy temperature

The maximum-entropy optimal policy $\pi^*(a \mid s) \propto \exp(Q(s,a)/\alpha)$ over a fixed Q-landscape. As $\alpha \to 0$ it collapses to a deterministic spike; raised, it hedges across every action worth considering.

Run this simulation in the web edition: /chapters/off-policy-continuous-control #ch11-entropy-dial

THREE REASONS A ROBOT BENEFITS FROM THIS

Exploration is automatic and state-dependent. The policy is stochastic where the values are similar and sharp where they are not — precisely where exploration is and is not worth its cost. No ε schedule to tune.
Multi-modality survives. When two solutions are nearly equally good, a deterministic policy must commit and discard the alternative. The max-entropy policy keeps both alive in proportion to value. If the dynamics shift slightly — a heavier payload, a different friction — the fallback is already in the policy.
Robustness follows. A policy that performs well across a distribution of actions rather than at one exact action tolerates the modelling error and actuation noise that separate simulation from hardware. This is a large part of why SAC transfers as well as it does.

SAC implements this with two critics (TD3's clipped double-Q), a stochastic Gaussian actor with a `tanh` squashing to respect action bounds, and — crucially — **automatic temperature tuning**. Rather than fixing α, solve a constrained problem: maximize return subject to the policy's entropy exceeding a target $\bar{\mathcal{H}}$. The Lagrangian dual gives

$$\alpha \leftarrow \alpha - \lambda \nabla_\alpha \mathbb{E}\left[-\alpha \log \pi(a \mid s) - \alpha \bar{\mathcal{H}}\right],$$

which raises α when the policy becomes too deterministic and lowers it when too random. Setting $\bar{\mathcal{H}} = -\dim(\mathcal{A})$ works across a wide range of tasks and removes the algorithm's most sensitive hyperparameter.

The actor's gradient uses the **reparameterization trick**: write $a = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \epsilon)$ with $\epsilon \sim \mathcal{N}(0, I)$, so the randomness is an input rather than something to differentiate through. Gradients then flow directly into μ and σ, giving far lower variance than the score-function estimator of Chapter 10.

THE TANH CORRECTION

Squashing through `tanh` changes the density, and the log-probability must be corrected by the Jacobian:

$$\log \pi(a \mid s) = \log \mathcal{N}(u \mid \mu, \sigma) - \sum_i \log\left(1 - \tanh^2(u_i)\right).$$

Omitting this term is one of the most common SAC implementation bugs, and it produces entropy estimates that are quietly and consistently wrong.

RUST **rl-deep/src/sac.rs**

```rust
use burn::prelude::*;

pub struct Sac<B: Backend> {
    actor: GaussianPolicy<B>,
    q1: Critic<B>,
    q2: Critic<B>,
    q1_target: Critic<B>,
    q2_target: Critic<B>,
    log_alpha: Tensor<B, 1>,
    target_entropy: f32,   // conventionally -dim(A)
    gamma: f32,
    eta: f64,              // Polyak coefficient for soft target updates
}

impl<B: AutodiffBackend> Sac<B> {
    fn critic_target(&self, batch: &Batch<B>) -> Tensor<B, 1> {
        // Sample the next action from the CURRENT policy (not a target actor).
        let (next_a, next_logp) = self.actor.sample(batch.next_states.clone());

        let q1 = self.q1_target.forward(batch.next_states.clone(), next_a.clone());
        let q2 = self.q2_target.forward(batch.next_states.clone(), next_a);

        // Clipped double-Q: believe the more pessimistic critic. Underestimation
        // is safe; overestimation is what the actor learns to exploit.
        let min_q = q1.min_pair(q2);
        let alpha = self.log_alpha.clone().exp();

        // Soft target: the entropy term enters the bootstrap itself.
        let soft = min_q - next_logp * alpha;
        (batch.rewards.clone() + soft.mul_scalar(self.gamma) * batch.not_done.clone()).
   detach()
    }

    fn actor_loss(&self, batch: &Batch<B>) -> (Tensor<B, 1>, Tensor<B, 1>) {
        // Reparameterized sample: a = tanh(mu + sigma * epsilon), so gradients flow into
        // mu and sigma directly rather than through a score-function estimator.
        let (a, logp) = self.actor.sample(batch.states.clone());

        let q1 = self.q1.forward(batch.states.clone(), a.clone());
        let q2 = self.q2.forward(batch.states.clone(), a);
        let min_q = q1.min_pair(q2);

        let alpha = self.log_alpha.clone().exp().detach();
        let loss = (logp.clone() * alpha - min_q).mean();
```

```
        (loss, logp)
    }

    /// Lagrangian dual: push alpha up when the policy gets too deterministic.
    fn temperature_loss(&self, logp: Tensor<B, 1>) -> Tensor<B, 1> {
        let target = self.target_entropy;
        -(self.log_alpha.clone().exp() * (logp.detach() + target)).mean()
    }
}
```

The SAC update in burn. Note η for the Polyak coefficient — τ is reserved for joint torque throughout this book. Replay and target-network infrastructure is imported unchanged from Chapter 9.

11.5 PPO or SAC? A decision procedure

The honest answer is that both are correct choices in different regimes, and the deciding factor is almost never algorithmic elegance.

Choose PPO when simulation is cheap and parallelizable, the sim-to-real gap is the dominant risk, or you value stability and easy debugging over sample count. Locomotion trained in massively parallel simulators is PPO territory, and Chapter 18 shows why: with 4096 environments, sample efficiency stops mattering and PPO's robustness to hyperparameters is worth more.

Choose SAC when samples are genuinely expensive — real-robot learning, slow simulators, contact-rich manipulation where each rollout takes seconds — or when you plan to fine-tune from offline data (Chapter 16). SAC typically reaches good policies in 5–20× fewer environment steps.

In practice many manipulation results use SAC while most locomotion results use PPO, and that split reflects exactly this trade rather than any deep truth about the competencies.

11.6 Chapter bridge

We recovered sample efficiency by returning to off-policy learning, then spent the chapter making it survivable: clipped double-Q against overestimation, delayed updates and target smoothing against resonance, and maximum entropy turning exploration from a tuned schedule into an optimized quantity.

Every method so far — value-based or policy-based, on-policy or off — learns from experience with the environment and nothing else. The environment remains a black box that is sampled, never understood.

Chapter 12 opens the box. If the robot learns a *model* of its dynamics, it can rehearse in imagination: generate synthetic experience, plan through predicted futures, and extract far more from each real transition. Kober's survey calls this mental rehearsal and identifies it as one of the few genuinely effective answers to the sample-cost problem. The catch — model bias compounding with the rollout horizon — is what the chapter spends most of its length managing.

Exercises

1 FOUNDATION • • • **Derive the DPG**

Derive the deterministic policy gradient theorem, and show it is the limit of the stochastic policy gradient as the policy variance goes to zero. Where does the action integral disappear?

2 FOUNDATION • • • **Bias accumulation**

Model per-update overestimation as a constant ε and show how bootstrapping compounds it over a horizon. Then show that taking the minimum of two independent critics induces a bias of the opposite sign, and explain why that asymmetry is safe.

3 FOUNDATION • • • **Soft policy iteration**

Prove that soft policy evaluation converges (the soft Bellman operator is a γ-contraction) and that soft policy improvement does not decrease the soft value. Together these give soft policy iteration — the SAC analogue of Chapter 5's GPI.

4 FOUNDATION • • **The tanh Jacobian**

Derive the log-probability correction for a tanh-squashed Gaussian. Then compute the entropy error that results from omitting it, and explain how that error would corrupt automatic temperature tuning.

5 CONCEPTUAL • **Temperature and commitment**

In the entropy dial with the bimodal Q-landscape, find the temperature at which the policy stops maintaining both modes and collapses onto one. Explain what a robot loses at that point.

6 CONCEPTUAL • • **Entropy targets**

The standard target entropy is $-\dim(\mathcal{A})$. Reason about what happens for a 1-DoF versus a 12-DoF robot, and argue whether the convention is scale-appropriate.

7 PRACTICAL • • • **The TD3 ablation**

Implement DDPG, then add TD3's three fixes one at a time. Run all four variants on Reacher across five seeds and report both mean performance and seed variance. Identify which single fix contributes most.

8 PRACTICAL • • • **PPO versus SAC, honestly**

Race PPO and SAC on Reacher with equal wall-clock budget and equal environment-step budget. The winner should differ between the two protocols — explain the reversal and say which protocol matches your robot.

11.7 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§3.2 the curse of real-world samples — the constraint that makes off-policy learning worth its risks.

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.

Chapter 6 for maximization bias, Chapter 13 for the actor–critic structure these methods extend.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.

§3.3 and §5 — the empirical split between on-policy locomotion and off-policy manipulation described in §11.5.

FURTHER READING AND MODERN SOURCES

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. & Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms *ICML 2014*.

Theorem 11.1, and the argument that deterministic gradients scale better with action dimension.

Lillicrap, T. P. et al. (2016). Continuous control with deep reinforcement learning *ICLR 2016*.

DDPG — the DPG theorem combined with DQN's replay and target networks.

Fujimoto, S., van Hoof, H. & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods *ICML 2018*.

TD3: the overestimation analysis and the three fixes of §11.3.

Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor *ICML 2018*.

SAC, with the soft policy iteration convergence proof.

Haarnoja, T. et al. (2019). Soft Actor-Critic Algorithms and Applications *arXiv:1812.05905*.

Automatic temperature tuning via the Lagrangian dual, and real-robot results.

Ziebart, B. D. (2010). Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy *PhD thesis, Carnegie Mellon University*.
The maximum-entropy framework that SAC operationalizes — and the ancestor of Chapter 16's MaxEnt IRL.

CHAPTER 12

Model-Based RL & World Models

> *Mental rehearsal — using a learned forward model to simulate experience — has long been one of the most promising routes to reducing the number of real-world trials a robot must perform.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On learning with forward models
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Every method so far treats the environment as a black box to be sampled. But a robot's dynamics are physics — structured, learnable, and largely reusable across tasks. If the robot learns a model, it can rehearse in imagination: plan through predicted futures and generate synthetic experience, extracting far more from each real transition. The catch is that a learned model is wrong, and rolling it forward compounds that error geometrically. This chapter is mostly about managing that compounding, which is what separates model-based methods that work from the ones that famously do not.

FOUNDATION

Ensemble dynamics models, the epistemic/aleatoric decomposition via the law of total variance, the compounding-error bound, CEM as cross-entropy optimization, MBPO's return-gap corollary, and the ELBO for latent dynamics.

CONCEPTUAL

An ensemble of learned models rolled forward from one state, fanning out — with the measured disagreement tracked against the geometric bound.

PRACTICAL

Ensemble dynamics in burn and a CEM-MPC controller solving Pendle's swing-up from a model learned in minutes of simulated experience.

AFTER THIS CHAPTER YOU CAN

- Explain why model-based methods are sample-efficient and where that efficiency comes from
- Decompose predictive uncertainty into epistemic and aleatoric parts and say which one ensembles capture
- State the compounding-error bound and use it to choose a rollout horizon
- Derive CEM as importance-sampled optimization and implement MPC around it
- Explain MBPO's branched rollouts and why short imagination beats long imagination
- Describe what a latent world model learns and why it enables learning from pixels

12.1 What a model buys

A **model** predicts what the environment will do: $\hat{p}(s' \mid s, a)$, and usually a reward model $\hat{r}(s, a)$ as well. Given one, three things become possible that were not before.

Generate synthetic experience. Chapter 7's Dyna already did this with a tabular model. Each real transition can be mined many times over by training on imagined ones.

Plan at decision time. Optimize a short action sequence against the model, execute the first action, re-plan. This is model predictive control, and it needs no policy network at all.

Transfer across tasks. The dynamics of a robot arm do not change when the goal moves. A model learned for reaching is immediately useful for pushing — whereas a value function must be relearned from scratch. For a robot expected to do many things, this is the deepest argument for model-based methods.

The empirical payoff is large. Model-based methods routinely reach good policies in 10–100× fewer environment steps than model-free ones. PILCO famously learned cart-pole swing-up in under 20 seconds of real robot interaction, at a time when model-free methods needed hours.

12.2 Learning dynamics, and knowing what you do not know

The naive approach — train a network to minimize $\|\hat{f}(s,a) - s'\|^2$ — produces a model that is confidently wrong off the training distribution, which is exactly where a planner will take it. The planner is an *adversary* that seeks out wherever the model is optimistic.

The fix is to model uncertainty explicitly, and to distinguish two kinds.

Aleatoric uncertainty is noise inherent in the system: sensor noise, contact stochasticity, unmodelled disturbances. More data does not reduce it. Capture it by predicting a distribution — a Gaussian with learned mean and variance — rather than a point.

Epistemic uncertainty is ignorance: the model has not seen this region of state space. More data *does* reduce it, and this is the kind that matters for planning, because it tells you where not to trust yourself. Capture it with an **ensemble** of K models trained on bootstrapped data with different initializations; where they agree, the model is confident, and where they disagree, it is guessing.

The law of total variance separates them cleanly:

$$\underbrace{\mathrm{Var}[s']}_{\text{total}} = \underbrace{\mathbb{E}_k\left[\sigma_k^2\right]}_{\text{aleatoric}} + \underbrace{\mathrm{Var}_k\left[\mu_k\right]}_{\text{epistemic}},$$

where k indexes ensemble members. The first term is the average predicted noise; the second is the disagreement between members.

12.3 Why imagination must be short

Here is the central difficulty, and it is not subtle once stated.

Suppose the model has per-step error at most ε, and the true dynamics are Lipschitz with constant L. Roll the model forward H steps, feeding each prediction back in as the next input. The error after H steps is bounded by

$$\varepsilon_H \le \varepsilon \cdot \frac{L^H - 1}{L - 1}.$$

For $L > 1$ — which holds for any system with unstable or chaotic modes, meaning essentially every interesting robot — this is **geometric in the horizon**. Doubling the rollout length does far more than double the error.

INTERACTIVE · ch12-imagination-fan

Imagination diverges from reality

An ensemble of learned models rolled forward from one state, fanning out as errors compound. Measured disagreement is plotted against the geometric bound $\varepsilon(L^H - 1)/(L - 1)$.

Run this simulation in the web edition: /chapters/model-based-rl #ch12-imagination-fan

Every ensemble member in that widget has small per-step error. The fan is not caused by bad models; it is caused by feedback. Each prediction becomes the next input, so the rollout steadily walks off the data distribution into a region where all bets are off.

THE MBPO INSIGHT: BRANCH, DO NOT ROLL

If long rollouts are unreliable, do not take them. **MBPO** starts each imagined rollout from a state drawn from the *real* replay buffer and runs it for only k steps — often $k = 1$ to 5 — then trains a model-free algorithm (SAC) on the mixture of real and imagined data.

The return gap between the true and model-based objectives can be bounded in terms of k and the model error, and the bound has an interior optimum: short rollouts give little benefit, long ones accumulate too much error. The practical consequence is a rule worth remembering — **branch many short rollouts off real states rather than imagining whole episodes.**

Notice that this is Chapter 7's Dyna with the branch length made a first-class hyperparameter and chosen by an error bound rather than by taste.

12.4 Planning with a learned model: CEM and MPC

Given a model, one can skip the policy entirely and optimize actions directly at each step.

Random shooting samples N action sequences, evaluates each through the model, and executes the first action of the best. Simple, embarrassingly parallel, and weak in high dimensions.

The cross-entropy method iterates that idea. Sample sequences from a Gaussian over action trajectories, keep the top-performing "elite" fraction, refit the Gaussian to the elites, repeat. It is importance-sampled optimization: each iteration reweights the sampling distribution toward the high-return region, and the elite-refitting step is the maximum-likelihood update of that reweighting.

Model predictive control wraps either in a receding horizon: plan H steps, execute one, discard the rest, re-plan from the state you actually reached. Re-planning is what makes MPC robust — the model only needs to be accurate for

the few steps before the next correction, which is exactly the regime §12.3 says it can manage.

PETS combines all of this: an ensemble of probabilistic models, CEM planning, and trajectory sampling that propagates particles through ensemble members so that the planner's return estimates account for both uncertainty types.

RUST **rl-deep/src/planning/cem.rs**

```rust
use burn::prelude::*;

pub struct CemPlanner {
    horizon: usize,
    population: usize,
    elites: usize,
    iterations: usize,
    action_dim: usize,
}

impl CemPlanner {
    /// Returns the first action of the best plan -- MPC discards the rest.
    pub fn plan<B: Backend>(
        &self,
        ensemble: &DynamicsEnsemble<B>,
        state: &Tensor<B, 1>,
        reward: impl Fn(&Tensor<B, 2>, &Tensor<B, 2>) -> Tensor<B, 1>,
        rng: &mut impl rand::Rng,
    ) -> Vec<f32> {
        // Sampling distribution over action SEQUENCES, refit each iteration.
        let mut mean = vec![0.0f32; self.horizon * self.action_dim];
        let mut std = vec![0.5f32; self.horizon * self.action_dim];

        for _ in 0..self.iterations {
            let mut candidates: Vec<(f32, Vec<f32>)> = Vec::with_capacity(self.population
    );

            for _ in 0..self.population {
                let seq: Vec<f32> = mean
                    .iter()
                    .zip(&std)
                    .map(|(&m, &s)| m + s * gaussian(rng))
                    .collect();

                // Roll the plan forward, switching ensemble member each step so
                // the return averages over epistemic uncertainty (TS-inf).
                let mut s = state.clone();
                let mut total = 0.0f32;
                for t in 0..self.horizon {
                    let a = slice_action(&seq, t, self.action_dim);
                    let member = rng.gen_range(0..ensemble.len());
                    let next = ensemble.predict(member, &s, &a);
                    total += reward(&s.clone().unsqueeze(), &a.clone().unsqueeze())
                        .into_scalar()
                        .elem::<f32>();
                    s = next;
                }
                candidates.push((total, seq));
            }

            // Keep the elites and refit -- the cross-entropy step.
```

```
            candidates.sort_by(|a, b| b.0.partial_cmp(&a.0).unwrap());
            let elite = &candidates[..self.elites];
            for i in 0..mean.len() {
                let vals: Vec<f32> = elite.iter().map(|(_, s)| s[i]).collect();
                let m = vals.iter().sum::<f32>() / vals.len() as f32;
                let v = vals.iter().map(|x| (x - m).powi(2)).sum::<f32>() / vals.len() as
    f32;
                mean[i] = m;
                std[i] = v.sqrt().max(0.05); // floor keeps the search from collapsing
            }
        }
        mean[..self.action_dim].to_vec()
    }
}
```

CEM-MPC with ensemble propagation. Each particle is assigned a random ensemble member per step, so returns average over epistemic uncertainty rather than trusting one model.

12.5 Latent world models

Everything above predicts in state space. For a robot with cameras, the observation is a 64×64×3 image, and predicting pixels is both expensive and wasteful — most pixel-level detail is irrelevant to control.

Latent world models learn a compact latent state z_t in which dynamics are simple, and predict *there*. The Dreamer family learns four components jointly: an encoder $q(z_t \mid z_{t-1}, a_{t-1}, o_t)$, a transition model $p(z_t \mid z_{t-1}, a_{t-1})$, a reward predictor, and a decoder used only as a training signal.

Training maximizes an evidence lower bound with the familiar two-term structure:

$$\mathcal{L} = \underbrace{\mathbb{E}_q\left[\log p(o_t \mid z_t) + \log p(r_t \mid z_t)\right]}_{\text{reconstruction}} - \underbrace{\beta\, D_{\mathrm{KL}}\big(q(z_t \mid \cdot)\,\|\,p(z_t \mid z_{t-1}, a_{t-1})\big)}_{\text{consistency}}.$$

The reconstruction term forces the latent to retain what matters; the KL term forces the transition model to predict it. The payoff is that the **policy is then trained entirely inside the latent model** — thousands of imagined trajectories per real episode, at negligible cost, with no rendering.

> **WHERE WORLD MODELS STAND FOR ROBOTICS**
>
> Dreamer-family agents have learned real-robot tasks in under an hour of interaction without simulators, which is a genuinely striking result given how much of this book's Part III is about building good simulators.
> The honest caveats: they are complex to implement and tune, they inherit the horizon limits of §12.3, and Tang's survey finds model-learning still used in a minority of successful real-world systems. The trajectory is

promising rather than settled — which is a fair description of most of the frontier.

12.6 Model-based versus model-free, decided

The trade is clean enough to state as a rule.

Model-based methods win when **samples are expensive and dynamics are learnable** — a real robot, a slow simulator, smooth dynamics without hard contact. Model-free methods win when **samples are cheap or dynamics are hard to model** — massively parallel simulation, contact-rich manipulation where the model would have to capture exactly the discontinuities it is worst at.

Hybrids increasingly win outright: learn a model, use it to generate short imagined rollouts, and train a model-free algorithm on the mixture. MBPO is that recipe, and the boundary between the two families is genuinely blurring.

There is also a robotics-specific option this chapter has been circling: rather than learning the dynamics from scratch, **fit the parameters of a physics simulator you already trust** — masses, friction coefficients, motor constants. That is system identification, it needs far less data than learning a network, and Chapter 15 builds it.

12.7 Chapter bridge

Part II is complete. We can approximate value functions (Chapter 8), train deep value-based agents on discrete actions (Chapter 9), optimize policies directly with stable on-policy updates (Chapter 10), learn off-policy in continuous action spaces with maximum entropy (Chapter 11), and learn dynamics models to rehearse in imagination (this chapter).

Every one of those chapters treated the robot as an abstract environment — a thing that returns s' and r. Part III opens the box. Chapter 13 asks what is actually inside when the environment is a physical machine: kinematics, Jacobians, the manipulator equation, and the classical controllers — PID, LQR, operational-space control — that RL must be measured against rather than assumed to beat. Reacher stops being a task and becomes a mechanism, with dynamics we derive from first principles.

Exercises

1 FOUNDATION •• **Decompose the variance**

Derive the law-of-total-variance decomposition of §12.2 and explain which term shrinks with more data. Then design an experiment on Pendle that would measure each

separately.

2 FOUNDATION • • • **The compounding bound**

Derive the geometric error bound for H-step rollouts under a Lipschitz assumption. Evaluate it for $\varepsilon = 0.01$ and $L = 1.1$ at $H = 5, 20, 100$, and state the largest horizon you would trust.

3 FOUNDATION • • • **CEM as importance sampling**

Show that CEM's elite-refitting step is the maximum-likelihood update of an importance-sampling distribution tilted toward high return. What does the variance floor in the code correspond to in that framing?

4 FOUNDATION • • • **The ELBO**

Derive the evidence lower bound for a latent dynamics model, identifying the reconstruction and KL terms. Explain what β controls and what happens at $\beta = 0$.

5 CONCEPTUAL • **Find the trust horizon**

In the imagination fan, find the horizon at which ensemble disagreement crosses 0.5 for $\varepsilon = 0.01, 0.03, 0.06$. Plot trust horizon against ε and comment on the shape.

6 CONCEPTUAL • • **Does more ensemble help?**

Increase the ensemble from 2 to 10 members at fixed ε. Does the disagreement estimate become more reliable, and does the trust horizon move? Distinguish between estimating uncertainty better and having less of it.

7 PRACTICAL • • • **CEM-MPC on Pendle**

Learn an ensemble dynamics model of Pendle from random interaction, then solve swing-up with CEM-MPC. Report the total real environment steps used, and compare against SAC from Chapter 11 on the same task.

8 PRACTICAL • • • **Branch length sweep**

Implement MBPO-style branched rollouts and sweep $k \in \{1, 2, 5, 10, 25\}$. Plot final performance against k. There should be an interior optimum — locate it and check it against the bound from §12.3.

12.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§6 tractability through models — core issues in mental rehearsal, and the successful forward-model approaches this chapter modernizes.

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapter 8 — Dyna and the planning/learning unification that MBPO refines.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.3 model-learning axis — the evidence on how often learned models appear in successful real-world systems.

FURTHER READING AND MODERN SOURCES

Deisenroth, M. P. & Rasmussen, C. E. (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search *ICML 2011*.
Gaussian-process dynamics with analytic uncertainty propagation — still the sample-efficiency benchmark to beat.

Chua, K., Calandra, R., McAllister, R. & Levine, S. (2018). Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models *NeurIPS 31*.
PETS: probabilistic ensembles with trajectory sampling, and the epistemic/aleatoric separation of §12.2.

Janner, M., Fu, J., Zhang, M. & Levine, S. (2019). When to Trust Your Model: Model-Based Policy Optimization *NeurIPS 32*.
MBPO, and the return-gap bound that justifies short branched rollouts.

Hafner, D., Lillicrap, T., Ba, J. & Norouzi, M. (2020). Dream to Control: Learning Behaviors by Latent Imagination *ICLR 2020*.
Dreamer — policy learning entirely inside a learned latent model.

Hafner, D., Pasukonis, J., Ba, J. & Lillicrap, T. (2023). Mastering Diverse Domains through World Models *arXiv:2301.04104*.
DreamerV3 — fixed hyperparameters across a wide domain range, which is the practical breakthrough.

Rubinstein, R. Y. & Kroese, D. P. (2004). The Cross-Entropy Method *Springer*.
CEM in its general form, including the importance-sampling derivation of §12.4.

Part III

The Robotics Side

CHAPTER 13

The Robot as an Environment: Kinematics, Dynamics & Control

> *A reinforcement learning practitioner who does not understand the plant is not doing control — they are doing curve fitting and hoping.*
>
> **Adapted from the classical-control tradition** · The premise of Part III
> *Siciliano & Khatib, Springer Handbook of Robotics*

Twelve chapters have treated the environment as a black box returning s′ and r. Part III opens it. When the environment is a physical machine, it has structure: rigid bodies with known geometry, dynamics that follow from Lagrangian mechanics, and decades of control theory that already solves large parts of the problem. This chapter derives that structure for Reacher from first principles, builds the classical controllers RL must be measured against, and identifies exactly where learning has something to add — which is a narrower and more interesting set of places than enthusiasm suggests.

FOUNDATION

SE(2)/SE(3), forward and inverse kinematics worked completely, the Jacobian and manipulability, the Lagrangian derivation of $M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau$, LQR via the Riccati recursion, and the EKF.

CONCEPTUAL

Reacher as a mechanism you can drag: watch joint space and task space respond to each other, and see the manipulability ellipsoid collapse at a singularity.

PRACTICAL

The rl-sim crate: URDF chains via urdf-rs and k, Reacher in rapier2d, and PID/LQR baselines with a benchmark suite that later chapters must beat.

AFTER THIS CHAPTER YOU CAN

- Compute forward and inverse kinematics for a two-link arm, including both elbow solutions
- Derive the Jacobian, use it to map velocities and forces, and identify singularities
- Derive the manipulator equation from the Lagrangian and interpret each term physically
- Derive the LQR controller from the Riccati equation and recognize it as Chapter 5's value iteration
- Explain what PID, LQR and operational-space control each do well, and what defeats them
- Identify the four distinct places learning can enter a robot control stack

13.1 Configuration, pose, and the two spaces

A robot's **configuration** $q \in \mathbb{R}^n$ lists its joint variables — for Reacher, two angles $q = (q_1, q_2)$. Its **pose** is where the end-effector actually is, an element of $SE(2)$ in the plane (position plus orientation) or $SE(3)$ in space.

Everything in robot control lives in the tension between these two descriptions. You *command* joint space, because that is where the motors are. You *care about* task space, because that is where the mug is.

Forward kinematics maps one to the other. For Reacher with link lengths ℓ_1, ℓ_2:

$$x = \ell_1 \cos q_1 + \ell_2 \cos(q_1 + q_2), \qquad y = \ell_1 \sin q_1 + \ell_2 \sin(q_1 + q_2).$$

This is always well defined, cheap, and unique. **Inverse kinematics** goes the other way and is where the difficulties live. Squaring and adding:

$$x^2 + y^2 = \ell_1^2 + \ell_2^2 + 2\ell_1\ell_2 \cos q_2 \quad \Longrightarrow \quad \cos q_2 = \frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2\ell_1\ell_2},$$

so

$$q_2 = \pm \arccos\left(\frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2\ell_1\ell_2}\right), \qquad q_1 = \operatorname{atan2}(y, x) - \operatorname{atan2}\left(\ell_2 \sin q_2,\ \ell_1 + \ell_2 \cos q_2\right).$$

READ THE ± SIGN

That sign is the **elbow-up / elbow-down** choice: two distinct joint configurations reaching the identical point. Real arms have more — a 7-DoF arm has a continuous infinity of them.

This is redundancy, and it is a resource rather than a nuisance: it lets a robot reach around obstacles, stay away from joint limits, and keep its elbow out of a human's way, all while holding the end-effector fixed. Chapter 19 uses exactly this freedom for whole-body control.

The solution also fails to exist when $\sqrt{x^2 + y^2} > \ell_1 + \ell_2$ (too far) or $< |\ell_1 - \ell_2|$ (inside the dead zone). The reachable workspace is an annulus, and a policy commanding task-space targets must respect it — an action-space design question Chapter 17 takes up.

13.2 The Jacobian: velocities, forces, and singularities

Differentiating forward kinematics gives the **Jacobian** $J(q)$, mapping joint velocities to end-effector velocity, $\dot{p} = J(q)\,\dot{q}$. For Reacher:

$$J(q) = \begin{bmatrix} -\ell_1 s_1 - \ell_2 s_{12} & -\ell_2 s_{12} \\ \ell_1 c_1 + \ell_2 c_{12} & \ell_2 c_{12} \end{bmatrix},$$

with $s_1 = \sin q_1$, $s_{12} = \sin(q_1 + q_2)$ and so on.

The Jacobian does three jobs at once. It maps velocities forward. It maps forces backward — the **static force relation** $\tau = J^\top F$ says the joint torques needed to exert an end-effector force F are the Jacobian transpose applied to it, which is how force control is implemented. And it diagnoses **singularities**.

For Reacher, $\det J = \ell_1 \ell_2 \sin q_2$, which vanishes at $q_2 = 0$ (arm fully extended) and $q_2 = \pi$ (fully folded). At those configurations the arm loses the ability to move in one direction entirely, and any controller that inverts J demands infinite joint velocity.

The **manipulability ellipsoid** — the image of the unit joint-velocity ball under J — visualizes this: it is round where the arm is dexterous and flattens to a line at a singularity.

INTERACTIVE · `ch13-kinematics-sandbox`

Reacher: joint space and task space

Drag the end-effector to solve inverse kinematics, or move the joints directly. The manipulability ellipsoid rounds out where the arm is dexterous and flattens to a line as $\det J = \ell_1 \ell_2 \sin q_2$ approaches zero.

Run this simulation in the web edition: `/chapters/robot-as-environment`

```
#ch13-kinematics-sandbox
```

⚠ WHAT THIS MEANS FOR A LEARNED POLICY

A policy that outputs task-space velocities inherits the singularity problem: it will happily command motions the arm cannot execute near a singular configuration.
A policy that outputs joint commands does not have this problem at all — it simply cannot express an impossible motion. This is the first concrete instance of a theme that runs through Part III: **the action space you choose determines which failures are possible.** Chapter 17 makes it the organizing question.

13.3 Dynamics: the manipulator equation

Kinematics ignores mass. To command torques we need dynamics, and the systematic route is Lagrangian mechanics.

Form the Lagrangian $\mathcal{L} = T - V$ (kinetic minus potential energy) and apply the Euler–Lagrange equation:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} = \tau_i.$$

For any serial manipulator this always produces the same structure — the **manipulator equation**:

$$M(q)\,\ddot{q} + C(q,\dot{q})\,\dot{q} + g(q) = \tau.$$

Each term has a clean physical reading. $M(q)$ is the **mass matrix**: symmetric, positive definite, and *configuration-dependent* — an extended arm has more rotational inertia than a folded one, which is why a controller tuned in one configuration misbehaves in another. $C(q,\dot{q})\dot{q}$ collects **Coriolis and centrifugal** effects, the velocity-dependent coupling that makes fast motion qualitatively harder than slow motion. $g(q)$ is **gravity**, the term that is usually largest and, helpfully, the easiest to cancel exactly.

For Reacher, writing $\alpha = m_1\ell_{c1}^2 + m_2(\ell_1^2 + \ell_{c2}^2) + I_1 + I_2$, $\beta = m_2\ell_1\ell_{c2}$, $\delta = m_2\ell_{c2}^2 + I_2$:

$$M(q) = \begin{bmatrix} \alpha + 2\beta\cos q_2 & \delta + \beta\cos q_2 \\ \delta + \beta\cos q_2 & \delta \end{bmatrix}, \qquad C(q,\dot{q}) = \begin{bmatrix} -\beta\dot{q}_2\sin q_2 & -\beta(\dot{q}_1+\dot{q}_2)\sin q_2 \\ \beta\dot{q}_1\sin q_2 & 0 \end{bmatrix}.$$

A useful structural fact: $\dot{M} - 2C$ is skew-symmetric, which encodes energy conservation and underpins the stability proofs of most classical robot controllers.

UNDERACTUATION IS THE INTERESTING CASE

Reacher has one actuator per degree of freedom — **fully actuated** — and a fully actuated robot can, in principle, be made to follow any trajectory by inverting its dynamics.

Pendle cannot. A pendulum with a weak motor at the pivot has fewer actuators than degrees of freedom in the directions it needs, and no amount of cleverness lets it move directly to upright; it must pump energy over several swings. That is **underactuation**, and it is why Pendle's swing-up is genuinely hard while Reacher's reaching is not.

Every legged robot is underactuated during flight. Every thrown or caught object is. This is a large part of why locomotion needed RL while pick-and-place, in structured settings, did not.

13.4 Classical control: the baselines RL must beat

Three controllers, in increasing order of how much they know about the robot.

PID knows nothing. It drives the error to zero using proportional, integral and derivative terms:

$$\tau = K_p e + K_i \int e\, dt + K_d \dot{e}.$$

It is model-free, ubiquitous, and remarkably effective for slow, well-geared, fully actuated systems. It struggles with coupling, gravity that varies with configuration, and anything fast.

Computed torque knows the model and cancels it. Choosing

$$\tau = M(q)\big(\ddot{q}_d + K_d \dot{e} + K_p e\big) + C(q, \dot{q})\dot{q} + g(q)$$

makes the closed-loop error dynamics exactly linear: $\ddot{e} + K_d \dot{e} + K_p e = 0$. This is feedback linearization, and it is beautiful when the model is right. Its weakness is precisely its strength — it depends entirely on model accuracy, and Chapter 15 is about how wrong models actually are.

LQR is optimal control for a linearized system, and it deserves special attention here because it is exactly the machinery of Chapter 5 in continuous form. For dynamics $x_{k+1} = Ax_k + Bu_k$ and cost $\sum_k (x_k^\top Q x_k + u_k^\top R u_k)$, the optimal policy is linear state feedback $u = -Kx$ with $K = (R + B^\top P B)^{-1} B^\top P A$, where P solves the discrete algebraic Riccati equation

$$P = Q + A^\top P A - A^\top P B \,(R + B^\top P B)^{-1} B^\top P A.$$

LQR IS VALUE ITERATION WITH QUADRATICS

Assume the value function is quadratic, $V(x) = x^\top P x$. Plug it into the Bellman optimality equation, minimize over u analytically (the objective is quadratic in u, so set the gradient to zero), and the result is another quadratic in x — with the new matrix given by the Riccati recursion.

The Riccati equation **is** the Bellman optimality equation for the linear-quadratic case, and iterating it is value iteration. Chapter 5's contraction argument applies, which is why the recursion converges.

This is worth internalizing: the divide between "control theory" and "reinforcement learning" is largely one of assumptions, not of ideas. LQR is what happens when you can solve the Bellman equation in closed form. RL is what happens when you cannot.

RUST **rl-sim/src/reacher/dynamics.rs**

```rust
use nalgebra::{Matrix2, Vector2};

pub struct ReacherParams {
    pub m1: f64, pub m2: f64,
    pub l1: f64, pub l2: f64,
    pub lc1: f64, pub lc2: f64,
    pub i1: f64, pub i2: f64,
    pub g: f64,
}

impl ReacherParams {
    /// Configuration-dependent mass matrix M(q) -- symmetric, positive definite.
    pub fn mass_matrix(&self, q: &Vector2<f64>) -> Matrix2<f64> {
        let alpha = self.m1 * self.lc1.powi(2)
            + self.m2 * (self.l1.powi(2) + self.lc2.powi(2))
            + self.i1 + self.i2;
        let beta = self.m2 * self.l1 * self.lc2;
        let delta = self.m2 * self.lc2.powi(2) + self.i2;
        let c2 = q[1].cos();

        Matrix2::new(
            alpha + 2.0 * beta * c2, delta + beta * c2,
            delta + beta * c2,       delta,
        )
    }

    /// Coriolis/centrifugal matrix C(q, q_dot).
    pub fn coriolis(&self, q: &Vector2<f64>, dq: &Vector2<f64>) -> Matrix2<f64> {
        let beta = self.m2 * self.l1 * self.lc2;
        let s2 = q[1].sin();
        Matrix2::new(
            -beta * dq[1] * s2, -beta * (dq[0] + dq[1]) * s2,
             beta * dq[0] * s2,  0.0,
        )
    }

    /// Forward dynamics: q_ddot = M-1(tau - C q_dot - g).
    pub fn accel(&self, q: &Vector2<f64>, dq: &Vector2<f64>, tau: &Vector2<f64>) ->
```

```
    Vector2<f64> {
        let m = self.mass_matrix(q);
        let c = self.coriolis(q, dq);
        let grav = self.gravity(q);
        m.try_inverse().expect("M(q) is positive definite") * (tau - c * dq - grav)
    }
}

/// Discrete LQR by iterating the Riccati recursion to its fixed point.
/// This is Chapter 5's value iteration, specialized to quadratic value functions.
pub fn dlqr(a: &Matrix2<f64>, b: &Matrix2<f64>, q: &Matrix2<f64>, r: &Matrix2<f64>)
    -> Matrix2<f64>
{
    let mut p = *q;
    for _ in 0..500 {
        let s = r + b.transpose() * p * b;
        let k = s.try_inverse().unwrap() * b.transpose() * p * a;
        let p_next = q + a.transpose() * p * (a - b * k);
        if (p_next - p).norm() < 1e-12 { p = p_next; break; }
        p = p_next;
    }
    (r + b.transpose() * p * b).try_inverse().unwrap() * b.transpose() * p * a
}
```

Reacher's analytic dynamics and a discrete LQR solver in nalgebra. These are the baselines Chapters 14–20 must beat — and it is worth knowing that on reaching tasks, they often do not lose.

13.5 Where learning actually enters

Given all this machinery, what is left for reinforcement learning? Four distinct entry points, and confusing them is a common source of wasted effort.

Learning the policy — replace the controller entirely. Correct when the dynamics are too complex or contact-rich to model, as in legged locomotion over rough terrain. Usually wasteful when a classical controller already works.

Learning the model — keep classical control, learn the dynamics it needs. This is Chapter 12's territory and Chapter 15's system identification. Often the highest-value option, because a small correction to a mostly-right model goes a long way.

Learning a residual — run a hand-designed controller and let a policy learn a correction on top, $\tau = \tau_{\text{classical}} + \tau_{\text{learned}}$. The classical part guarantees baseline competence and safety; the learned part handles what the model missed. Chapter 17 builds this properly, and it is frequently the pragmatic answer for real deployments.

Learning to tune — treat controller gains as the action space and let RL adapt them online. Modest, low-risk, and underused.

> **⚠ BEAT THE BASELINE, OR EXPLAIN WHY NOT**
>
> Before applying RL to a robot problem, implement the classical controller.

It is usually a day of work, and one of two things will happen.
Either it works — in which case you have saved months and should ship it.
Or it fails — in which case you now know precisely *how* it fails, which tells you what the learned policy must do differently and gives you a number to beat.
Published robot-RL results that do not report a classical baseline should be read with that omission in mind. The benchmark suite this chapter builds exists so that every later chapter in this book can report one.

13.6 State estimation, and Chapter 4's confession revisited

One more piece of reality. The controllers above assume you know q and $\dot{q}$. You measure encoder counts and integrate noisy signals.

The **extended Kalman filter** is the standard answer: maintain a Gaussian belief over the state, propagate it through linearized dynamics (predict), and correct it with each measurement (update). It is Chapter 4's belief update, specialized to Gaussians and linearized dynamics.

This is where Chapter 4's confession becomes concrete. The observation is not the state; a filter produces an estimate with real error; and a policy trained on ground-truth simulator state will encounter estimation error it has never seen. Chapter 15's teacher–student method exists precisely to handle this, and Chapter 19's recurrent policies learn to do the filtering themselves.

13.7 Chapter bridge

The black box is open. Reacher is a mechanism with derivable kinematics, a Jacobian that says what it can and cannot do, dynamics that follow from the Lagrangian, and classical controllers that already solve a respectable fraction of the problem. We know where learning belongs — and, equally useful, where it does not.

Chapter 14 asks what makes robot RL hard in ways that have nothing to do with algorithms. Kober's four curses: the dimensionality we quantified in Chapter 5, the sample cost that shaped Chapter 11, the under-modelling that Chapter 12 fought, and the one we have so far avoided entirely — **goal specification**. Reward functions are the interface between human intent and an optimizer that will exploit any gap in your phrasing, and the results are a catalogue of instructive disasters.

Exercises

1 FOUNDATION •• **Both elbows**

Derive Reacher's inverse kinematics completely, including both elbow solutions and the reachability conditions. Then determine which solution a continuous IK-following controller should choose, and what happens at the boundary between them.

2 FOUNDATION •• **Jacobian and singularities**

Compute $\det J$ for Reacher and identify all singular configurations. At $q_2 = 0$, determine the direction in which end-effector motion becomes impossible, and verify it against the manipulability ellipsoid.

3 FOUNDATION ••• **The full Lagrangian derivation**

Derive $M(q)$, $C(q,\dot{q})$ and $g(q)$ for Reacher from the Lagrangian, showing every step. Verify that M is symmetric positive definite and that $\dot{M} - 2C$ is skew-symmetric.

4 FOUNDATION ••• **LQR is value iteration**

Starting from the Bellman optimality equation with a quadratic value function $V(x) = x^T P x$ and linear dynamics, derive the Riccati recursion. Identify precisely which step corresponds to Chapter 5's max over actions.

5 CONCEPTUAL • **Feel the redundancy**

In the kinematics sandbox, hold the end-effector at a fixed point and move through both elbow solutions. Then move to the workspace boundary and describe what happens to the manipulability ellipsoid.

6 CONCEPTUAL •• **PID versus LQR under disturbance**

Compare PID and LQR on Pendle's stabilization with an impulse disturbance. Tune PID as well as you can. Then increase the disturbance until each fails, and report the margin between them.

7 PRACTICAL ••• **Computed torque**

Implement computed-torque control for Reacher and verify the error dynamics are linear when the model is exact. Then perturb the mass parameters by 10%, 25% and 50%, and measure how tracking degrades. This is a preview of Chapter 15.

8 PRACTICAL •• **The baseline suite**

Build a benchmark harness that evaluates any controller on Reacher reaching and Pendle swing-up with pinned seeds, reporting success rate, settling time and energy. Every later chapter's learned policy will be measured against it.

13.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§1.2 RL in the context of optimal control, and §1.3 on the physical structure that distinguishes robot problems from generic MDPs.

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
Chapter 4 — the value iteration that §13.4 shows LQR to be a closed-form instance of.

FURTHER READING AND MODERN SOURCES

Siciliano, B., Sciavicco, L., Villani, L. & Oriolo, G. (2009). Robotics: Modelling, Planning and Control *Springer*.
The standard reference for everything in §13.1–13.4: kinematics, Jacobians, dynamics, and control.

Spong, M. W., Hutchinson, S. & Vidyasagar, M. (2006). Robot Modeling and Control *Wiley*.
The Lagrangian derivation of the manipulator equation, and the skew-symmetry property used in stability proofs.

Yoshikawa, T. (1985). Manipulability of Robotic Mechanisms *International Journal of Robotics Research 4(2)*.
The manipulability ellipsoid, in the original.

Khatib, O. (1987). A unified approach for motion and force control of robot manipulators: The operational space formulation *IEEE Journal on Robotics and Automation 3(1)*.
Operational-space control — the task-space formulation that Chapter 20's impedance action spaces build on.

Anderson, B. D. O. & Moore, J. B. (1990). Optimal Control: Linear Quadratic Methods *Prentice Hall*.
LQR and the Riccati equation, with the convergence analysis.

Thrun, S., Burgard, W. & Fox, D. (2005). Probabilistic Robotics *MIT Press*.
The EKF and the belief-state machinery of §13.6, developed in full.

CHAPTER 14

The Four Curses of Robot RL

> *The high dimensionality of robot state and action spaces, the cost of real-world experience, model uncertainty, and the difficulty of specifying goals — these four curses shape every practical decision in robot reinforcement learning.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · The organizing framework of Part III
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Nothing in this chapter is an algorithm. It is about the four structural difficulties that make robot reinforcement learning hard regardless of which algorithm you choose — and that decide, more often than algorithmic choices do, whether a project succeeds. Three of them we have met in passing. The fourth we have carefully avoided: reward functions are the interface between human intent and an optimizer that will exploit any gap in your phrasing, and the results are a catalogue of instructive disasters.

FOUNDATION

Sample-complexity intuition, delay-augmented MDPs, the compounding transfer bound, Ng's potential-based shaping invariance theorem with proof, and the constrained-MDP Lagrangian.

CONCEPTUAL

The exponential wall in arithmetic you can set yourself, and a reward-hacking gallery where each specification produces its own pathology.

PRACTICAL

A reward-design lab — one Reacher task, six reward functions, six trained behaviours — plus sample-cost accounting and a Lagrangian-SAC prototype.

AFTER THIS CHAPTER YOU CAN

- Quantify the dimensionality curse and explain why generalization, not computation, is the answer
- Account honestly for the true cost of a real-world sample, including resets and wear
- Explain how model error compounds and why it caps how far simulation can be trusted
- Prove the potential-based shaping theorem and use it to add guidance without changing the optimum
- Recognize reward hacking and diagnose which gap in the specification produced it
- Formulate a constrained MDP and explain why safety belongs in the constraint, not the reward

14.1 The curse of dimensionality

Kober's first curse we have already met twice: Chapter 5 computed the wall, Chapter 8 responded to it.

A 7-DoF arm has a 14-dimensional state before you add an object. Discretize each dimension into 10 bins and you have 10^{14} states. The number is not merely large — it is large in a way that no hardware advance touches, because it grows exponentially in the exponent.

INTERACTIVE · ch14-dimensionality-wall

The exponential wall

Set the degrees of freedom and bins per dimension, and read how long one exhaustive tabular sweep would take. Seven degrees of freedom at ten bins already outlives the solar system.

Run this simulation in the web edition: `/chapters/four-curses #ch14-dimensionality-wall`

The correct response is not more computation but **generalization**: function approximation (Chapter 8) so that experience in one state informs nearby states, and structured policy representations (Chapter 17) so that the space of behaviours the robot considers is far smaller than the space of behaviours it could physically produce.

There is also a **curse of dimensionality in exploration** that function approximation does not address. Random torques on a 7-DoF arm produce flailing that never reaches anything interesting. The volume of state space reachable by a random walk is a vanishing fraction of the whole, and no amount of ε-greedy

fixes that. This is why demonstrations (Chapter 16) and structured primitives (Chapter 17) matter so much in practice.

14.2 The curse of real-world samples

Chapter 11 motivated off-policy learning with sample cost. It is worth accounting honestly for what a sample actually costs, because the accounting changes decisions.

A single episode on a physical arm consumes wall-clock time (seconds to minutes), mechanical wear on gearboxes and cable harnesses, energy, and — most expensively — **a reset**. Something must return the robot and the objects to a start state. Often that something is a human, and human attention is the binding constraint on nearly every real-robot learning project.

Add safety: some transitions damage the robot, the environment, or a person. You cannot sample those freely, which means the very transitions most informative about failure are the ones you must not collect.

> **⚠ LATENCY IS A MODELLING ERROR, NOT AN ANNOYANCE**
>
> Real robots have delay: sensing, computation, communication, and actuator response together give tens of milliseconds between an observation and its action taking effect. The action executed at time t was chosen from the state at $t - d$.
>
> A delayed MDP is **not Markov in the instantaneous state**. The honest formulation augments the state with the actions still in flight, $\tilde{s}_t = (s_t, a_{t-d}, \ldots, a_{t-1})$, which restores the Markov property at the cost of state dimension.
>
> Ignoring this is one of the most common causes of a policy that works in simulation and oscillates on hardware. Chapter 15's randomization includes latency for exactly this reason.

The responses are the architecture of this book's Part III: **simulate** (Chapter 15), **reuse data off-policy** (Chapter 11), **learn from demonstrations** (Chapter 16), and **learn a model** to extract more from each transition (Chapter 12).

14.3 The curse of under-modelling

Simulation is the standard escape from sample cost, and it introduces its own curse: the simulator is wrong.

Contact is where it is most wrong. Rigid-body contact involves complementarity conditions, friction cones, and impacts — phenomena that simulators approximate with solver parameters chosen for numerical stability rather than physical fidelity. Chapter 2 already showed a version of this in miniature: an

integrator that manufactures energy.

The consequence compounds. If per-step model error is ε and the dynamics have Lipschitz constant $L > 1$, then trajectory error after H steps grows like $\varepsilon(L^H - 1)/(L - 1)$ — the same geometric bound as Chapter 12's imagination fan, now applied to the gap between simulation and reality. A policy that exploits simulator artifacts over a long horizon will fail on hardware in exactly the way it succeeded in simulation.

14.4 The curse of goal specification

The fourth curse is the one we have avoided, and it is where most real projects actually founder.

Reward is the entire interface between what you want and what the robot optimizes. It must be a scalar, it must be computable from things the robot can sense, and it will be optimized *literally* — including any gap between what you wrote and what the robot optimizes.

Sparse rewards (+1 on success, 0 otherwise) are easy to specify honestly and nearly impossible to learn from: random exploration essentially never succeeds, so there is no gradient. **Dense rewards** are learnable but must be engineered, and every engineered term is a new opportunity for the optimizer to find something you did not intend.

> ⚠ **A GALLERY OF SPECIFICATION FAILURES**
>
> Each of these is a real pattern, and each traces to a specific gap:
> **Reward the distance to the goal decreasing** → the robot learns to oscillate, collecting the decrease repeatedly. *Gap: you rewarded a rate, not a state.*
> **Reward object height for a lifting task** → the robot flips the table. *Gap: you specified the measurement, not the intent.*
> **Reward forward velocity for locomotion** → the robot learns to fall forward repeatedly. *Gap: no term for staying upright afterwards.*
> **Penalize collisions detected by the sensor** → the robot learns to avoid the sensor's field of view. *Gap: you rewarded the observation, not the fact.*
> **Reward task completion time** → the robot moves violently, damaging itself within its warranty period. *Gap: no cost on effort or wear.*
> The common structure is that reward hacking is not the optimizer misbehaving. It is the optimizer working correctly on the objective you actually specified.

There is one principled tool, and it is worth knowing precisely because it is the only shaping that is guaranteed safe.

Theorem 14.1 — Potential-based shaping invariance (Ng, Harada & Russell, 1999). Let $\Phi : \mathcal{S} \to \mathbb{R}$ be any function of state, and define the shaped reward

$$r'(s, a, s') = r(s, a, s') + \gamma\, \Phi(s') - \Phi(s).$$

Then the optimal policy of the shaped MDP is identical to that of the original. Moreover, this form is the **only** state-dependent shaping with that guarantee.

PROOF

Compute the shaped return along any trajectory. The shaping terms telescope:

$$\begin{aligned} G'_t &= \sum_{k=0}^{\infty} \gamma^k \left[r_{t+k+1} + \gamma \Phi(s_{t+k+1}) - \Phi(s_{t+k}) \right] \\ &= G_t + \sum_{k=0}^{\infty} \left(\gamma^{k+1} \Phi(s_{t+k+1}) - \gamma^k \Phi(s_{t+k}) \right) \\ &= G_t - \Phi(s_t), \end{aligned}$$

since all intermediate terms cancel and $\gamma^k \Phi(s_{t+k}) \to 0$. So $v'_\pi(s) = v_\pi(s) - \Phi(s)$ for every policy: the shaped value differs from the original by a quantity that **does not depend on the policy**. Ordering between policies is therefore preserved, and the argmax is unchanged. $$ ■

HOW TO USE THIS IN PRACTICE

Want to hint that being closer to the goal is better? Set $\Phi(s) = -c \cdot \text{distance}(s, \text{goal})$ and shape with $\gamma\Phi(s') - \Phi(s)$. The hint accelerates learning and the theorem guarantees you have not changed what "optimal" means. Compare with adding $-c \cdot \text{distance}$ directly to the reward. That *does* change the optimal policy — it now trades goal-reaching against staying near the goal, and can produce a robot that hovers near the target rather than completing the task. The difference between these two is one of the highest-leverage pieces of theory in applied robot RL.

The limitation is real, though: potential-based shaping only helps when you can write a Φ that reflects genuine progress. For a long-horizon assembly task, you often cannot, and Chapter 16's demonstrations become the practical answer instead.

14.5 Safety belongs in constraints

Practitioners routinely encode safety as a large negative reward. This is a mistake, and the reason is structural: a scalar reward forces you to price safety against task success, so there always exists a task reward large enough to justify a violation. That is precisely the wrong semantics.

The right formulation is a **constrained MDP**: maximize return subject to expected cost constraints,

$$\max_{\pi} \; \mathbb{E}_{\pi}\left[\sum_{t} \gamma^{t} r_{t}\right] \quad \text{subject to} \quad \mathbb{E}_{\pi}\left[\sum_{t} \gamma^{t} c_{i}(s_{t}, a_{t})\right] \leq d_{i}.$$

The Lagrangian $\mathcal{L}(\pi, \lambda) = J_r(\pi) - \sum_i \lambda_i (J_{c_i}(\pi) - d_i)$ turns this into a saddle-point problem: ascend on π, descend on λ. The multipliers adapt automatically — when a constraint is violated, its price rises until the policy respects it.

Notice this is the same mechanism as SAC's automatic temperature tuning in Chapter 11. Entropy there was a constraint with an adaptively priced multiplier, and safety here is the same construction with a different constrained quantity.

RUST **rl-deep/src/safety/lagrangian.rs**

```rust
/// Adaptive multiplier for a single expected-cost constraint E[Sigma gamma^t c] <= d.
pub struct LagrangianDual {
    log_lambda: f64,
    limit: f64,       // the budget d
    lr: f64,
}

impl LagrangianDual {
    pub fn multiplier(&self) -> f64 {
        self.log_lambda.exp()
    }

    /// Dual ascent: raise the price when the constraint is violated,
    /// lower it (toward zero) when there is slack.
    pub fn update(&mut self, measured_cost: f64) {
        let violation = measured_cost - self.limit;
        self.log_lambda += self.lr * violation;
        // Clamp to keep the price finite when a constraint is badly violated.
        self.log_lambda = self.log_lambda.clamp(-8.0, 4.0);
    }

    /// The effective reward the policy optimizes.
    pub fn shaped_reward(&self, reward: f64, cost: f64) -> f64 {
        reward - self.multiplier() * cost
    }
}
```

A Lagrangian dual for constrained RL. The multiplier is the price of violating the constraint, and it is learned rather than tuned — the same construction as SAC's temperature.

14.6 Evaluating claims honestly

The last section of this chapter is methodological, and it is the one most often skipped.

Robot RL results are hard to evaluate because runs are expensive, so sample sizes are small; seeds matter enormously, so single runs mislead; and success is often reported without confidence intervals or without stating how many attempts were made.

The standards this book holds itself to, and recommends:

Report the level of real-world success. Chapter 1's L0–L5 rubric is a checkable claim, unlike "we demonstrate real-world capability".

Report seeds. Five minimum, with the spread shown, not just the mean. Deep RL's seed variance is large enough that a three-seed comparison frequently reverses on the fourth.

Report the classical baseline. Chapter 13 built the suite for this reason.

Report the sample budget in real units. Not "500k steps" but "14 hours of robot time and 62 human-supervised resets".

Report failures. A 90% success rate means one attempt in ten failed, and how it failed is more informative than the nine that worked.

> **SMALL-SAMPLE STATISTICS**
>
> With 20 trials and 18 successes, the naive rate is 90%. The Wilson score interval — appropriate for proportions near the boundary, where the normal approximation is poor — gives roughly [70%, 97%]. Reporting "90%" without that interval overstates what 20 trials can establish.
> Chapter 22's capstone reports Wilson intervals throughout, because the alternative is to claim precision the experiment did not buy.

14.7 Chapter bridge

Four curses, each with a response. Dimensionality is answered by generalization and structure. Sample cost is answered by simulation, off-policy reuse, demonstrations, and models. Under-modelling is answered by randomization and system identification. Goal specification is answered by potential-based shaping where possible, by demonstrations where reward cannot be written, and by constraints where safety is at stake.

Chapter 15 takes the second and third curses head-on. If simulation is the escape from sample cost, and simulation is wrong, then the central question of practical robot learning is: **how do you train in a simulator you know to be wrong, and get a policy that works anyway?** The answers — domain randomization, system identification, teacher–student distillation — are the machinery behind essentially every L4 result in Chapter 1's ladder. Ferris arrives to carry them.

Exercises

1 FOUNDATION • • • **Prove the shaping theorem**

Reproduce the proof of Theorem 14.1 with every step justified. Then construct a non-potential-based shaping and exhibit a concrete MDP where it changes the optimal policy.

2 FOUNDATION • • **Delay breaks Markov**

Show that a system with one-step actuation delay is not Markov in the instantaneous state, and that augmenting with the in-flight action restores it. What is the state-dimension cost for a delay of d steps on an n-dimensional action?

3 FOUNDATION • • **The transfer bound**

Derive the compounding bound for the sim-to-real gap and evaluate it for $\varepsilon = 0.001$ with $L = 1.05$ at horizons of 100, 500 and 2000 steps. At what horizon does simulation stop being informative?

4 FOUNDATION • • **Why a scalar cannot encode safety**

Show that for any finite safety penalty, there exists a task reward scale at which the optimal policy accepts the violation. Then show the constrained formulation does not have this property.

5 CONCEPTUAL • **Set the wall yourself**

Using the dimensionality widget, find the largest robot (in DoF) for which a tabular sweep at 10 bins per dimension completes within one hour at a million states per second. Compare with the smallest robot you would call interesting.

6 CONCEPTUAL • • **Hack your own reward**

Design a reward for a Reacher task that you believe is unhackable. Then find the exploit — there is almost always one. Write down which gap in the specification you missed.

7 PRACTICAL • • • **The reward-design lab**

Train SAC on the same Reacher task under six reward functions: sparse, dense-distance, potential-shaped, velocity-penalized, effort-penalized, and one you design. Record the resulting behaviours and rank them by what a user would actually want — which will not match the returns.

8 PRACTICAL • • • **Constrained SAC**

Implement Lagrangian-SAC with a joint-velocity constraint. Verify the multiplier rises when the constraint is violated and settles when it is satisfied. Then compare against encoding the same limit as a fixed reward penalty, and report which one respects the limit reliably.

14.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§3 in full: §3.1 dimensionality, §3.2 real-world samples, §3.3 under-modelling and model uncertainty, §3.4 goal specification. This chapter is that section, updated.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.4 the L0–L5 rubric adopted in §14.6, and §5 on real-world learning, reward design and safe exploration as open challenges.

FURTHER READING AND MODERN SOURCES

Ng, A. Y., Harada, D. & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping *ICML 1999*.
Theorem 14.1, including the converse that potential-based shaping is the only safe form.

Altman, E. (1999). Constrained Markov Decision Processes *Chapman & Hall/CRC*.
The CMDP formalism and its Lagrangian theory.

Achiam, J., Held, D., Tamar, A. & Abbeel, P. (2017). Constrained Policy Optimization *ICML 2017*.
A trust-region method with constraint satisfaction guarantees during training, not merely at convergence.

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S. & Hester, T. (2021). Challenges of real-world reinforcement learning: definitions, benchmarks and analyses *Machine Learning 110*.
A systematic catalogue of the practical obstacles surveyed here, with benchmarks that isolate each.

Brunke, L., Greeff, M., Hall, A. W., Yuan, Z., Zhou, S., Panerati, J. & Schoellig, A. P. (2022). Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning *Annual Review of Control, Robotics, and Autonomous Systems 5*.
The modern survey of §14.5's territory, bridging control-theoretic safety and constrained RL.

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. & Bellemare, M. G. (2021). Deep Reinforcement Learning at the Edge of the Statistical Precipice *NeurIPS 34*.
Why small-sample RL comparisons mislead, and what to report instead — the basis for §14.6.

CHAPTER 15

Simulation & the Sim-to-Real Bridge

> *More mature solutions have often followed the zero-shot sim-to-real transfer scheme, which works particularly well for locomotion and navigation — the dynamics involved are relatively stable and easy to simulate.*
>
> **Chen Tang and colleagues** · UT Austin · University of Virginia · Sony AI
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Chapter 14 named two curses that pull against each other: real samples are expensive, and simulation is wrong. This chapter is about the resolution that produced nearly every L4 result in the field — train in a simulator you know to be wrong, but train across a distribution of wrong simulators wide enough that reality is one of them. We derive why that works, build the system identification that narrows the distribution, and construct the teacher–student scheme that lets a policy use information at training time it will never have at deployment. Ferris arrives.

FOUNDATION

Integrator stability bounds, the contact LCP with friction cones, domain randomization as distributional robustness, CMA-ES system identification, asymmetric actor–critic unbiasedness, and the teacher–student KL objective.

CONCEPTUAL

The randomization wall: watch a narrow policy's perfect peak collapse into a broad, lower, shippable competence curve.

PRACTICAL

rl-sim's parameterized rapier environments with randomization hooks, a rayon rollout farm, and CMA-ES parameter fitting.

AFTER THIS CHAPTER YOU CAN

- Explain why explicit integrators go unstable and how the step size bound relates to system stiffness
- Describe what a contact solver actually computes and why contact is where simulators diverge from reality
- Formulate domain randomization as a distributionally-robust objective and explain the peak/robustness trade
- Use system identification to fit simulator parameters to logged real data
- Explain the asymmetric actor–critic and why privileged information is safe for the critic but not the actor
- Construct a teacher–student pipeline and say why distillation beats training the student directly

15.1 What a physics engine actually does

A simulator advances rigid-body state by integrating Newton–Euler dynamics subject to constraints. Three components decide its fidelity, and each is a place where reality departs.

The integrator. Chapter 2 showed explicit Euler manufacturing energy on Pendle. The stability condition for an explicit scheme is roughly $\Delta t \leq 2/|\lambda_{\max}|$, where $\lambda_{\max}$ is the fastest eigenvalue of the linearized dynamics. Stiff systems — high-gain contacts, rigid transmissions — have enormous $\lambda_{\max}$, forcing tiny steps. Simulators therefore use semi-implicit or fully implicit schemes, which add artificial damping: stable, and systematically wrong in a direction that makes simulated robots feel more sluggish than real ones.

The constraint solver. Joints and contacts are constraints, resolved by solving a system each step. Most engines use iterative solvers with a fixed iteration budget, so constraints are satisfied *approximately*. Insufficient iterations produce visible artifacts: joints that stretch, stacked objects that sink.

Contact. This is the hard one. Non-penetration and Coulomb friction together form a complementarity problem: either the bodies are separated and the normal force is zero, or they touch and the force is non-negative — and the friction force lies inside a cone whose size depends on the normal force. Written as an LCP,

$$0 \leq \lambda_n \perp \left(J_n v^+ + b\right) \geq 0, \qquad \|\lambda_t\| \leq \mu \lambda_n,$$

where λ_n, λ_t are normal and tangential impulses. Exact solutions are expensive, so engines approximate — linearizing the friction cone into a pyramid, softening the complementarity with regularization parameters (`erp`, `cfm` and their relatives).

> **⚠ SOLVER PARAMETERS ARE NOT PHYSICS**
>
> Those regularization parameters have no physical counterpart. They exist to make the numerics converge, and their values change how a simulated robot behaves — how much a foot slips, how a grasp settles, whether a peg jams.
>
> This is the mechanism behind the curse of under-modelling. Locomotion and navigation transfer well because their dominant dynamics are rigid-body motion under gravity, which simulators get right. Contact-rich manipulation transfers poorly because its dominant dynamics are exactly the part simulators fudge. That is not a coincidence, and it explains Chapter 1's maturity ladder better than any algorithmic difference.

15.2 Meet Ferris

Ferris is a quadruped: four legs, three actuated joints each, twelve degrees of freedom. His state includes base position and orientation, base linear and angular velocity, joint positions and velocities — 36 numbers before any terrain sensing.

Ferris is the right robot for this chapter because he is the archetype of the field's clearest success. Quadruped locomotion reached commercial deployment through exactly the pipeline built here, and the reasons are worth naming: his dynamics are dominated by rigid-body motion, his contacts are intermittent and roughly point-like (much easier than the sustained sliding contact of manipulation), and a fall is recoverable rather than catastrophic.

His low-level control is a PD controller at each joint, following position targets:

$$\tau_j = K_p(q_j^{\text{des}} - q_j) - K_d\,\dot{q}_j.$$

The policy outputs q^{des} at 50 Hz; the PD loop runs at 1 kHz. That choice is itself a piece of design — Chapter 17 examines why almost everyone makes it — and it means the policy commands *positions*, not torques, with the PD gains absorbing a great deal of high-frequency dynamics the policy would otherwise have to learn.

15.3 Domain randomization

Here is the central idea, and it is counterintuitive at first: **do not try to build one accurate simulator. Build a distribution of inaccurate ones, and train a policy**

that works across all of them.

Formally, let ξ parameterize the simulator — masses, friction coefficients, motor gains, latencies, sensor noise — with $\xi \sim p(\xi)$. Train

$$\max_{\theta} \; \mathbb{E}_{\xi \sim p(\xi)} \left[J_{\xi}(\pi_{\theta}) \right].$$

If the real robot's parameters ξ^{*} lie within the support of p, a policy that performs well in expectation over p should perform acceptably at ξ^{*} — without ever having seen it. That is **zero-shot transfer**.

INTERACTIVE · `ch15-randomization-wall`

Trading peak for robustness

Success against the real robot's friction, for policies trained on progressively wider parameter distributions. The narrow policy peaks at nominal and fails elsewhere; the randomized one is merely good everywhere.

Run this simulation in the web edition: `/chapters/sim-to-real #ch15-randomization-wall`

The trade is visible in the widget and worth stating plainly: **you buy robustness by giving up peak performance**. A policy trained at one friction value is superb at exactly that value and useless a little away from it. A broadly randomized policy is merely good everywhere. Since you cannot measure the real value precisely, merely-good-everywhere is what ships.

WHY RANDOMIZATION WORKS, MATHEMATICALLY

Two framings, both useful.

Distributional robustness. Optimizing the expectation over $p(\xi)$ approximates optimizing the worst case over a set — and the connection is exact for particular choices: a soft-min over ξ with a KL penalty of radius ρ is equivalent to maximizing the expectation under an exponentially tilted distribution. Wide randomization is soft worst-case optimization.

Implicit system identification. With a recurrent policy, randomization does something more interesting than robustness. The policy cannot know ξ, but the recent history of states and actions is informative about it — a heavier robot accelerates less for the same command. A recurrent policy trained across ξ learns to *infer* the dynamics from its own recent experience and adapt.

This is the mechanism behind rapid motor adaptation, and it is why the same architecture can handle payload changes it never saw explicitly. The policy has learned to be a system identifier as a side effect of being asked to perform under uncertainty.

What to randomize on Ferris: link masses and inertias (±20%), friction coefficients (0.4–1.2), motor gains and strength (±30%), actuation latency (0–30 ms), sensor noise and bias, external pushes, and terrain geometry. **What not to randomize**: things you can measure precisely, like link lengths from CAD. Randomizing those wastes capacity on uncertainty you do not have.

15.4 System identification: narrow the distribution

Randomization is expensive in policy capacity — the wider $p(\xi)$, the more conservative the policy. So narrow it where you can, by fitting simulator parameters to real data.

Collect trajectories from the real robot, then solve

$$\xi^* = \arg\min_{\xi} \sum_{t} \left\| s_t^{\text{real}} - s_t^{\text{sim}}(\xi) \right\|^2$$

over the parameters you care about. The objective is non-convex, non-differentiable through the simulator, and low-dimensional — which makes it ideal for **CMA-ES**, an evolutionary strategy that maintains a Gaussian over parameters and adapts its covariance to the local landscape.

> **⚠ EXCITATION DECIDES WHAT YOU CAN IDENTIFY**
>
> A crucial and easily-missed point: you can only identify parameters that the data actually excites. Trajectories where the robot stands still tell you nothing about friction. Slow motion tells you nothing about inertia, because inertial forces scale with acceleration.
> Design identification experiments deliberately: fast motions for inertia, sliding motions under varying normal load for friction, step commands for motor gains and latency. If a parameter's gradient is flat across your dataset, no optimizer will recover it, and the fit will report a confident wrong answer.

The practical pipeline is: identify what you can, randomize around the residual uncertainty, and use the identification's own uncertainty to set the randomization width. That is more principled than choosing ±20% because it sounds reasonable.

RUST **rl-sim/src/randomize.rs**

```rust
use rand::{Rng, SeedableRng};
use rand_chacha::ChaCha8Rng;
use rayon::prelude::*;

/// Everything the simulator lets us vary -- one draw per episode.
#[derive(Clone, Debug, serde::Serialize, serde::Deserialize)]
```

```rust
pub struct DomainParams {
    pub mass_scale: [f64; 13],   // base + 12 links
    pub friction: f64,
    pub motor_kp: f64,
    pub motor_kd: f64,
    pub latency_steps: usize,
    pub terrain_roughness: f64,
}

#[derive(Clone)]
pub struct RandomizationRanges {
    pub mass_scale: (f64, f64),
    pub friction: (f64, f64),
    pub motor_kp: (f64, f64),
    pub latency_steps: (usize, usize),
    pub terrain_roughness: (f64, f64),
}

impl RandomizationRanges {
    pub fn sample<R: Rng>(&self, rng: &mut R) -> DomainParams {
        DomainParams {
            mass_scale: std::array::from_fn(|_| {
                rng.gen_range(self.mass_scale.0..=self.mass_scale.1)
            }),
            friction: rng.gen_range(self.friction.0..=self.friction.1),
            motor_kp: rng.gen_range(self.motor_kp.0..=self.motor_kp.1),
            motor_kd: rng.gen_range(0.5..=2.0),
            latency_steps: rng.gen_range(self.latency_steps.0..=self.latency_steps.1),
            terrain_roughness: rng.gen_range(
                self.terrain_roughness.0..=self.terrain_roughness.1
            ),
        }
    }
}

/// Collect rollouts across many randomized worlds in parallel.
/// Each worker derives its stream from the run seed, so results reproduce
/// exactly regardless of how many cores are available.
pub fn parallel_rollouts(
    policy: &Policy,
    ranges: &RandomizationRanges,
    n_envs: usize,
    horizon: usize,
    seed: u64,
) -> Vec<Episode> {
    (0..n_envs)
        .into_par_iter()
        .map(|i| {
            let mut rng = ChaCha8Rng::seed_from_u64(seed.wrapping_add(i as u64));
            let params = ranges.sample(&mut rng);
            let mut env = FerrisEnv::with_params(params);
            env.rollout(policy, horizon, &mut rng)
        })
        .collect()
}
```

Parameterized environments and a rayon rollout farm. Note the per-worker seeded RNG: reproducibility survives parallelism only if each worker's stream is derived deterministically from the run seed.

15.5 Privileged information: asymmetric actors and teacher–student

Here is the observation that unlocked rough-terrain locomotion.

In simulation you know everything: exact contact forces, the true friction coefficient, terrain height beneath every foot, the robot's precise velocity. On hardware you know almost none of it. The naive response is to train only on what the robot will have, which throws away information that would make training far easier.

Asymmetric actor–critic exploits the asymmetry. The critic sees privileged state s^{priv}; the actor sees only deployable observations o. This is legitimate: the critic is a training-time device that is discarded at deployment, and giving it privileged information reduces its variance without biasing the policy gradient — the gradient's correctness depends on the *actor's* conditioning, not the critic's.

Teacher–student distillation goes further, and is the ANYmal recipe:

1. **Train a teacher** with full privileged access — terrain map, friction, contact states. This policy learns quickly because the problem is nearly fully observed.
2. **Train a student** that sees only real sensors (joint encoders, IMU, and a short history), by supervised imitation of the teacher's actions:

$$\min_{\phi} \; \mathbb{E}_{\tau \sim \pi_\phi} \left[D_{\mathrm{KL}} \left(\pi_{\text{teacher}}(\cdot \mid s^{\text{priv}}) \,\middle\|\, \pi_\phi(\cdot \mid o_{t-H:t}) \right) \right].$$

1. **Deploy the student.**

> **WHY TWO STAGES BEAT ONE**
>
> Training the student directly from scratch is hard: it must simultaneously discover *what* to do and *how* to infer the hidden state that determines what to do. Those are two credit-assignment problems entangled.
>
> Distillation separates them. The teacher solves the control problem under full observation. The student solves an inference problem — recover enough of the hidden state from history to reproduce the teacher's actions — and that is supervised learning, which is far better behaved.
>
> Note the expectation in the objective: it is over the **student's** trajectory distribution, not the teacher's. That detail matters enormously, and Chapter 16 explains why — it is exactly DAgger, and training on the teacher's states instead produces the compounding error that plain behaviour cloning suffers.

15.6 Zero-shot or few-shot? A decision procedure

Tang's survey gives an empirical answer, and it is worth stating as a procedure.

Zero-shot transfer — train entirely in simulation, deploy without adaptation — works when the dominant dynamics simulate faithfully. That means locomotion, navigation, and flight. These are the L4 results.

Few-shot adaptation — fine-tune with a small amount of real data — is needed when simulation captures the structure but not the details: contact-rich assembly, deformable objects, anything where friction and compliance decide the outcome.

Real-world learning without simulation is necessary when there is no usable simulator at all: physical human–robot interaction, where the human is the unmodelable part. These results sit at L1–L2, and Chapter 16's offline methods are the main tool.

The decision rule: ask what fraction of your task's difficulty lies in phenomena your simulator represents faithfully. Locomotion is mostly rigid-body dynamics — simulate it. Cloth manipulation is mostly deformation — do not expect zero-shot to work.

15.7 Chapter bridge

The sim-to-real bridge is built. Randomize what you cannot measure, identify what you can, use privileged information at training time and distill it away for deployment, and choose zero-shot or few-shot by asking what your simulator actually gets right.

But some tasks have no usable simulator, and some have no writable reward. When the human is part of the environment, or when the task is easier to *show* than to *specify*, a different source of information is needed.

Chapter 16 turns to demonstrations. We prove why naive imitation fails in a way supervised learning does not — errors compound quadratically in the horizon — build the correction, and then develop offline RL: learning good policies from a fixed dataset without any environment interaction at all. It is the closest thing robot learning has to the pretraining paradigm that transformed the rest of machine learning.

Exercises

1 FOUNDATION •• **Integrator stability**

Derive the stability bound $\Delta t \leq 2/|\lambda|$ for explicit Euler on the linear system $\dot{x} = \lambda x$. Then estimate λ_max for a joint with PD gains Kp = 100, Kd = 5 and inertia 0.1, and state the maximum stable step size.

2 FOUNDATION • • • **The friction cone**

Write the contact complementarity conditions for a single point contact with Coulomb friction. Explain what is lost when the friction cone is linearized into a pyramid, and in which direction the approximation errs.

3 FOUNDATION • • • **Randomization as robust optimization**

Show that maximizing $E_\xi[J_\xi]$ under an exponentially tilted distribution corresponds to a soft-min over ξ with a KL penalty. What does the tilting temperature correspond to in the robust-optimization picture?

4 FOUNDATION • • **Why the critic may cheat**

Prove that the policy gradient remains unbiased when the critic is conditioned on privileged state unavailable to the actor. Then explain precisely why the same is not true if the ACTOR is conditioned on it.

5 CONCEPTUAL • • **The randomization frontier**

In the randomization widget, find the range that maximizes worst-case performance over friction $\in [0.7, 1.6]$. Compare against the range that maximizes performance at the nominal value. They will differ — explain which one you would deploy.

6 CONCEPTUAL • **Crank the step size**

Return to Chapter 2's integrator playground and find the Δt at which each integrator destabilizes. Then argue what the equivalent failure looks like for a quadruped's contact solver.

7 PRACTICAL • • • **System identification with CMA-ES**

Generate "real" trajectories from a Ferris simulation with hidden parameters, then recover them with CMA-ES from logged data. Investigate which parameters are identifiable from a standing trajectory versus a trotting one, and explain the difference in terms of excitation.

8 PRACTICAL • • • **Measure the transfer gap**

Train two Reacher policies — one at nominal parameters, one with ±25% randomization — then evaluate both across a grid of held-out "real" parameter values. Plot both surfaces and report the worst case for each. The narrow policy should win at nominal and lose everywhere else.

15.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§6.1 core issues in mental rehearsal, and §7.5 on the use of simulation in robot RL — the pre-deep-learning statement of this chapter's problem.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.3 simulator-usage axis, and §5 — the empirical basis for the zero-shot/few-shot decision procedure in §15.6.

FURTHER READING AND MODERN SOURCES

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W. & Abbeel, P. (2017). Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World *IROS 2017*.
The visual randomization result that named the technique.

Peng, X. B., Andrychowicz, M., Zaremba, W. & Abbeel, P. (2018). Sim-to-Real Transfer of Robotic Control with Dynamics Randomization *ICRA 2018*.
Dynamics randomization with recurrent policies — the implicit system identification of §15.3.

Hwangbo, J. et al. (2019). Learning agile and dynamic motor skills for legged robots *Science Robotics 4(26)*.
Learned actuator models — identifying the part of the robot the rigid-body simulator gets most wrong.

Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain *Science Robotics 5(47)*.
The teacher–student pipeline of §15.5, in the form this chapter reconstructs.

Kumar, A., Fu, Z., Pathak, D. & Malik, J. (2021). RMA: Rapid Motor Adaptation for Legged Robots *RSS 2021*.
Explicit latent estimation of the dynamics parameters — making implicit system identification a designed component.

Pinto, L., Andrychowicz, M., Welinder, P., Zaremba, W. & Abbeel, P. (2018). Asymmetric Actor Critic for Image-Based Robot Learning *RSS 2018*.
The privileged-critic construction and its unbiasedness argument.

Hansen, N. (2016). The CMA Evolution Strategy: A Tutorial *arXiv:1604.00772*.
CMA-ES, used here for system identification and in Chapter 17 for policy search over movement-primitive weights.

CHAPTER 16

Demonstrations, Imitation & Offline RL

> *Human demonstrations are effective for enabling real-world learning, particularly in manipulation tasks that are not prohibitively complex to demonstrate.*
>
> **Chen Tang and colleagues** · UT Austin · University of Virginia · Sony AI
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Some tasks have no usable simulator and no writable reward. But many of them are easy to demonstrate — a person can show the robot in thirty seconds what would take days to specify. This chapter starts with why naive imitation fails in a way supervised learning does not (errors compound quadratically in the horizon, and we prove it), builds the correction, and then develops offline RL: extracting good policies from a fixed dataset without any environment interaction at all. It is the closest thing robot learning has to the pretraining paradigm that transformed the rest of machine learning.

FOUNDATION

Behaviour cloning's compounding-error bound, DAgger's no-regret guarantee, MaxEnt IRL with its partition function, CQL's lower-bound property, and IQL's expectile regression.

CONCEPTUAL

A cloned policy drifting off the demonstrated lane inside a growing error cone — and the cone collapsing to a linear band the moment DAgger is switched on.

PRACTICAL

Browser teleoperation recording demonstration datasets, BC and DAgger in burn, IQL

on logged Reacher data, then SAC fine-tuning.

AFTER THIS CHAPTER YOU CAN

- Compare demonstration modalities and state what each one can and cannot capture
- Prove the T^2 compounding-error bound for behaviour cloning and explain the mechanism
- Explain DAgger's correction and why it is the same objective as Chapter 15's distillation
- Derive the MaxEnt IRL objective and explain what the partition function costs
- Explain distributional shift in offline RL and how CQL and IQL each avoid it
- Design an offline-to-online fine-tuning schedule that does not destroy the pretrained policy

16.1 Where data can come from

Reinforcement learning's default assumption is that the robot generates its own data by exploring. Chapter 14 priced that assumption honestly. Demonstrations offer a different bargain: a human supplies the data, and the robot never has to discover the behaviour from scratch.

The modalities differ in what they can express and what they cost.

Modality	How it works	Captures	Fails at
Teleoperation	Human drives the robot through its own actuators	Exact robot-executable actions	Contact forces are hard to feel through the interface
Kinesthetic teaching	Human physically guides the arm	Natural, dynamically feasible motion	Only works on backdrivable, gravity-compensated arms
Motion capture	Track a human performing the task	Rich, fast, natural motion	Correspondence problem — human bodies are not robot bodies
Video	Learn from ordinary recordings	Enormous scale, no special equipment	No action labels at all; must be inferred

DEMONSTRATION REPLACES REWARD DESIGN, NOT EXPLORATION

The deepest reason demonstrations help is not that they save samples. It is that they **sidestep Chapter 14's fourth curse entirely.**
You do not need to write a reward for "fold the towel neatly" — a concept

that has defeated every attempt at scalar specification. You show the robot three folds and the specification problem evaporates. This is why manipulation, the competency where reward design is hardest, is also the one where demonstrations dominate the successful results.

16.2 Behaviour cloning, and why it fails

The obvious approach: treat $D = \{(s_i, a_i)\}$ as a supervised learning dataset and fit $\pi_\phi(s) \approx a$. This is **behaviour cloning**, and it is genuinely the right first thing to try — it is simple, stable, and sometimes sufficient.

But it fails in a way that supervised learning does not, and the failure is structural rather than a matter of insufficient data.

Theorem 16.1 — Compounding error in behaviour cloning (Ross & Bagnell, 2010). Let π_ϕ have expected 0–1 loss at most ε under the demonstrator's state distribution d_{π^*}. Then over a horizon T, the expected cost of the cloned policy satisfies

$$J(\pi_\phi) \le J(\pi^*) + T^2\varepsilon,$$

and this bound is tight — there exist problems achieving it.

PROOF

The mechanism is a cascade. Suppose the policy first errs at step t. That single mistake moves the robot to a state slightly outside the demonstrated distribution — a state where the training data contains no examples, so the policy's behaviour there is undefined by the data and can be arbitrarily bad.

Formally: let p_t be the probability that the first mistake occurs at step t. Once off-distribution, the policy may incur cost up to $O(T - t)$ for the remaining horizon, since nothing constrains its behaviour there. Summing over when the first mistake happens:

$$J(\pi_\phi) - J(\pi^*) \le \sum_{t=1}^{T} p_t \cdot O(T - t) \le \varepsilon \sum_{t=1}^{T} O(T - t) = O(T^2\varepsilon).$$

The factor T counts opportunities to make a first mistake; the second factor T counts how long the consequences persist. **Tightness** is shown by a chain MDP where a single wrong action moves the agent to an absorbing state the demonstrator never visited, from which all subsequent cost is incurred. $$ ∎

INTERACTIVE · ch16-covariate-drift

The cloned robot drifts

A cloned policy drifting off the demonstrated lane inside a growing error cone, with the $O(\varepsilon T^2)$ and $O(\varepsilon T)$ bounds plotted. Switching on DAgger collapses the cone to a linear band.

Run this simulation in the web edition: `/chapters/imitation-and-offline-rl#ch16-covariate-drift`

THE DIFFERENCE FROM ORDINARY SUPERVISED LEARNING

In supervised learning, test inputs are drawn from the same distribution as training inputs, so error ε gives cost $O(T\varepsilon)$ over T predictions — linear. In sequential decision-making, **the policy's own mistakes change the distribution it will face next**. Training and test distributions are coupled through the policy, which is why the bound is quadratic. This is **covariate shift**, and no amount of additional data from the demonstrator's distribution removes it — the missing data is precisely the data the demonstrator never generated.

DAgger breaks the loop with an idea that is obvious in retrospect: query the expert *on the states the learner actually visits*. Roll out the current policy, ask the expert what it would have done at each visited state, add those labels to the dataset, and retrain. Iterate.

Because the training distribution now converges to the learner's own state distribution, the bound improves to $O(T\varepsilon)$ — linear, and DAgger comes with a no-regret guarantee. The cost is that the expert must remain available and answerable during training, which is fine for a scripted or privileged expert and expensive for a human.

YOU HAVE ALREADY SEEN DAGGER

Chapter 15's teacher–student distillation minimized a KL divergence under **the student's** trajectory distribution, and the text flagged that detail as mattering enormously.
This is why. Training the student on the teacher's states is behaviour cloning, with the T^2 bound. Training on the student's own states is DAgger, with the linear bound. The privileged teacher is always available to answer, so the expensive part of DAgger is free — which is a large part of why the teacher–student recipe works as well as it does on quadrupeds.

16.3 Inverse RL: recover the reward instead

Behaviour cloning copies actions. **Inverse reinforcement learning** aims higher: infer the *reward function* the demonstrator was optimizing, then optimize it yourself. The prize is generalization — a reward function transfers to new situations where a copied policy would flounder.

The problem is famously ill-posed: infinitely many reward functions explain any behaviour, including the degenerate $r \equiv 0$. **Maximum-entropy IRL** resolves the ambiguity with a principled tie-breaker: among all reward functions consistent with the demonstrations, prefer the one whose induced trajectory distribution has maximum entropy — commit to nothing beyond what the data forces.

The model is that trajectories are exponentially preferred by return:

$$p(\tau) = \frac{1}{Z} \exp\big(R_\psi(\tau)\big), \qquad Z = \int \exp\big(R_\psi(\tau)\big)\, d\tau,$$

and the reward parameters are fitted by maximum likelihood on the demonstrations. The gradient takes a clean and instructive form:

$$\nabla_\psi \mathcal{L} = \mathbb{E}_{\tau \sim \text{demos}} \left[\nabla_\psi R_\psi(\tau)\right] - \mathbb{E}_{\tau \sim p_\psi} \left[\nabla_\psi R_\psi(\tau)\right].$$

Raise the reward of what the expert did; lower the reward of what your current model predicts. The two terms balance exactly when the model's expected features match the demonstrator's.

> **⚠ THE PARTITION FUNCTION IS THE WHOLE DIFFICULTY**
>
> That second expectation requires sampling from p_ψ — which means solving the forward RL problem for the current reward estimate. Every gradient step contains a full RL problem.
>
> For small discrete MDPs, soft value iteration computes it exactly. For robots, it must be approximated, and the standard approach is adversarial: **GAIL** replaces the explicit reward with a discriminator distinguishing agent from expert trajectories, turning IRL into a GAN. That works, and inherits GAN training instability.
>
> The practical assessment: IRL is the right tool when you genuinely need the reward function — to transfer, to inspect, or to combine with other objectives. When you only need the behaviour, cloning plus fine-tuning is usually less trouble.

16.4 Offline RL: learning from a fixed dataset

Now the modern framing, and the one Tang's survey identifies as most promising for real-world robot learning.

Given a fixed dataset D of transitions — from demonstrations, from earlier policies, from scripted controllers, from anything — learn the best possible policy **with no further environment interaction**. No exploration, no resets, no risk.

The obstacle is specific and lethal. Off-policy algorithms like SAC evaluate $Q(s, a)$ at actions a proposed by the current policy. If the policy proposes an action absent from the dataset, Q is an extrapolation — and neural networks extrapolate confidently and wrongly. The policy then optimizes toward that erroneous high value, queries even further out of distribution, and the estimates diverge. Running SAC on a fixed dataset typically produces a policy far worse than the data it learned from.

THE FIX IS PESSIMISM, AND THERE ARE TWO FLAVOURS

CQL adds a term that pushes down Q at out-of-distribution actions while pushing it up at actions in the dataset:

$$\min_Q \; \alpha\Big(\mathbb{E}_{s\sim D, a\sim\mu}\left[Q(s,a)\right] - \mathbb{E}_{(s,a)\sim D}\left[Q(s,a)\right]\Big) + \mathcal{L}_{\mathrm{TD}}(Q).$$

The result provably lower-bounds the true Q, so the policy cannot be seduced by fictitious value. It is safe, and tuning α is fiddly — too much pessimism and the policy refuses to improve on the data at all.
IQL avoids querying out-of-distribution actions entirely. It never evaluates Q at unseen actions; instead it estimates the *upper expectile* of the value distribution over actions present in the data, using expectile regression with $\tau > 0.5$:

$$L(V) = \mathbb{E}_{(s,a)\sim D}\left[\left|\tau - \mathbb{1}[Q(s,a) - V(s) < 0]\right| \cdot \big(Q(s,a) - V(s)\big)^2\right],$$

then extracts a policy by advantage-weighted regression. Because it only ever touches in-distribution actions, it is markedly easier to tune — which is why it has become the default starting point.

RUST `rl-deep/src/offline/iql.rs`

```rust
use burn::prelude::*;

pub struct Iql<B: Backend> {
    q1: Critic<B>,
    q2: Critic<B>,
    q_target: (Critic<B>, Critic<B>),
    value: ValueNet<B>,
    actor: GaussianPolicy<B>,
    /// Expectile parameter. tau = 0.5 is plain regression; tau -> 1 approaches a max
```

```rust
    /// over the actions the dataset actually contains.
    expectile: f32,
    /// Temperature for advantage weighting; higher copies the best actions harder.
    beta: f32,
    gamma: f32,
}

impl<B: AutodiffBackend> Iql<B> {
    /// Asymmetric loss: underestimates are penalized more than overestimates,
    /// so V converges to an upper expectile of Q over in-dataset actions.
    fn expectile_loss(&self, diff: Tensor<B, 1>) -> Tensor<B, 1> {
        let weight = diff
            .clone()
            .lower_elem(0.0)
            .float()
            .mul_scalar(-1.0)
            .add_scalar(1.0)              // 1 where diff >= 0, 0 where diff < 0
            .mul_scalar(self.expectile)
            .add(
                diff.clone().lower_elem(0.0).float().mul_scalar(1.0 - self.expectile)
            );
        (weight * diff.powf_scalar(2.0)).mean()
    }

    pub fn value_loss(&self, batch: &Batch<B>) -> Tensor<B, 1> {
        // Q at DATASET actions only -- never at actions the policy proposes.
        let q1 = self.q_target.0.forward(batch.states.clone(), batch.actions.clone());
        let q2 = self.q_target.1.forward(batch.states.clone(), batch.actions.clone());
        let q = q1.min_pair(q2).detach();
        let v = self.value.forward(batch.states.clone());
        self.expectile_loss(q - v)
    }

    /// Advantage-weighted regression: imitate dataset actions, weighted by how
    /// much better than average they were.
    pub fn actor_loss(&self, batch: &Batch<B>) -> Tensor<B, 1> {
        let q1 = self.q_target.0.forward(batch.states.clone(), batch.actions.clone());
        let q2 = self.q_target.1.forward(batch.states.clone(), batch.actions.clone());
        let adv = (q1.min_pair(q2) - self.value.forward(batch.states.clone())).detach();

        // exp(beta.A), clamped so one lucky transition cannot dominate the batch.
        let weight = adv.mul_scalar(self.beta).exp().clamp_max(100.0);
        let log_prob = self.actor.log_prob(batch.states.clone(), batch.actions.clone());
        -(weight * log_prob).mean()
    }
}
```

IQL's three components in burn. Note that the actor is trained by weighted regression onto dataset actions — the policy never proposes an action the data cannot vouch for, which is exactly what makes it stable offline.

16.5 Offline-to-online: the deployment recipe

The pattern that Tang's survey finds behind the most capable manipulation systems is a three-stage pipeline, and it is worth stating as a recipe.

1. **Collect** a few hundred demonstrations by teleoperation.
2. **Pretrain offline** with IQL or CQL — a competent policy with zero risk, zero

resets, and no supervision.

3. **Fine-tune online** with a small amount of real interaction, which is where performance exceeds the demonstrator's.

Stage 3 has a well-documented failure mode. Switching directly to an online off-policy algorithm often causes an immediate performance collapse: the fresh online data is off-distribution relative to the offline dataset, value estimates lurch, and the pretrained policy is destroyed within a few hundred steps.

The mitigations are all forms of easing the transition: keep the offline data in the replay buffer and sample a mixture; anneal the pessimism coefficient down rather than dropping it; and use a conservative policy-update constraint for the first phase of online training. Getting this transition right is more consequential than the choice between CQL and IQL.

16.6 Chapter bridge

We have three ways to use data that the robot did not generate by exploring: clone the actions (cheap, and quadratically fragile), recover the reward (general, and expensive), or learn a policy offline from whatever transitions exist (the modern default). And we have the pipeline that combines them with online RL.

One question remains open, and it has been quietly present since Chapter 13. Everything here learns *what to do*. Nothing has examined **what the policy should output** — joint torques, position targets, task-space velocities, or entire parameterized movements.

Chapter 17 takes that up, and it turns out to matter as much as the learning algorithm. Kober's insight — that the representation of motor skills is a first-class design decision — survives intact into the deep era, and Tang's action-space taxonomy is its modern statement. We build dynamic movement primitives, and recreate the ball-in-a-cup experiment with both the 2013 pipeline and a modern one, honestly compared.

Exercises

1 FOUNDATION • • • **The T^2 bound**

Reproduce the proof of Theorem 16.1. Then construct the tight example explicitly: a chain MDP where one wrong action is unrecoverable, and verify the bound is achieved.

2 FOUNDATION • • • **DAgger's linear bound**

State DAgger's no-regret guarantee and explain which property of online learning it relies on. Why does training on the learner's own state distribution remove the extra factor of T?

3 FOUNDATION • • • **The MaxEnt IRL gradient**

Derive the MaxEnt IRL likelihood gradient, showing where the partition function produces the second expectation. Then explain why computing it exactly requires solving the forward RL problem.

4 FOUNDATION • • • **CQL lower-bounds Q**

Sketch the argument that CQL's learned Q lower-bounds the true Q for sufficiently large α. Then explain the failure mode when α is too large — what does the policy do?

5 CONCEPTUAL • **The cone and the fix**

In the covariate-shift widget, record the final deviation at horizons 40, 120 and 280 with DAgger off, then on. Confirm the quadratic-versus-linear growth empirically.

6 CONCEPTUAL • • **When is cloning enough?**

Find the combination of per-step error ε and horizon T at which behaviour cloning's final deviation stays acceptable. Then name a real robot task that falls inside that regime and one that does not.

7 PRACTICAL • • **Teleoperate and clone**

Record 50 Reacher demonstrations through the browser interface, train a BC policy, and measure success rate. Then deliberately start episodes off the demonstrated distribution and measure again — the gap is covariate shift, quantified.

8 PRACTICAL • • • **Offline to online, without the collapse**

Pretrain IQL on logged Reacher data, then fine-tune with SAC. First switch abruptly and record the performance collapse. Then retry with the offline data retained in the buffer and the pessimism annealed, and report how much of the collapse each mitigation removes.

16.7 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§5.1 prior knowledge through demonstration — the pre-deep-learning statement of why imitation matters for robots.

BASELINE Akinola, I. (n.d.). Reinforcement Learning for Robotics *Columbia University lecture notes*.
The demonstration-modality taxonomy of §16.1, and the covariate-shift framing of behaviour cloning.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§3.3 expert-usage axis and §5 real-world learning — the evidence for the offline-to-online recipe in §16.5.

FURTHER READING AND MODERN SOURCES

Ross, S. & Bagnell, J. A. (2010). Efficient Reductions for Imitation Learning *AISTATS 2010*.
Theorem 16.1 and its tightness.

Ross, S., Gordon, G. J. & Bagnell, J. A. (2011). A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning *AISTATS 2011*.
DAgger, with the linear-regret guarantee.

Ziebart, B. D., Maas, A., Bagnell, J. A. & Dey, A. K. (2008). Maximum Entropy Inverse Reinforcement Learning *AAAI 2008*.
The MaxEnt formulation and its partition-function gradient.

Ho, J. & Ermon, S. (2016). Generative Adversarial Imitation Learning *NeurIPS 29*.
GAIL — sidestepping the partition function by making the reward a discriminator.

Kumar, A., Zhou, A., Tucker, G. & Levine, S. (2020). Conservative Q-Learning for Offline Reinforcement Learning *NeurIPS 33*.
CQL and its lower-bound property.

Kostrikov, I., Nair, A. & Levine, S. (2021). Offline Reinforcement Learning with Implicit Q-Learning *arXiv:2110.06169*.
IQL: expectile regression that never queries out-of-distribution actions.

Levine, S., Kumar, A., Tucker, G. & Fu, J. (2020). Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems *arXiv:2005.01643*.
The reference survey for §16.4, including a careful treatment of distributional shift.

CHAPTER 17

Motor-Skill Policy Representations

> *A good policy representation reduces the search space dramatically. The choice of representation is often more important than the choice of learning algorithm.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On tractability through representation
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

Sixteen chapters have asked how a policy should learn. This one asks what it should output — and the answer matters at least as much. A policy commanding joint torques must discover gravity compensation from scratch; a policy commanding a movement primitive's goal has ten parameters and inherits stability by construction. Kober's insight that representation dominates algorithm survives intact into the deep era, and Tang's action-space taxonomy is its modern statement. We derive dynamic movement primitives, then recreate the ball-in-a-cup experiment with both the 2013 pipeline and a modern one, compared honestly.

FOUNDATION

Action-space levels with their trade-offs, the DMP canonical and transformation systems with a stability proof, CPG phase-locking, residual policy stability, and the semi-MDP option framework.

CONCEPTUAL

A movement primitive you reshape and re-target: drag the goal and watch the demonstrated style survive the deformation.

PRACTICAL

The DMP module in nalgebra, a CPG oscillator bank, a residual-RL wrapper, and ball-in-a-cup solved by both eras' pipelines.

AFTER THIS CHAPTER YOU CAN

- Classify an action space as low, mid, or high level and predict the consequences of each choice
- Derive the DMP equations and prove convergence to the goal regardless of the learned forcing term
- Explain how DMPs generalize across goals and time scalings, and what they cannot represent
- Describe CPGs and the phase-locking condition that produces coordinated gaits
- Formulate residual RL and explain why the classical prior bounds the learned policy's damage
- Write the semi-MDP Bellman equations for options and explain what temporal abstraction buys

17.1 What should the policy output?

Reacher must move its end-effector to a target. Which of these should the network produce?

- **Joint torques** $\tau \in \mathbb{R}^2$, at 1 kHz.
- **Joint position targets** q^{des}, tracked by a PD loop.
- **End-effector velocity** $\dot{p}$, converted through the Jacobian.
- **The goal of a movement primitive**, executed over a second.

All four are valid, and they produce radically different learning problems for identical hardware. Tang's survey organizes them into three levels.

Level	Output	Learns fast?	Expressive?	Safety
Low — joint torques	τ at 1 kHz	Slow — must discover gravity, inertia, damping	Maximal	Poor: a bad torque is immediately dangerous
Mid — position/velocity targets	q^{des}, $\dot{p}$ at 50 Hz	Faster — PD loop absorbs fast dynamics	High	Good: PD gains bound the force
High — primitives, skills	goal, duration, skill index	Fastest — search space is tiny	Limited to the primitive library	Best: primitives are pre-validated

DIFFICULTY MOVES; IT DOES NOT DISAPPEAR

Chapter 4 made this point about MDP formulation, and here is its concrete form. Choosing a higher action level makes the RL problem easier **only because a hand-written controller is absorbing the difficulty.** The composite system is exactly as complex as before; you have decided which part learns. That decision affects project outcomes more reliably than the choice between PPO and SAC. And it explains an empirical pattern in Chapter 1's maturity ladder: results at higher action levels reach real robots sooner, because there is less for the learned component to get catastrophically wrong.

17.2 Dynamic movement primitives

The most influential motor representation in robotics, and still the right answer for a surprising number of problems.

A DMP models a movement as a **spring–damper system pulled toward a goal, perturbed by a learned forcing term**. Two coupled systems.

The **canonical system** is a clock that decays monotonically to zero:

$$\tau \dot{x} = -\alpha_x x, \qquad x(0) = 1.$$

The **transformation system** is the movement itself:

$$\tau^2 \ddot{y} = \alpha_y \left(\beta_y (g - y) - \tau \dot{y}\right) + f(x),$$

where g is the goal and $f(x)$ is the learned forcing term, represented as a normalized weighted sum of Gaussian basis functions:

$$f(x) = \frac{\sum_i \psi_i(x)\, w_i}{\sum_i \psi_i(x)}\, x\,(g - y_0), \qquad \psi_i(x) = \exp\left(-h_i (x - c_i)^2\right).$$

Theorem 17.1 — DMPs converge to the goal for any weights. For $\alpha_y > 0$, $\beta_y = \alpha_y/4$ (critical damping), and any bounded weights w, the transformation system converges to $y = g$, $\dot{y} = 0$.

PROOF

Three steps.

The clock vanishes. $\tau \dot{x} = -\alpha_x x$ has solution $x(t) = e^{-\alpha_x t/\tau} \to 0$.

The forcing term vanishes with it. $f(x)$ carries an explicit factor of x, and the normalized basis sum is bounded. Hence $f(x(t)) \to 0$ as $t \to \infty$, regardless of the weights.

The remaining system is a stable spring. With $f = 0$, writing $e = g - y$ gives $\tau^2\ddot{e} + \alpha_y\tau\dot{e} + \alpha_y\beta_y e = 0$, a linear second-order system with characteristic roots $\lambda = \frac{-\alpha_y \pm \sqrt{\alpha_y^2 - 4\alpha_y\beta_y}}{2\tau}$. At $\beta_y = \alpha_y/4$ the discriminant is zero, giving a repeated negative root — critical damping, and the fastest non-oscillatory convergence. So $e \to 0$. $$ ■

WHY THIS THEOREM IS THE WHOLE POINT

The weights w are **learned** — fitted to a demonstration, or optimized by policy search. They could be anything, including nonsense produced by a half-trained optimizer.

The theorem says it does not matter: **the movement always terminates at the goal.** A learned component cannot make the robot fly off, because the learning only shapes the path, never the destination.

That is what a good motor representation does. It makes whole classes of failure *unrepresentable* rather than merely unlikely. Compare with a neural network outputting raw torques, where every safety property must be enforced externally.

INTERACTIVE · `ch17-dmp-sculptor`

A movement primitive you can reshape

A dynamic movement primitive: a spring toward the goal plus a learned forcing term. Move the goal and the demonstrated shape survives the deformation while the trajectory still terminates exactly at g.

Run this simulation in the web edition: `/chapters/motor-skill-representations#ch17-dmp-sculptor`

DMPs also generalize in two ways that fall out of the structure for free. Changing g re-targets the movement while preserving its shape — one demonstration covers a continuum of targets. Changing τ rescales time without changing the path — one demonstration covers a range of speeds.

Fitting from a demonstration is a linear least-squares problem, not an RL problem. Given a recorded trajectory $y_{\text{demo}}(t)$, invert the transformation system for the forcing term it must have produced:

$$f_{\text{target}}(t) = \tau^2\ddot{y}_{\text{demo}} - \alpha_y\left(\beta_y(g - y_{\text{demo}}) - \tau\dot{y}_{\text{demo}}\right),$$

then solve for w by locally weighted regression. One demonstration, one linear solve, done.

RUST **rl-core/src/primitives/dmp.rs**

```rust
use nalgebra::DVector;

pub struct Dmp {
    alpha_x: f64,
    alpha_y: f64,
    beta_y: f64,        // = alpha_y / 4 for critical damping
    centers: DVector<f64>,
    widths: DVector<f64>,
    pub weights: DVector<f64>,   // the learned parameters -- all of them
}

pub struct DmpState { pub y: f64, pub dy: f64, pub x: f64 }

impl Dmp {
    /// Normalized basis activation, gated by the canonical clock.
    fn forcing(&self, x: f64, y0: f64, goal: f64) -> f64 {
        let psi = (&self.centers - DVector::repeat(self.centers.len(), x))
            .map(|d| d * d)
            .component_mul(&self.widths)
            .map(|v| (-v).exp());
        let den = psi.sum();
        if den < 1e-10 { return 0.0; }
        // The factor x forces f -> 0 as the clock decays: this is what makes
        // Theorem 17.1 hold for ANY weights.
        (psi.dot(&self.weights) / den) * x * (goal - y0)
    }

    pub fn step(&self, s: &DmpState, y0: f64, goal: f64, tau: f64, dt: f64) -> DmpState {
        let f = self.forcing(s.x, y0, goal);
        let ddy = (self.alpha_y * (self.beta_y * (goal - s.y) - tau * s.dy) + f) / (tau *
    tau);
        DmpState {
            dy: s.dy + ddy * dt,
            y: s.y + (s.dy + ddy * dt) * dt,
            x: s.x - (self.alpha_x * s.x * dt) / tau,
        }
    }

    /// Fit weights from one demonstration by locally weighted regression.
    /// Note: a linear solve, not an RL problem.
    pub fn fit(&mut self, demo: &[f64], dt: f64, tau: f64) {
        let (y0, goal) = (demo[0], *demo.last().unwrap());
        let n = demo.len();

        let dy: Vec<f64> = (0..n).map(|i|
            if i == 0 { 0.0 } else { (demo[i] - demo[i - 1]) / dt }
        ).collect();
        let ddy: Vec<f64> = (0..n).map(|i|
            if i == 0 { 0.0 } else { (dy[i] - dy[i - 1]) / dt }
        ).collect();

        // Invert the transformation system for the forcing term it implies.
        let mut x = 1.0;
        let mut xs = Vec::with_capacity(n);
        let mut targets = Vec::with_capacity(n);
        for i in 0..n {
            targets.push(
                tau * tau * ddy[i]
                    - self.alpha_y * (self.beta_y * (goal - demo[i]) - tau * dy[i])
```

```
            );
            xs.push(x);
            x -= self.alpha_x * x * dt / tau;
        }

        // Weighted least squares, one basis function at a time.
        for k in 0..self.weights.len() {
            let (mut num, mut den) = (0.0, 0.0);
            for i in 0..n {
                let psi = (-self.widths[k] * (xs[i] - self.centers[k]).powi(2)).exp();
                let s = xs[i] * (goal - y0);
                num += psi * s * targets[i];
                den += psi * s * s;
            }
            self.weights[k] = if den.abs() > 1e-10 { num / den } else { 0.0 };
        }
    }
}
```

A DMP in nalgebra. The whole movement is ten to thirty numbers — which is why CMA-ES could optimize these on real hardware in 2013, long before anyone could train a network on a robot.

17.3 Central pattern generators

DMPs represent discrete point-to-point movements. Locomotion is **rhythmic**, and the natural representation is different: a bank of coupled oscillators, one per limb, whose relative phases define the gait.

Each oscillator has phase ϕ_i evolving as

$$\dot{\phi}_i = \omega + \sum_j K_{ij} \sin\left(\phi_j - \phi_i - \Phi_{ij}\right),$$

where ω is the base frequency and Φ_{ij} the desired phase offset between limbs i and j. The coupling drives the system toward those offsets and holds it there — a **phase-locked** solution exists and is stable when the coupling gains K_{ij} are large enough relative to frequency mismatch.

Gaits are then just phase-offset patterns: a trot sets diagonal pairs in phase and the two diagonals in antiphase; a bound pairs front and rear. Switching gait is changing Φ, not retraining.

For learned locomotion, CPGs appear in two roles: as an explicit action space where the policy modulates frequency and amplitude, or — more commonly now — as a **phase input** to an otherwise-free policy, giving it a clock to organize its behaviour around. Chapter 18 uses the second.

17.4 Residual RL: keep the controller, learn the correction

Chapter 13 built classical controllers that work well when the model is right. Chapter 15 showed the model is never quite right. **Residual RL** takes both facts

seriously:

$$a = \underbrace{\pi_{\text{classical}}(s)}_{\text{known, validated}} + \underbrace{\pi_\theta(s)}_{\text{learned correction}} .$$

The properties are exactly what a deployment engineer wants. At initialization $\pi_\theta \approx 0$, so the system starts at the classical controller's competence rather than at random flailing — no dangerous exploration phase. Bounding $\|\pi_\theta\| \le \Delta$ bounds how far the composite can deviate from validated behaviour, which converts an unbounded safety question into a tunable one. And the learned component only has to model what the classical controller *missed*, which is typically a far smaller function than the whole policy.

This is the pragmatic answer for many real deployments, and it is under-represented in the literature relative to how often it is the right choice.

17.5 Options: temporal abstraction

The highest action level is **skills**: policies that run for many steps and terminate on a condition. The formalism is the **option**, a triple $\langle \mathcal{I}, \pi_o, \beta \rangle$ — an initiation set, an internal policy, and a termination condition.

With options the problem becomes a **semi-MDP**, and the Bellman equation acquires a variable duration:

$$Q(s, o) = \mathbb{E}\left[r_{t+1} + \cdots + \gamma^{k-1} r_{t+k} + \gamma^k \max_{o'} Q(s_{t+k}, o') \right],$$

where k is the (random) number of steps the option ran.

The benefit is that a decision every 50 steps rather than every step shortens the effective horizon by that factor, which transforms credit assignment. The open question — which Tang's survey names as central — is **what skills should the robot learn at all?** Hand-specified skills work and require domain knowledge; discovering them automatically remains unsolved in general. Chapter 19 builds a hierarchical navigation system with hand-specified skills, and Chapter 21 returns to the discovery question.

17.6 Ball-in-a-cup: 2013 and now

Kober's survey closes with a case study: a robot swinging a ball on a string into a cup attached to its end-effector. It is a good benchmark — underactuated, contact-rich at the moment of capture, and genuinely hard to demonstrate perfectly.

The 2013 pipeline. Represent the movement as a DMP fitted from a kinesthetic demonstration. Optimize the ~30 DMP weights by episodic policy search — Kober's PoWER algorithm, an EM-based method exploiting the fact that the reward is a function of a low-dimensional parameter vector. Roughly 75 real rollouts suffice.

A modern pipeline. SAC from images with demonstrations seeded into the replay buffer, an action space of joint-position targets at 20 Hz, and domain randomization for transfer. Roughly 10^5–10^6 simulated steps plus fine-tuning.

> ⚠ **WHAT THE HONEST COMPARISON SHOWS**
>
> The 2013 pipeline uses **three orders of magnitude fewer real samples**. It also requires a demonstration, a hand-designed parameterization, and a reward computable from tracked ball position — and it produces a policy that does exactly one task and cannot adapt if the string length changes.
> The modern pipeline needs vastly more data, most of it simulated, and produces a policy that can work from raw images and tolerate variation. It also required someone to build a simulator good enough to transfer.
> Neither dominates. The correct reading is that **structure and data are substitutes**: DMPs encode strong priors and need little data; deep policies encode weak priors and need much. Choose based on which you actually have. Practitioners who reach for deep RL when a movement primitive would do are spending data to avoid thinking, and it is often the wrong trade.
> *(Implementation note: this book's code uses CMA-ES for the primitive-weight optimization rather than Kober's PoWER — a substitution made for library availability. PoWER is derived in the exercises; the comparison above is unaffected, as both are episodic policy-search methods over the same parameter vector.)*

17.7 Chapter bridge

Part III is complete. Chapter 13 opened the robot; Chapter 14 named the four structural difficulties; Chapter 15 built the sim-to-real bridge; Chapter 16 brought in data the robot did not generate; and this chapter established that what the policy outputs is a design decision on par with how it learns.

Part IV puts all of it to work, one competency at a time, following the taxonomy Tang's survey uses to organize the field's real results. Chapter 18 is locomotion — the flagship, the one that reached commercial deployment, and the place where every technique in Part III appears together. Ferris learns to walk, and we build the reward function, the terrain curriculum, and the teacher–student transfer that make it happen.

Exercises

1 FOUNDATION • • • **Prove DMP convergence**

Reproduce Theorem 17.1 in full, including the characteristic-root analysis showing $\beta = \alpha/4$ gives critical damping. Then determine what happens for $\beta > \alpha/4$ and $\beta < \alpha/4$, and say which you would choose for a robot that must not overshoot.

2 FOUNDATION • • **Fitting is linear**

Show that fitting DMP weights from a demonstration is a linear least-squares problem. Then explain why the same is not true of fitting a neural-network policy to the same demonstration, and what that costs.

3 FOUNDATION • • • **Phase locking**

For two coupled oscillators, derive the condition on coupling strength K under which a phase-locked solution exists and is stable, given a frequency mismatch $\Delta\omega$. Relate the result to how robustly a CPG maintains a gait under disturbance.

4 FOUNDATION • • **Semi-MDP Bellman**

Write the option-value Bellman equation and verify it reduces to the standard one when every option terminates after a single step. What replaces γ when option durations vary?

5 CONCEPTUAL • **Style survives re-targeting**

In the DMP sculptor, note the trajectory shape at goal g = 1.0, then move the goal to 2.0 and to −0.5. Describe precisely what is preserved and what deforms.

6 CONCEPTUAL • • **Basis capacity**

Reduce the basis count to 3 and raise it to 30 at fixed forcing amplitude. Identify the point at which the primitive can no longer represent the intended shape, and the point past which extra basis functions buy nothing.

7 PRACTICAL • • • **The action-space race**

Solve the same Reacher task with three action spaces — joint torques, joint position targets, and DMP goal parameters. Plot learning curves on a shared axis of environment steps. The ordering should be dramatic; explain it in terms of what each policy must discover.

8 PRACTICAL • • • **Ball-in-a-cup, both eras**

Implement ball-in-a-cup in rapier2d. Solve it once with CMA-ES over DMP weights initialized from a demonstration, and once with SAC on joint-position targets. Report real-equivalent sample counts, final success rates, and robustness to a 10% change in string length.

17.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§4 tractability through representation (§4.3 pre-structured policies), and §7 the ball-in-a-cup case study recreated in §17.6.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.

§3.2 the low/mid/high action-space taxonomy of §17.1, and §5 on skill discovery as an open problem.

FURTHER READING AND MODERN SOURCES

Ijspeert, A. J., Nakanishi, J., Hoffmann, H., Pastor, P. & Schaal, S. (2013). Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors *Neural Computation 25(2)*.

The definitive DMP reference — formulation, stability, and the generalization properties of §17.2.

Kober, J. & Peters, J. (2009). Policy Search for Motor Primitives in Robotics *NeurIPS 22*.

PoWER, and the ball-in-a-cup result — roughly 75 real rollouts.

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review *Neural Networks 21(4)*.

CPGs, phase coupling, and gait transitions.

Johannink, T., Bahl, S., Nair, A., Luo, J., Kumar, A., Loskyll, M., Ojea, J. A., Solowjow, E. & Levine, S. (2019). Residual Reinforcement Learning for Robot Control *ICRA 2019*.

The residual formulation of §17.4, with real-robot assembly results.

Sutton, R. S., Precup, D. & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning *Artificial Intelligence 112(1–2)*.

The options framework and the semi-MDP Bellman equations of §17.5.

Schaal, S. (2006). Dynamic Movement Primitives — A Framework for Motor Control in Humans and Humanoid Robotics *Adaptive Motion of Animals and Machines, Springer*.
The motor-control framing behind the representation.

Part IV

Competencies: RL on Real Robots

CHAPTER 18

Learning Locomotion

> *DRL has shown effectiveness in synthesizing robust and adaptive locomotion controllers for challenging conditions — the dynamics involved are relatively stable and easy to simulate, and dense shaped rewards simplify exploration.*
>
> **Chen Tang and colleagues** · UT Austin · University of Virginia · Sony AI
>
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Locomotion is the competency where robot RL genuinely won. Learned controllers ship in commercial quadrupeds, and rough-terrain walking that defeated three decades of model-based control fell to a fairly standard PPO pipeline. This chapter explains why locomotion and not something else, then builds the full recipe on Ferris: observation and action design, the reward function term by term with its units, terrain curricula, and the teacher–student transfer of Chapter 15. It closes on bipeds and flight, where the same recipe meets harder problems.

FOUNDATION

Gait phase formalism, reward-term algebra with dimensional analysis, curricula as distribution scheduling, and the privileged-information argument specialized to terrain.

CONCEPTUAL

The reward mixer: slide the weights and watch the gait that would emerge, with each term's physical units shown.

PRACTICAL

Ferris as a 12-DoF rapier3d quadruped with PD joints, PPO over rayon-parallel randomized environments, and a terrain generator with difficulty scheduling.

◎ AFTER THIS CHAPTER YOU CAN

- Explain why locomotion transferred to hardware before manipulation did
- Design a locomotion observation and action space, and justify each choice
- Write a locomotion reward term by term with correct units and defensible weights
- Construct a terrain curriculum and explain it as scheduling a distribution
- Assemble the full teacher–student pipeline and say what each stage contributes
- Explain what makes bipeds and quadrotors harder than quadrupeds

18.1 Why locomotion won first

Chapter 1's maturity ladder put locomotion at L4–L5 while manipulation sat at L2. That gap is not about effort or talent — it follows from four structural properties, and knowing them tells you when to expect RL to work on a new problem.

The dynamics simulate well. A walking robot is rigid bodies under gravity with intermittent point contacts. Physics engines are good at exactly that. Chapter 15 explained where they are bad — sustained sliding contact, deformation, precise friction — and locomotion mostly avoids those regimes.

Rewards are dense and writable. "Track this velocity command" gives a signal at every timestep. Compare with "assemble this device", where the reward is a single bit arriving minutes later.

Failure is cheap and recoverable. A quadruped that falls gets up. A manipulator that misjudges contact can destroy the workpiece, itself, or a person. That difference decides how freely you can explore.

The task is naturally continuous and repetitive. Millions of gait cycles are one long trajectory, which is exactly the regime where on-policy methods with massive parallelism shine.

THE TRANSFERABLE LESSON

Before starting an RL robot project, score it on those four axes. A task that simulates faithfully, admits a dense reward, tolerates failure, and repeats is a task where the Chapter 15 pipeline will probably work.
A task failing two or more — cloth manipulation, physical human interaction, one-shot assembly of an expensive part — needs a different plan, and the honest options are Chapter 16's demonstrations and offline data rather

than a better algorithm.

18.2 Ferris: observations and actions

Ferris has 12 actuated joints — hip abduction, hip flexion and knee per leg.

Observations (the deployable set, what the student policy sees):

Component	Dimension	Why
Base angular velocity	3	IMU gyro, directly measured
Projected gravity	3	Orientation without an absolute heading — invariant to yaw
Velocity command	3	(v_x, v_y, ω_z) — the task
Joint positions	12	Encoders
Joint velocities	12	Differentiated encoders
Previous action	12	Gives the policy access to its own recent output
Gait phase	2	$(\sin\phi, \cos\phi)$ — Chapter 17's CPG clock

That is 47 numbers, all available on real hardware at 50 Hz.

TWO DESIGN CHOICES WORTH EXPLAINING

Projected gravity instead of orientation. The gravity vector expressed in the body frame tells the policy which way is down without telling it which way is north. A policy conditioned on absolute yaw would learn direction-dependent behaviour and fail when the robot is placed facing a different way. Removing the information removes the failure mode.

Previous action in the observation. This lets the policy produce smooth outputs — it can see what it just did. Without it, policies tend to chatter, which is hard on gearboxes and is a common cause of a simulation-trained policy sounding alarming on real hardware.

Actions are joint position targets at 50 Hz, tracked by a 1 kHz PD loop with $K_p \approx 40$, $K_d \approx 1$. This is the mid-level choice from Chapter 17, and essentially the entire field has converged on it: the PD loop absorbs high-frequency dynamics, bounds the force the policy can command, and gives well-behaved gradients. Torque-level policies exist and learn much more slowly for no demonstrated benefit.

18.3 The reward function, term by term

Here is where locomotion papers actually differ from one another, and where most of the engineering time goes.

$$r = w_1 r_{\text{vel}} + w_2 r_{\text{yaw}} - w_3 c_{\text{effort}} - w_4 c_{\text{orient}} + w_5 r_{\text{air}} - w_6 c_{\text{slip}} - w_7 c_{\text{smooth}}$$

Term	Formula	Units	Purpose		
Velocity tracking	$\exp(-\	v_{xy} - v^*_{xy}\	^2/\sigma)$	dimensionless	The task
Yaw tracking	$\exp(-(\omega_z - \omega_z^*)^2/\sigma)$	dimensionless	Turn as commanded		
Effort	$\	\tau\	^2$	$N^2 \cdot m^2$	Do not slam the actuators
Orientation	$\	g_{\text{proj},xy}\	^2$	dimensionless	Stay level
Foot air time	$\sum_f (t_{\text{air},f} - 0.5)$	s	Take real steps, do not shuffle		
Foot slip	$\sum_f \	v_f\	^2 \cdot \mathbb{1}[\text{contact}]$	m^2/s^2	Walk, do not skate
Action smoothness	$\	a_t - a_{t-1}\	^2$	rad^2	Protect the gearboxes

THE WEIGHTS ARE UNIT CONVERSIONS THAT ALSO EXPRESS PREFERENCES

Those terms have genuinely different physical units. Summing them requires weights that convert everything to a common scale — so a weight is never purely a preference; it is a preference entangled with a unit conversion.

That is why locomotion reward tuning is the reproducibility problem of the subfield. Two papers with identical algorithms and different weights produce visibly different gaits, and a paper that does not publish its weights has not published its method. Chapter 14's warning about goal specification is not abstract here: it is most of the work.

INTERACTIVE · `ch18-reward-mixer`

Reward anatomy: the weights decide the gait

Slide the weight on each locomotion reward term — velocity tracking, torque penalty, foot air time, orientation, foot slip — and see the gait that would emerge, with each term's physical units shown.

Run this simulation in the web edition: `/chapters/learning-locomotion #ch18-reward-mixer`

Set the effort penalty high and Ferris creeps. Set the air-time reward high and he prances, wasting energy on exaggerated swings. Drop the orientation term

and he pitches forward until he falls. None of those is an algorithm failure — each is the optimizer correctly serving the objective it was given.

18.4 Gait as phase

A gait is a phase relationship between limbs. Give each foot f a phase $\phi_f \in [0, 2\pi)$ advancing at frequency ω, with a fixed offset Φ_f defining the gait:

$$\phi_f(t) = \omega t + \Phi_f \pmod{2\pi}.$$

A **trot** sets diagonal pairs together: $\Phi = (0, \pi, \pi, 0)$. A **bound** pairs front and rear: $(0, 0, \pi, \pi)$. Whether a foot should be in stance or swing is then a function of its phase.

Two ways to use this. **Prescribe it** — feed $(\sin\phi_f, \cos\phi_f)$ as observations and reward matching the intended contact schedule, giving fast, predictable learning of a specific gait. Or **let it emerge** — provide only a velocity-tracking reward and let the policy discover its own footfall pattern, which yields gaits that transition naturally with speed but takes longer and needs more careful reward shaping.

Modern systems typically prescribe a phase clock as an *input* while leaving the contact schedule unrewarded, getting organization without over-constraining the solution.

18.5 The terrain curriculum

Ferris cannot learn stairs by starting on stairs — he falls immediately, every episode, and there is no gradient. He must start on flat ground and progress.

Formally a curriculum schedules a distribution: at training stage k, terrain parameters are drawn from $p_k(\xi)$, and the schedule advances only when performance justifies it. The standard mechanism is per-environment difficulty with promotion and demotion: an environment whose robot traversed most of its terrain advances a level; one whose robot fell drops a level. Difficulty tracks competence automatically, and no global schedule needs tuning.

Terrain types progress roughly as: flat → rough (Perlin noise) → gentle slopes → steps → stairs → discrete obstacles → gaps.

> **CURRICULA ARE EXPLORATION, NOT CONVENIENCE**
>
> It is tempting to see a curriculum as a training speed-up. It is more fundamental than that: **it is how the agent gets any learning signal at all on a task where naive exploration never succeeds.**
> This is Chapter 14's dimensionality-of-exploration curse, addressed structurally. Random actions on stairs produce falls with no information. Random actions on flat ground produce a spread of outcomes with real

> gradient. The curriculum manufactures a path of solvable problems from one that is not.

18.6 The full pipeline

Assembling everything from Part III:

1. **Build the simulation** — Ferris in `rapier3d`, 12 PD joints, procedural terrain, 4096 parallel environments.
2. **Randomize** (Chapter 15) — masses ±20%, friction 0.4–1.2, motor gains ±30%, latency 0–30 ms, plus random pushes.
3. **Train a teacher with privileged access** — exact terrain heights under each foot, true friction, contact states. PPO (Chapter 10), a few billion simulated steps, hours on a GPU.
4. **Distill into a student** (Chapter 15) — the student sees only the 47 deployable observations plus a history window, trained by DAgger-style imitation on its own state distribution (Chapter 16).
5. **Deploy zero-shot.**

RUST `rl-envs/src/ferris/reward.rs`

```rust
/// One additive term of the locomotion reward, with its physical units
/// recorded so logs and ablations remain interpretable.
pub struct RewardTerm {
    pub name: &'static str,
    pub units: &'static str,
    pub weight: f64,
    pub eval: fn(&FerrisState, &Command) -> f64,
}

pub fn velocity_tracking(s: &FerrisState, cmd: &Command) -> f64 {
    let err = (s.base_lin_vel[0] - cmd.vx).powi(2) + (s.base_lin_vel[1] - cmd.vy).powi(2)
   ;
    (-err / 0.25).exp()          // dimensionless, in [0, 1]
}

pub fn foot_air_time(s: &FerrisState, _cmd: &Command) -> f64 {
    // Reward deliberate swings -- the standard cure for shuffling. Credited
    // only at touchdown, so it cannot be farmed by holding a foot up forever.
    s.feet
        .iter()
        .filter(|f| f.just_landed)
        .map(|f| f.air_time - 0.5)
        .sum::<f64>()
}

pub fn foot_slip(s: &FerrisState, _cmd: &Command) -> f64 {
    s.feet
        .iter()
        .filter(|f| f.in_contact)
        .map(|f| f.lin_vel.norm_squared())
```

```
        .sum::<f64>()
}

pub fn default_terms() -> Vec<RewardTerm> {
    vec![
        RewardTerm { name: "vel_track",  units: "-",      weight:  1.0,   eval:
   velocity_tracking },
        RewardTerm { name: "yaw_track",  units: "-",      weight:  0.5,   eval:
   yaw_tracking },
        RewardTerm { name: "effort",     units: "N2m2",   weight: -2.0e-4, eval:
   torque_squared },
        RewardTerm { name: "orient",     units: "-",      weight: -5.0,   eval:
   gravity_xy_squared },
        RewardTerm { name: "air_time",   units: "s",      weight:  1.0,   eval:
   foot_air_time },
        RewardTerm { name: "slip",       units: "m2/s2",  weight: -0.1,   eval: foot_slip
    },
        RewardTerm { name: "smooth",     units: "rad2",   weight: -0.01,  eval:
   action_rate_squared },
    ]
}

/// Sum the terms, and return the per-term breakdown for the dashboard.
/// Logging terms separately is what makes reward debugging tractable.
pub fn total(terms: &[RewardTerm], s: &FerrisState, cmd: &Command) -> (f64, Vec<f64>) {
    let parts: Vec<f64> = terms.iter().map(|t| t.weight * (t.eval)(s, cmd)).collect();
    (parts.iter().sum(), parts)
}
```

Composable reward terms with their units declared. Making units explicit in the type is not decoration — it is what lets the logging distinguish 'the weight is wrong' from 'the term is wrong'.

18.7 Bipeds and flight

Bipeds are harder for reasons that are structural rather than incidental. A quadruped in a trot always has two feet down and is statically stable much of the time; a biped in walking has long single-support phases and is never statically stable during them. Its support polygon is small — the area of one foot. Falling is expensive, since humanoids are top-heavy and fragile. And angular momentum management matters: arms and torso are not decoration, they are part of the control problem.

The recipe still applies, with more careful reward design (foot placement, centre-of-mass tracking, contact scheduling) and heavier reliance on reference motions — often motion-captured human walking — as either an imitation objective or a curriculum.

Quadrotors are the other end of the spectrum: fast, underactuated in translation, but with clean, well-modelled dynamics and no contact at all. That combination made them the setting for the most striking single result in the field — a learned policy beating human champions at drone racing on a physical track. The interesting detail is that its residual dynamics model was fitted from real flight data, which is Chapter 15's system identification doing the decisive work.

18.8 Chapter bridge

Locomotion's recipe is complete: mid-level actions on a PD loop, a dense multi-term reward whose weights are unit conversions, a terrain curriculum that manufactures a learnable gradient, randomization for transfer, and teacher–student distillation to reach a deployable observation set.

What Ferris does not have is anywhere to go. He tracks a velocity command; someone else decides what the command should be.

Chapter 19 supplies that. Navigation is the perception-heavy competency — partial observability in its natural habitat, where the state is a map the robot has never seen. Rusty graduates from Chapter 4's gridworld to continuous lidar-based navigation, and we confront the architectural question the survey finds genuinely unsettled: how much of the stack should be learned at all?

Exercises

1 FOUNDATION •• **Dimensional audit**

For each reward term in §18.3, verify the units and determine what the weight's units must be for the sum to be dimensionally consistent. Then explain why a paper reporting weights without units is not fully reproducible.

2 FOUNDATION •• **Gait phases**

Write phase offsets for trot, pace, bound and gallop. For each, state the number of feet in contact over a cycle and relate it to static stability. Which gaits require dynamic balance throughout?

3 FOUNDATION •• **Curriculum as scheduling**

Formalize a terrain curriculum as a sequence of distributions p_k(ξ) with a promotion rule. Then state a condition under which the curriculum could stall — the robot neither promoting nor demoting — and propose a fix.

4 FOUNDATION ••• **Why privileged terrain helps**

Argue why a teacher with exact terrain heights learns faster than a student with only proprioception. Frame it as the difference between a nearly-fully-observed MDP and a POMDP, using Chapter 4's belief formalism.

5 CONCEPTUAL • **Break the gait four ways**

In the reward mixer, produce four distinct failures: standing still, creeping, prancing, and falling. Record the weight settings for each and name the term responsible.

6 CONCEPTUAL •• **Find the trotting region**

Identify the region of weight space that yields a plausible trot. Is it a broad basin or a narrow ridge? Relate your answer to how much tuning effort locomotion papers actually report.

7 PRACTICAL ••• **Ferris walks**

Build Ferris in rapier3d with PD joints and train PPO on flat ground with velocity tracking only. Then add terms one at a time and record the gait change from each. Report which single term most improved the result.

8 PRACTICAL ••• **Curriculum ablation**

Train two policies on stairs: one with the terrain curriculum, one starting directly on the hardest terrain. Report the sample count each needs to reach a 50% traversal rate — the second may never get there, which is itself the finding.

18.9 References

BASELINE REFERENCES

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§4.1 in full — quadruped, biped and quadrotor locomotion, with the trend analysis behind §18.1.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
The pre-deep-learning locomotion lineage, including Kohl & Stone's policy-gradient gait optimization.

FURTHER READING AND MODERN SOURCES

Hwangbo, J. et al. (2019). Learning agile and dynamic motor skills for legged robots *Science Robotics 4(26)*.
Where the modern pipeline begins, including the learned actuator model.

Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain *Science Robotics 5(47)*.
The teacher–student pipeline assembled in §18.6.

Miki, T., Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2022). Learning robust perceptive locomotion for quadrupedal robots in the wild *Science Robotics 7(62)*.
Adding exteroception while keeping proprioceptive fallback — L4 in the wild.

Rudin, N., Hoeller, D., Reist, P. & Hutter, M. (2022). Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning *CoRL 2021*.
The parallel-simulation regime and the per-environment terrain curriculum of §18.5.

Kumar, A., Fu, Z., Pathak, D. & Malik, J. (2021). RMA: Rapid Motor Adaptation for Legged Robots *RSS 2021*.
Explicit estimation of dynamics parameters from proprioceptive history.

Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V. & Scaramuzza, D. (2023). Champion-level drone racing using deep reinforcement learning *Nature 620*.
The flight result of §18.7, with residual dynamics identified from real flight data.

Peng, X. B., Abbeel, P., Levine, S. & van de Panne, M. (2018). DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills *ACM Transactions on Graphics 37(4)*.
Reference-motion imitation — the technique bipedal work leans on most heavily.

CHAPTER 19

Learning Navigation & Mobile Manipulation

> *The effectiveness of end-to-end versus hybrid modular solutions varies by problem. Neither approach is universally superior.*
>
> **Chen Tang and colleagues** · UT Austin · University of Virginia · Sony AI
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Navigation is where partial observability stops being a footnote. A robot in an unmapped building genuinely does not know where it is, and Chapter 4's honest formalism — the POMDP — is unavoidable rather than optional. This chapter derives the belief-MDP response, shows how a recurrent policy learns to track a belief without being told to, and confronts the architectural question the survey finds genuinely open: how much of the navigation stack should be learned? Then mobile manipulation, where a base and an arm must coordinate, and long-horizon tasks force the question of what a skill even is.

FOUNDATION

Belief MDPs for navigation, recurrent policies as approximate filters, semi-MDP skill composition, and weighted-pseudoinverse redundancy resolution for whole-body control.

CONCEPTUAL

The pipeline switcher: the same task under three architectures, with the success/interpretability/data trade made explicit as the environment gets less structured.

PRACTICAL

Lidar Rusty in rapier2d, a recurrent GRU policy in burn, a hierarchical skill selector,

and a mobile-Reacher whole-body fetch task.

◎ AFTER THIS CHAPTER YOU CAN

- Construct the belief MDP for navigation and explain why it restores the Markov property
- Explain how a recurrent policy approximates a belief tracker, and how to probe what it stores
- Compare end-to-end, hybrid and modular architectures on the axes that actually differ
- Resolve kinematic redundancy for a mobile manipulator with a weighted pseudoinverse
- Formulate a long-horizon task as a semi-MDP over skills and state the credit-assignment benefit
- Explain why "which skills should the robot learn?" is the field's open question

19.1 Rusty graduates

Chapter 4's Rusty lived in a 12×9 grid and always knew which cell he occupied. The real Rusty has a 2-D lidar returning 64 noisy range measurements, wheel odometry that drifts, and a building he has never seen.

The observation is now $o_t = (\text{scan}_t, \text{goal vector}_t, \text{velocity}_t)$ — about 70 numbers, none of which is the state. The state would include his true pose and the complete geometry of the building.

This is the POMDP that Chapter 4 introduced and postponed. Here it is unavoidable, and it is worth seeing what goes wrong if you ignore it: a policy treating the current scan as state cannot distinguish two identical-looking corridors, cannot remember that it already tried the left branch, and will oscillate at symmetric junctions. Those are not tuning problems; they are consequences of a state representation that is not Markov.

19.2 Belief, and how a network learns to track it

Chapter 4 gave the repair. Maintain a belief $b_t(s) = \mathbb{P}(S_t = s \mid o_{1:t}, a_{1:t-1})$, updated by Bayes' rule, and the belief *is* Markov: b_{t+1} depends only on b_t, a_t and o_{t+1}. Planning over beliefs is therefore an ordinary MDP — over a continuous, high-dimensional space, which is why exact solutions are out of reach.

The practical answer is to let the policy learn its own belief representation. A recurrent policy carries a hidden state h_t updated as $h_t = \text{GRU}(h_{t-1}, o_t, a_{t-1})$,

and acts as $\pi(a_t \mid h_t)$.

> **THE RECURRENT STATE IS A LEARNED, TASK-SPECIFIC FILTER**
>
> Nothing tells the GRU to estimate pose. It is trained only to maximize return. But return depends on reaching goals, reaching goals requires knowing roughly where you are, so **the hidden state learns to encode whatever about the history is useful** — and no more.
>
> That last clause is the advantage over a hand-built filter. An EKF estimates full pose with a covariance, at a cost. A recurrent policy learns to track only the aspects of location that matter for the decisions it faces, which is typically far less information.
>
> You can check this. Freeze the policy, collect trajectories, and train a linear probe from h_t to true pose. High probe accuracy means the network is genuinely localizing; low accuracy with good performance means it found a shortcut that does not require localization — which is worth knowing before deployment, because such shortcuts often depend on training-environment regularities.

Two implementation details decide whether recurrent PPO works. **Sequence-based batching**: hidden states must be carried across a trajectory, so batches are sequences, not shuffled transitions. And **burn-in**: when replaying a stored sequence, run the recurrent state forward over some initial steps without computing loss, so the hidden state is warm before gradients are taken.

19.3 The architecture question

Here the survey's finding is unusually direct: neither end-to-end nor modular wins in general. That is a real result rather than a failure to decide, and the reason is that they fail differently.

> **INTERACTIVE · `ch19-pipeline-switcher`**
>
> **End-to-end or modular?**
>
> The same navigation task under three architectures as the environment becomes less structured, with success, interpretability, tuning burden and data requirement tracked for each.
>
> Run this simulation in the web edition: `/chapters/navigation-and-mobile-manipulation#ch19-pipeline-switcher`

The classical modular stack — SLAM, global planner, local planner, tracking controller — is inspectable, independently testable, and needs no training data.

In mapped, static, structured environments it is excellent, and reaching for RL there is a mistake.

End-to-end policies handle situations nobody wrote a cost function for. They also fail without explanation, need substantial data, and cannot be debugged stage by stage. When an end-to-end policy drives into a glass door, the diagnosis is "the network did that".

The hybrid — classical mapping and global routing, learned local navigation — dominates deployed systems, and the reason is visible in the widget: hand-tuned local planners degrade fastest as environments become crowded and dynamic, because their cost functions encode assumptions that stop holding. That is precisely the layer worth learning, and precisely the layer where a mistake is cheapest.

Social navigation deserves a note. Moving among people is not obstacle avoidance with moving obstacles: humans react to the robot, so the environment is a multi-agent system where the robot's action changes others' behaviour. That makes simulation hard for exactly the reason Chapter 1 gave — humans are the unmodelable part — and it is why social navigation sits lower on the maturity ladder. Chapter 21 treats it properly under human–robot interaction.

19.4 Mobile manipulation: base plus arm

Attach Reacher to Rusty and you have a mobile manipulator with a genuinely new problem: the base and the arm can both contribute to end-effector motion, so the system is **kinematically redundant**.

With combined configuration $q = (q_{\text{base}}, q_{\text{arm}})$ and a task Jacobian $J(q)$ mapping to end-effector velocity, the task $\dot{p} = J\dot{q}$ has infinitely many solutions when $\dim(q) > \dim(p)$. The standard resolution is the weighted pseudoinverse:

$$\dot{q} = W^{-1}J^{\top}\left(JW^{-1}J^{\top}\right)^{-1}\dot{p} + \left(I - J^{\dagger}J\right)\dot{q}_0,$$

where W weights joint motion — heavy for the base, light for the arm, so small corrections use the arm and large repositioning uses the base — and the second term projects a secondary objective (joint-limit avoidance, manipulability maximization, keeping clear of a person) into the null space, where it cannot disturb the task.

> **WHERE LEARNING HELPS HERE**
>
> The pseudoinverse solves the *instantaneous* redundancy question well. What it cannot answer is the strategic one: should the robot drive closer before reaching, or reach from here? That depends on what comes next, which is exactly a sequential decision problem.
> The productive division is therefore to learn the high-level allocation —

how much of the task each subsystem should take, and when to reposition — while keeping the instantaneous resolution analytic. This is Chapter 17's residual and hierarchical framing applied to whole-body control.

19.5 Long horizons and the skill question

"Fetch the mug from the kitchen" decomposes into navigate, locate, approach, grasp, lift, navigate back, place. Minutes of operation, thousands of timesteps, and a reward that arrives once at the end. Flat RL has no chance: random exploration will not stumble on that sequence in any number of episodes.

Chapter 17's options framework is the formal response. With skills as temporally extended actions, the high-level policy decides every few seconds rather than every 20 ms, and the effective horizon shortens by two orders of magnitude. Credit assignment becomes tractable because there are only a handful of decisions to assign credit among.

Three ways to compose skills, each with a real cost. **Hierarchical RL** learns the selector by RL — general, and unstable, since the high level's environment changes as the low level improves. **Planning over skills** uses a symbolic planner with learned skills as operators — reliable, and requires hand-written preconditions and effects. **End-to-end with skill priors** trains one policy conditioned on a skill embedding — flexible, and data-hungry.

⚠ THE OPEN QUESTION, STATED PLAINLY

Tang's survey identifies this as central and unresolved: **what skills should the robot learn?**
Hand-specified skills work — most deployed systems use them — but require exactly the domain knowledge RL was supposed to remove. Automatic discovery, via goal-conditioned or unsupervised RL, produces skills that look reasonable in benchmarks and rarely correspond to the decomposition a task actually needs.
This is not a detail awaiting an engineering fix. It is the main obstacle between the competencies of Part IV and a general-purpose robot, and it is worth watching as the field's most consequential open problem.

RUST `rl-deep/src/recurrent/gru_policy.rs`

```rust
use burn::prelude::*;
use burn::nn::gru::{Gru, GruConfig};

pub struct RecurrentPolicy<B: Backend> {
    encoder: Mlp<B>,        // lidar scan -> compact embedding
    gru: Gru<B>,            // the learned belief tracker
    actor_head: Mlp<B>,
```

```
    critic_head: Mlp<B>,    // may see privileged state during training (Ch 15)
    /// Diagnostic head predicting true pose from the hidden state. Not used for
    /// control -- it exists to answer "is this thing actually localizing?"
    probe_head: Option<Mlp<B>>,
}

pub struct PolicyStep<B: Backend> {
    pub action_mean: Tensor<B, 2>,
    pub action_log_std: Tensor<B, 2>,
    pub value: Tensor<B, 1>,
    pub hidden: Tensor<B, 2>,
}

impl<B: Backend> RecurrentPolicy<B> {
    pub fn step(
        &self,
        scan: Tensor<B, 2>,        // [batch, n_beams]
        goal_vec: Tensor<B, 2>,    // [batch, 2] -- goal in the robot frame
        prev_action: Tensor<B, 2>,
        hidden: Tensor<B, 2>,
    ) -> PolicyStep<B> {
        let obs = Tensor::cat(vec![scan, goal_vec, prev_action], 1);
        let embedded = self.encoder.forward(obs);

        // One GRU step: h_t = GRU(h_{t-1}, o_t). The hidden state carries
        // everything the policy has decided is worth remembering.
        let hidden = self.gru.forward(embedded.unsqueeze_dim(1), Some(hidden))
            .squeeze(1);

        let actor_out = self.actor_head.forward(hidden.clone());
        let (mean, log_std) = actor_out.split_with_sizes(vec![2, 2], 1)
            .into_iter()
            .collect_tuple()
            .unwrap();

        PolicyStep {
            action_mean: mean,
            action_log_std: log_std.clamp(-5.0, 2.0),
            value: self.critic_head.forward(hidden.clone()).squeeze(1),
            hidden,
        }
    }

    /// Train a linear probe from the hidden state to true pose, on frozen
    /// trajectories. High accuracy => the network learned to localize.
    pub fn probe_localization(&self, hidden: Tensor<B, 2>) -> Option<Tensor<B, 2>> {
        self.probe_head.as_ref().map(|p| p.forward(hidden))
    }
}
```

A recurrent policy step in burn. The hidden state is the learned belief; the probe head is optional and exists so you can check what the network actually tracks — which is the difference between trusting it and hoping.

19.6 Chapter bridge

Navigation forced partial observability into the open and answered it with recurrence rather than explicit filtering. Mobile manipulation added redundancy

— a resource, resolved analytically at the instant and strategically by learning. Long horizons pushed us to skills, and left the open question of where skills come from.

Chapter 20 takes on the competency that has resisted hardest. Manipulation is where simulators are least trustworthy, where object diversity defeats generalization, and where the field's maturity ladder tops out below the real-world tiers. It is also where every technique in this book converges: grasp scoring with the value methods of Chapter 9, contact-rich insertion with the continuous control of Chapter 11 and the impedance action spaces of Chapter 13, demonstrations from Chapter 16, and the randomization of Chapter 15 pushed to its limit.

Exercises

1 FOUNDATION •• **The belief update**

Write the Bayes filter for lidar-based localization on a known map, identifying the prediction and correction terms. Then explain why the belief is Markov even though the raw observation is not.

2 FOUNDATION •• **Symmetric corridors**

Construct an environment where two distinct states produce identical observations, and show that any memoryless policy is provably suboptimal there. Then show that a policy with one bit of memory can be optimal.

3 FOUNDATION ••• **Weighted redundancy resolution**

Derive the weighted pseudoinverse solution and verify that the null-space term does not disturb the task velocity. Then determine the weighting W that makes a mobile manipulator prefer arm motion over base motion, and quantify "prefer".

4 FOUNDATION •• **Horizon reduction**

A task takes 6000 timesteps at 50 Hz. Using skills of average duration 3 seconds, compute the effective horizon for the high-level policy. Then explain the effect on TD credit assignment using Chapter 7's n-step analysis.

5 CONCEPTUAL •• **Find the crossover**

In the pipeline switcher, find the unstructuredness at which the hybrid architecture overtakes the modular one, and where end-to-end overtakes hybrid (if it does). Relate both crossovers to what each stack assumes about the world.

6 CONCEPTUAL •• **Price the interpretability**

At 60% unstructuredness, note the success and interpretability of each architecture. Then argue which you would deploy in a hospital, and which in a warehouse after hours — the answers should differ.

7 PRACTICAL ••• **Probe the hidden state**

Train a recurrent policy on lidar navigation, then freeze it and fit a linear probe from the GRU hidden state to true pose. Report probe accuracy. Then retrain in an environment with a distinctive landmark and see whether the network stops localizing and starts recognizing.

8 PRACTICAL ••• **Whole-body fetch**

Build mobile-Reacher and implement a fetch task with a learned high-level allocator over analytic whole-body control. Compare against a fixed policy that always drives to a nominal standoff distance before reaching, and report where learning actually helped.

19.7 References

BASELINE REFERENCES

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§4.2 navigation (wheeled, legged, aerial) and §4.4 mobile manipulation, including the architecture finding of §19.3 and the skill question of §19.5.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§1.3 on partial observability as the normal condition for robots, and §5.2 prior knowledge through task structuring.

FURTHER READING AND MODERN SOURCES

Kaelbling, L. P., Littman, M. L. & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains *Artificial Intelligence 101(1–2)*.
The belief-MDP construction that §19.2 approximates with recurrence.

Thrun, S., Burgard, W. & Fox, D. (2005). Probabilistic Robotics *MIT Press*.
The classical localization and SLAM stack that the modular architecture is built from.

Hausknecht, M. & Stone, P. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs *AAAI Fall Symposium*.
Recurrent policies for POMDPs, and the sequence-batching details of §19.2.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R. & Dabney, W. (2019). Recurrent Experience Replay in Distributed Reinforcement Learning *ICLR 2019*.
The burn-in technique for stored recurrent states.

Xiao, X., Liu, B., Warnell, G. & Stone, P. (2022). Motion planning and control for mobile robot navigation using machine learning: a survey *Autonomous Robots 46*.
The learned-versus-classical navigation landscape in detail.

Sutton, R. S., Precup, D. & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction *Artificial Intelligence 112(1–2)*.
The options framework underlying §19.5.

Siciliano, B., Sciavicco, L., Villani, L. & Oriolo, G. (2009). Robotics: Modelling, Planning and Control *Springer*.
Redundancy resolution and null-space projection, as used in §19.4.

CHAPTER 20

Learning Manipulation

> *RL is beginning to achieve real-world success in manipulation, but the diversity of objects and the difficulty of modelling contact keep general-purpose solutions below the maturity reached in locomotion.*
>
> **Chen Tang and colleagues** · UT Austin · University of Virginia · Sony AI
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Manipulation is the competency that has resisted hardest, and understanding why is more instructive than any single result. Contact is where simulators are least trustworthy, object diversity defeats the generalization that made locomotion work, and failure is expensive. This chapter works through the subproblems in order of how well they have gone — grasping, then contact-rich assembly, then in-hand dexterity, then general pick-and-place — and shows where each technique from earlier chapters lands. It closes on evaluation, because manipulation is where success rates are most often reported in ways that will not survive scrutiny.

FOUNDATION

Grasp wrench space and force closure, the Ferrari–Canny metric, contact-mode combinatorics, impedance control as an action space, and Wilson intervals for small-sample success rates.

CONCEPTUAL

Force closure you can break by dragging contacts — the analytic criterion that learned grasp scorers approximate from pixels.

PRACTICAL

A parallel-jaw gripper in rapier3d, grasp-selection DQN paying off Chapter 9, SAC insertion with impedance actions, and demo-boosted training from browser teleopera-

tion.

AFTER THIS CHAPTER YOU CAN

- Explain force closure and compute grasp quality from contact geometry
- Explain why contact-mode combinatorics make planning through contact intractable
- Formulate impedance control as an action space and explain why it beats position control under uncertainty
- Describe the in-hand dexterity recipe and what it cost
- Say precisely why general pick-and-place remains below the real-world tiers
- Report a manipulation success rate honestly, with an interval that reflects the sample size

20.1 Why manipulation is the hard one

Chapter 18 listed four properties that made locomotion tractable. Manipulation fails on all four, and the symmetry is worth spelling out.

The dynamics do not simulate well. Manipulation is *made of* the contact interactions that Chapter 15 identified as simulators' weakest point — sustained sliding, friction that decides outcomes, compliance, deformation. A locomotion policy needs contact to be roughly right; a manipulation policy needs it to be right in detail.

Rewards are sparse and hard to write. "Did the assembly succeed?" is one bit, arriving after a long sequence. Dense shaping requires knowing what intermediate progress looks like, which for contact-rich tasks is often exactly what you do not know.

Failure is expensive. A dropped object may break; a mis-inserted peg can damage the workpiece or the robot; a manipulator near a person is a safety case.

Objects are diverse. A quadruped's world is terrain — endless variation over one type of thing. A manipulator's world contains objects it has never seen, whose mass, friction and compliance it cannot observe.

READ THE MATURITY LADDER AS A DIAGNOSIS

Chapter 1's ladder showed contact-rich assembly reaching L3 while general pick-and-place stalls at L2. That ordering looks backwards — surely inserting a connector is harder than picking things up?
It is not, under the four criteria. Assembly is usually *structured*: known

parts, fixtured, in a controlled cell. Dense rewards can be designed a priori because the goal geometry is known. General pick-and-place is *unstructured*: novel objects, arbitrary poses, cluttered scenes.
The lesson generalizes. **Structure, not intrinsic task difficulty, predicts whether RL will transfer.**

20.2 Grasping: the analytic criterion and its learned approximation

Grasping has the cleanest theory in manipulation, and it is worth knowing even though modern systems learn rather than compute it.

A contact exerting force f at position p produces a **wrench** — force plus torque — of $w = (f, p \times f)$. With Coulomb friction, the forces a contact can exert lie in a cone of half-angle $\arctan \mu$ about the surface normal.

A grasp has **force closure** when its contact wrenches positively span the wrench space: any disturbance can be opposed by some non-negative combination of contact forces. In the plane this reduces to a readable test — no half-plane may contain all the wrench generators.

INTERACTIVE · `ch20-grasp-wrench`

Force closure: when a grasp actually holds

Drag contact points around an object and watch their friction cones rotate. When the cones' span collapses into a half-plane, force closure fails and a direction of push exists that the grasp cannot resist.

Run this simulation in the web edition: `/chapters/learning-manipulation #ch20-grasp-wrench`

Force closure is binary; quality is continuous. The **Ferrari–Canny metric** takes the radius of the largest wrench ball the grasp can resist — the worst-case disturbance it survives. That is the difference between a grasp that technically holds and one that survives being carried across a room.

What modern systems actually do is skip the geometry. A network takes a depth image, proposes candidate grasps, and scores each one — approximating the force-closure computation without ever reconstructing the object. This is the payoff for Chapter 9: **grasp selection is exactly the setting where value-based methods belong**, because the action space is a few hundred discrete candidates and the argmax is both tractable and correct.

WHERE THE ANALYTIC THEORY STILL EARNS ITS PLACE

Learned scorers generalize to novel objects; analytic metrics require geometry you rarely have. But the theory remains useful in three ways: as a **reward signal** for training in simulation where geometry *is* known; as a **sanity check** on what a learned scorer has discovered; and as an **explanation** when a grasp fails — "the contacts were on the same side" is a diagnosis, "the network scored it 0.3" is not.

20.3 Contact-rich manipulation and impedance actions

Insertion is the canonical contact-rich task: a peg into a hole with clearance smaller than the robot's positioning error. Pure position control fails by construction — commanding a position the peg cannot reach produces enormous contact forces and a jammed or damaged part.

The answer, from Chapter 13's operational-space material, is **impedance control**. Instead of commanding position, command a target position *and a stiffness*, making the robot behave like a programmable spring:

$$F = K_p(x_d - x) + K_d(\dot{x}_d - \dot{x}).$$

Low stiffness in the directions where contact is expected lets the environment guide the part; high stiffness along the insertion axis maintains the push.

WHY THIS IS AN ACTION-SPACE DECISION, NOT A CONTROL DETAIL

This is Chapter 17's thesis in its sharpest form. A policy commanding *positions* must learn to avoid excessive contact forces — a hard, safety-critical thing to learn, where every training mistake is a collision. A policy commanding *impedance* has compliance built into the action space: it physically cannot generate large forces in the low-stiffness directions.

The mechanical compliance performs the search. The classic "insertion funnel" — where contact geometry guides a compliant part into alignment — is doing work that a stiff controller would have to compute exactly. The policy's job shrinks from "solve the contact problem" to "choose stiffness and approach direction", which is a far smaller problem.

Published comparisons repeatedly find impedance action spaces outperforming position control on contact-rich tasks by large margins, using the same algorithm. The algorithm was never the bottleneck.

Why planning through contact is intractable is worth stating precisely, because it explains why learning is attractive here. With n contact points, each is separated, sticking, or sliding — 3^n discrete modes, each with different dynamics. A peg-in-hole with 8 potential contacts has 6561 modes. Planning must search

over mode sequences, and the search is combinatorial in a quantity that grows with the task's contact richness. Learned policies sidestep the enumeration entirely by never representing modes explicitly.

20.4 In-hand manipulation: what dexterity cost

Reorienting an object within the hand, using the fingers alone, is the hardest thing manipulators do — many contacts, all of them making and breaking, with the object's pose partially occluded by the hand doing the manipulating.

The result that defined the area used no new algorithm. It used everything from Part III, at maximum intensity: **extreme domain randomization** (masses, frictions, object dimensions, gravity, actuator dynamics, visual appearance), a **recurrent policy** performing the implicit system identification of Chapter 15, distributed PPO from Chapter 10, and enormous compute.

> ⚠ **THE HONEST READING**
>
> It worked, and it sits at L2 — validated in diverse lab conditions, not deployed. The compute was measured in thousands of CPU-years. The policy handled one object class in a controlled setting, with a motion-capture-instrumented environment.
>
> The right conclusion is not that dexterity is solved, nor that the approach failed. It is that **randomization and scale can substitute for simulation accuracy, at a price** — and the price is high enough that the interesting question is what cheaper substitute exists. Chapter 16's demonstrations and Chapter 12's learned models are the two leading candidates, and neither has settled it.

20.5 Where the techniques land

Assembling Part IV's map for manipulation:

Subproblem	What works	Chapter
Grasp selection	Learned scoring over discrete candidates	9 (value methods)
Contact-rich insertion	SAC with impedance action space	11, 13, 17
In-hand reorientation	Extreme randomization + recurrent policy	15, 19
Novel-object picking	Demonstrations + offline RL, fine-tuned	16

Subproblem	What works	Chapter
Deformable objects	Largely open; demonstrations most promising	16, 21

The pattern in Tang's tables is consistent: manipulation successes lean on **expert data** far more than locomotion successes, which lean on **simulation**. That is exactly what the four criteria of §20.1 predict — when simulation is untrustworthy and reward design is hard, human demonstrations supply both the data and the specification.

RUST **rl-envs/src/manipulation/impedance.rs**

```rust
use nalgebra::{Vector3, Matrix3};

/// The policy's action: where to go, and how hard to insist on getting there.
pub struct ImpedanceAction {
    pub target_pos: Vector3<f64>,
    /// Per-axis stiffness in the TASK frame -- low along expected contact
    /// directions lets the environment guide the part.
    pub stiffness: Vector3<f64>,
}

pub struct ImpedanceWrapper {
    kp_range: (f64, f64),
    damping_ratio: f64,   // zeta; 1.0 = critically damped
}

impl ImpedanceWrapper {
    /// Map a normalized policy output in [-1, 1] to a physical command.
    pub fn decode(&self, raw: &[f64], current_pos: &Vector3<f64>) -> ImpedanceAction {
        let delta = Vector3::new(raw[0], raw[1], raw[2]) * 0.05;   // +/-5 cm
        let stiffness = Vector3::from_iterator((3..6).map(|i| {
            let t = (raw[i] + 1.0) / 2.0;                          // -> [0, 1]
            self.kp_range.0 * (self.kp_range.1 / self.kp_range.0).powf(t)  // log-spaced
        }));
        ImpedanceAction { target_pos: current_pos + delta, stiffness }
    }

    /// Cartesian impedance law, mapped to joint torques by J^T (Chapter 13).
    /// Damping is derived from stiffness so the system stays critically damped
    /// as the policy varies Kp -- otherwise a stiffness change causes ringing.
    pub fn to_joint_torques(
        &self,
        action: &ImpedanceAction,
        x: &Vector3<f64>,
        xdot: &Vector3<f64>,
        jacobian: &nalgebra::Matrix3<f64>,
        mass_est: f64,
    ) -> Vector3<f64> {
        let kp = Matrix3::from_diagonal(&action.stiffness);
        let kd = Matrix3::from_diagonal(&action.stiffness.map(|k| {
            2.0 * self.damping_ratio * (k * mass_est).sqrt()
        }));

        let force = kp * (action.target_pos - x) - kd * xdot;
        jacobian.transpose() * force
    }
```

```
}
```

An impedance action-space wrapper. The policy outputs a target and a stiffness; the wrapper turns that into joint torques through the Jacobian, so compliance is structural rather than learned.

20.6 Reporting results honestly

Manipulation is where evaluation goes wrong most often, so Chapter 14's methodology deserves restating with the specifics.

State the object set. "90% success" over five known objects in fixed poses is a different claim from 90% over fifty novel objects in clutter. Both may be worth publishing; conflating them is not.

Give an interval. With 20 trials and 18 successes, the point estimate is 90%. The Wilson score interval — appropriate near the boundary where the normal approximation fails — is roughly [70%, 97%]. Reporting 90% alone claims precision the experiment did not buy.

Report the failures. Which two failed, and how? A grasp that slipped is a different problem from one that never closed.

Report the human cost. How many resets, and did a person perform them? For manipulation this is usually the binding constraint on the entire experiment, and omitting it hides the real cost of the method.

20.7 Chapter bridge

Part IV is complete. Locomotion won because its dynamics simulate and its rewards shape. Navigation is unsettled between learned and classical because each fails differently. Manipulation lags because contact, diversity and cost all cut against it — and where it succeeds, it does so by leaning on human data rather than simulation.

Three competencies remain in Tang's taxonomy, and they share a property: **the environment contains other agents.** Chapter 21 takes on human–robot interaction, where the unmodelable part of the world is a person; multi-robot systems, where coordination is a Dec-POMDP with unpleasant complexity; and the frontier that has moved fastest since the survey was written — foundation models supplying the priors, goals and even rewards that this book has so far asked engineers to write by hand.

Exercises

1 FOUNDATION •• **Force closure by hand**

For two point contacts with friction coefficient μ on a circular object, derive the condition on their angular separation for force closure. Verify your answer against the widget by finding the boundary experimentally.

2 FOUNDATION ••• **Ferrari–Canny**

Define the Ferrari–Canny metric precisely as the radius of the largest wrench ball contained in the grasp wrench space. Then explain why a grasp can have force closure and near-zero quality, and what that looks like physically.

3 FOUNDATION •• **Contact-mode explosion**

Count contact modes for a square peg entering a square hole with 8 potential contact points. Then explain why sampling-based planners struggle here in a way they do not for free-space motion planning.

4 FOUNDATION ••• **Why compliance searches**

Model the insertion funnel: show that with a chamfered hole and low lateral stiffness, contact forces produce a lateral correction proportional to misalignment. Derive the maximum initial misalignment that self-corrects.

5 CONCEPTUAL • **Break the grasp**

In the grasp widget, find the smallest friction coefficient at which two opposed contacts still achieve force closure. Then move them to the same side and confirm no friction value rescues the grasp.

6 CONCEPTUAL •• **Does a third contact help?**

Add a third contact and find a placement that maximizes quality. Compare against the best two-contact grasp at the same friction. Quantify what the extra finger bought.

7 PRACTICAL ••• **Grasp-selection DQN**

Build a parallel-jaw gripper in rapier3d, generate candidate grasps on procedurally-varied objects, and train a DQN scorer. Compare against the analytic Ferrari–Canny ranking on objects where geometry is known — the disagreements are the interesting part.

8 PRACTICAL • • • **Impedance versus position**

Train SAC on peg insertion twice: once with position actions, once with impedance actions. Hold everything else fixed. Report success rate, peak contact force and sample count, then report success with a Wilson interval as §20.6 requires.

20.8 References

BASELINE REFERENCES

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§4.3 in full — grasping, pick-and-place, contact-rich manipulation, articulated and deformable objects, in-hand and non-prehensile manipulation, with the trend analysis behind §20.1.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
The manipulation lineage, including the peg-in-hole results that motivated compliant action spaces.

FURTHER READING AND MODERN SOURCES

Ferrari, C. & Canny, J. (1992). Planning optimal grasps *ICRA 1992*.
The grasp quality metric of §20.2.

Mason, M. T. (2001). Mechanics of Robotic Manipulation *MIT Press*.
Contact mechanics, friction cones and force closure, developed properly.

Mahler, J., Liang, J., Niyaz, S., Laskey, M., Doan, R., Liu, X., Ojea, J. A. & Goldberg, K. (2017). Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics *RSS 2017*.
Learned grasp scoring trained against analytic metrics — the bridge between §20.2's two halves.

Akkaya, I. et al. (OpenAI) (2019). Solving Rubik's Cube with a Robot Hand *arXiv:1910.07113*.
The in-hand result of §20.4, including automatic domain randomization.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J. & Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection *International Journal of Robotics Research 37(4–5)*.
Large-scale real-world grasp learning — the data-collection alternative to simulation.

Martín-Martín, R., Lee, M. A., Gardner, R., Savarese, S., Bohg, J. & Garg, A. (2019). Variable Impedance Control in End-Effector Space: An Action Space for Reinforcement Learning in Contact-Rich Tasks *IROS 2019*.

The impedance action space of §20.3, with the comparison against position control.

Kroemer, O., Niekum, S. & Konidaris, G. (2021). A Review of Robot Learning for Manipulation: Challenges, Representations, and Algorithms *Journal of Machine Learning Research 22(30)*.

The comprehensive manipulation-learning survey.

Part V

Frontiers and Capstone

CHAPTER 21

Frontiers: HRI, Multi-Robot & Foundation Models

> *For competencies where both accurate simulation and real-world rollouts are prohibitive, or where stable scalable algorithms are missing, successful real-world examples are much sparser.*
>
> **Chen Tang and colleagues** · On the frontier that remains
> *Deep Reinforcement Learning for Robotics — A Survey of Real-World Successes, 2024*

Three competencies remain, and they share one property: the environment contains other agents. Humans cannot be simulated, which removes the technique that made locomotion work. Multiple robots turn a single decision problem into one whose complexity class is genuinely worse. And the frontier that has moved fastest — foundation models — offers to supply the priors, goals and rewards this book has so far asked engineers to write by hand. We close by comparing Kober's 2013 open problems against Tang's 2024 list, which is the most honest way to see what a decade actually bought.

FOUNDATION

Latent-goal shared autonomy, the Dec-POMDP formalism with NEXP-completeness, CTDE and the MAPPO gradient, and the KL-regularized RLHF objective with its Boltzmann solution.

CONCEPTUAL

A shared-autonomy blend you control directly, and the 2013-versus-2024 open-problem diff.

PRACTICAL

Multi-agent gridworld with CTDE PPO, a shared-autonomy Reacher blending your input with a trained policy, and a VLM reward-labelling feasibility sketch.

AFTER THIS CHAPTER YOU CAN

- Explain why HRI is simulation-starved and what follows for algorithm choice
- Formulate shared autonomy as inference over a latent human goal
- State the Dec-POMDP formalism and its complexity, and explain what CTDE buys
- Derive the RLHF objective and identify its closed-form optimal policy
- Describe how vision-language-action models change the priors available to a robot
- Compare the 2013 and 2024 open-problem lists and say what genuinely changed

21.1 Humans are the unmodelable part

Every technique in Part III depended on simulation. Domain randomization needs a simulator to randomize; teacher–student needs privileged simulator state; massively parallel PPO needs thousands of simulated worlds.

None of that is available when a human is in the loop. There is no physics engine for a person deciding to reach for the same object you were reaching for. You can model human motion kinematically, but you cannot simulate the reaction to the robot's behaviour — and that reaction is precisely what physical human–robot interaction consists of.

The consequences follow directly. Zero-shot sim-to-real is unavailable, so real-world learning is mandatory. Real-world learning with humans is slow, supervised, and safety-critical, so data is scarce. Scarce data means Chapter 16's offline methods rather than Chapter 10's on-policy ones. And the safety case means Chapter 14's constrained formulation rather than a scalar penalty.

That chain of reasoning — not any algorithmic gap — is why HRI sits at L1–L2 on Chapter 1's ladder.

COLLABORATIVE, NON-COLLABORATIVE, AND SHARED

Three distinct problems get filed under HRI, and conflating them causes confusion.

Collaborative physical interaction — human and robot manipulate the

same object, with aligned goals. The difficulty is inferring intent from force cues in real time.
Non-collaborative — human and robot share space with different objectives, as in a robot crossing a busy corridor. Nobody is helping anybody; the robot must predict and not obstruct.
Shared autonomy — no physical contact, but control is blended. The human indicates intent through an interface and the robot supplies competence. This is the most deployed of the three, in assistive robotics and teleoperation.

21.2 Shared autonomy as goal inference

Shared autonomy has the cleanest formulation, so it is worth doing properly.

The human has a goal g the robot cannot observe. The robot receives human input u_t — joystick, gaze, muscle signal — which is noisy evidence about g. Maintain a belief over goals and update it with each input:

$$b_{t+1}(g) \propto P(u_t \mid g,\, s_t)\, b_t(g).$$

Then act to maximize expected value under that belief:

$$a_t = \arg\max_a \sum_g b_t(g)\, Q_g(s_t, a).$$

This is Chapter 4's belief-MDP machinery with the hidden variable being *the human's intention* rather than the robot's location. Everything transfers, including the honest observation that the belief space is continuous and exact solutions are out of reach.

INTERACTIVE · `ch21-shared-autonomy`

Assistance that helps, and assistance that annoys

The robot infers which target you want from noisy input and blends its action with yours. Raising the blend improves task success and lowers your sense of control — the trade no reward function here captures.

Run this simulation in the web edition: `/chapters/frontiers #ch21-shared-autonomy`

ASSISTANCE THAT IS CORRECT AND UNWELCOME

The formulation above optimizes task success. Users frequently dislike the result.
An assistant confident about your goal takes over, and the loss of agency is experienced as the system fighting you — especially when the inference is

wrong, which it periodically is. The standard mitigations are to blend rather than replace, $a = (1 - \alpha)a_{\text{human}} + \alpha\, a_{\text{robot}}$ with α tied to belief confidence; to apply a **Q-filter**, intervening only where the human's action is clearly suboptimal; and to preserve a predictable manual mode the user can always fall back to.
The deeper point is that the objective is not task success alone. It includes the human's sense of control, which is real, measurable by user study, and absent from every reward function in this book. That gap is itself a research frontier.

21.3 Multi-robot systems

Two robots in a warehouse are not two independent problems. Each is part of the other's environment, and since both are learning, each faces a **non-stationary** environment — the thing every convergence proof in Part I assumed away.

The formalism is the **decentralized POMDP**: a shared state and transition function, but each agent i has its own observation function O_i and must act on its own observation alone. The team shares one reward.

Theorem 21.1 — Dec-POMDPs are NEXP-complete (Bernstein et al., 2002). Deciding whether a finite-horizon Dec-POMDP has a joint policy achieving expected value at least V is NEXP-complete.

That is worse than the PSPACE-hardness of single-agent POMDPs, and it is not an artifact of a bad formulation. The difficulty is intrinsic: agents must reason about what other agents know, which requires reasoning about what they observe, recursively.

Centralized training with decentralized execution is the practical response, and it is exactly Chapter 15's asymmetric actor–critic applied to a team. During training the critic sees the joint state and all agents' observations, which makes its estimates stationary and low-variance. At execution each agent runs a policy conditioned only on its own observation. MAPPO is PPO with this structure and is a strong baseline despite its simplicity.

The residual difficulty is **credit assignment**: the team reward does not say who contributed. The standard fix is a counterfactual baseline — compare the achieved return against what would have happened had agent i acted differently, holding others fixed — which is Chapter 10's baseline argument, extended to teams.

WHERE MULTI-ROBOT RL ACTUALLY WORKS

Tang's survey finds the successes concentrated where coordination is *implicit*: decentralized collision avoidance, where each robot has a local policy and coordination emerges from everyone following compatible rules; and robot soccer, where full-body control on physical platforms reached L4 through a combination of learned low-level skills and structured high-level play.

Where it does not work is anything requiring explicit negotiated coordination at scale. The complexity result is not merely theoretical — it shows up as training instability that grows with the number of agents.

21.4 Foundation models

The fastest-moving frontier, and the one most likely to date this chapter.

Large vision-language models bring something robot RL has always lacked: **priors about the world that nobody had to encode**. Four ways they are being used, in increasing order of ambition.

Reward specification. Ask a VLM whether an image shows the task completed. This attacks Chapter 14's fourth curse directly — instead of engineering a reward, describe the goal in language. It works for visually-checkable goals and fails where success is a matter of forces or internal state.

Goal and plan generation. An LLM decomposes "clear the table" into a sequence of skills the robot already has. This is Chapter 19's skill-composition problem with the composer supplied by a language model rather than learned, and it partially answers the open question of where high-level structure comes from.

Vision-language-action models. Train a single large policy on diverse robot data with language conditioning, so it maps instruction plus image to action. The bet is that scale and diversity produce generalization to novel objects and phrasings — a bet that has paid off in other domains.

RL fine-tuning of large policies. Pretrain on diverse data, then improve with RL on the specific robot. The objective is the RLHF one:

$$\max_{\theta} \; \mathbb{E}_{a\sim\pi_\theta}\left[r_\psi(s,a)\right] - \beta\, D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}}),$$

whose optimal solution is available in closed form,

$$\pi^*(a \mid s) \propto \pi_{\mathrm{ref}}(a \mid s)\, \exp\big(r_\psi(s,a)/\beta\big).$$

YOU HAVE SEEN THAT SOLUTION BEFORE

That is a Boltzmann distribution over the reward, tilted by a reference policy — structurally identical to Chapter 11's maximum-entropy optimal policy $\pi^* \propto \exp(Q/\alpha)$, with the KL to a reference playing the role entropy played there.

The connection is not superficial. Both are instances of regularized policy optimization, where a divergence penalty keeps the solution near something trusted: a uniform policy in the entropy case, a pretrained policy here. Chapter 10's trust region is the same idea with a hard constraint.

Recognizing this saves effort. The same intuitions about temperature, the same failure modes when the penalty is too weak, and the same closed-form structure all carry over.

RUST `rl-deep/src/multiagent/ctde.rs`

```rust
use burn::prelude::*;

pub struct CtdePpo<B: Backend> {
    /// One actor per agent, each conditioned on that agent's observation only.
    actors: Vec<GaussianPolicy<B>>,
    /// A single centralized critic over the JOINT observation. Training-time
    /// only -- it is discarded before deployment.
    critic: Critic<B>,
    n_agents: usize,
}

impl<B: AutodiffBackend> CtdePpo<B> {
    /// Execution: each agent acts on what it can actually see.
    pub fn act(&self, obs: &[Tensor<B, 2>]) -> Vec<Tensor<B, 2>> {
        obs.iter()
            .zip(&self.actors)
            .map(|(o, actor)| actor.sample(o.clone()).0)
            .collect()
    }

    /// Training: the value estimate may use the joint state, which makes it
    /// stationary from each agent's perspective even while the others learn.
    pub fn value(&self, joint_obs: Tensor<B, 2>) -> Tensor<B, 1> {
        self.critic.forward(joint_obs).squeeze(1)
    }

    /// Counterfactual advantage: how much better was agent i's action than its
    /// average alternative, holding the other agents fixed? This is Chapter 10's
    /// baseline argument, extended to assign team credit.
    pub fn counterfactual_advantage(
        &self,
        agent: usize,
        joint_obs: Tensor<B, 2>,
        joint_actions: &[Tensor<B, 2>],
        n_samples: usize,
    ) -> Tensor<B, 1> {
        let q_actual = self.joint_q(joint_obs.clone(), joint_actions);

```

```
        // Marginalize agent `agent` out by resampling its action.
        let mut baseline = q_actual.zeros_like();
        for _ in 0..n_samples {
            let mut counterfactual = joint_actions.to_vec();
            counterfactual[agent] = self.actors[agent]
                .sample(self.agent_obs(&joint_obs, agent))
                .0;
            baseline = baseline + self.joint_q(joint_obs.clone(), &counterfactual);
        }
        q_actual - baseline.div_scalar(n_samples as f64)
    }
}
```

Centralized training with decentralized execution. The critic sees everything; each actor sees only its own observation — which is what makes the learned policies deployable on robots that cannot share state at runtime.

21.5 What a decade actually bought

The most useful thing this chapter can do is compare the open problems Kober's survey listed in 2013 against those Tang's listed in 2024.

2013 open problem	Status in 2024
Sample efficiency on real robots	**Partly solved by avoidance.** Sim-to-real made real samples largely unnecessary for locomotion and navigation. For contact-rich manipulation and HRI it remains open, and the honest summary is that we routed around the problem rather than solving it.
Reward/goal specification	**Open, with new tools.** Potential-based shaping was known in 1999. What is new is demonstrations at scale, preference learning, and language-model reward specification. Reward hacking remains a live hazard.
Model learning and mental rehearsal	**Advanced substantially.** Ensembles with calibrated uncertainty, latent world models, and short branched rollouts are real progress on 2013's forward-model methods. Still a minority of deployed systems.
Prior knowledge and structure	**Transformed.** In 2013 priors were hand-designed primitives. Now they also come from internet-scale pretraining — a source that did not exist.
Safe exploration	**Open.** Constrained RL matured, but the field mostly avoids the problem by exploring in simulation. Real-robot safe exploration is still largely human supervision.
Tractable representations	**Solved differently than expected.** 2013 sought compact hand-designed representations; deep networks made high-dimensional representations tractable instead. DMPs remain the right answer for some problems.

And the genuinely new problems, absent from the 2013 list:

Long-horizon tasks and skill discovery. Not prominent in 2013 because short-horizon tasks were still hard. Now the main obstacle to general-purpose robots, and Chapter 19 stated why.

Principled system design. Tang's survey is pointed about this: action spaces, reward terms and architectures are chosen by heuristic and expert taste, with

few controlled comparisons. This book's insistence on classical baselines and reported units is a small response to that complaint.

Scaling multi-robot learning. Barely addressed in 2013; now blocked by the complexity result of §21.3.

> **THE HONEST SUMMARY**
>
> A decade of extraordinary progress, and the four curses are all still there. Dimensionality is handled by generalization rather than removed. Sample cost is routed around by simulation rather than reduced, for the tasks where simulation works. Under-modelling is managed by randomization rather than fixed. Goal specification has more tools and no solution.
> That is not a pessimistic reading. It is what mature engineering fields look like — the fundamentals persist, and the craft consists of knowing which technique to apply to which manifestation. A reader who finishes this book knowing the four curses, the techniques that manage each, and how to tell which one is biting is equipped for the frontier as it actually is.

21.6 Chapter bridge

The frontier is where other agents enter: humans who cannot be simulated, teams whose complexity is provably worse, and foundation models supplying priors nobody wrote.

One thing remains. This book has built twenty-one chapters of machinery and demonstrated each piece separately. Chapter 22 assembles all of it into a single project: specify a task, formalize the MDP with every choice justified against the evidence in Parts III and IV, build a randomized simulation, train teacher–student PPO, evaluate against the L0–L5 rubric with honest statistics, and ship a native binary alongside a browser demo. It is the chapter that turns a book into a template you can fork.

Exercises

> 1 FOUNDATION •• **Shared-autonomy belief update**
>
> Write the belief update over human goals given joystick input, specifying an observation model $P(u \mid g, s)$. Then show how the blending coefficient should depend on belief entropy, and what happens at maximum entropy.

> 2 FOUNDATION ••• **Why Dec-POMDPs are worse**
>
> Explain intuitively why decentralized execution raises complexity from PSPACE to

NEXP. What must each agent reason about that a single agent need not?

3 FOUNDATION • • • **The CTDE gradient**

Write the policy gradient for agent i under CTDE with a centralized critic. Prove it is unbiased despite the critic seeing information the actor does not — and connect the argument to Chapter 15's asymmetric actor–critic.

4 FOUNDATION • • • **The RLHF solution**

Derive the closed-form optimal policy for the KL-regularized objective. Then show it reduces to Chapter 11's maximum-entropy policy when the reference policy is uniform.

5 CONCEPTUAL • • **Assistance that helps**

In the shared-autonomy widget, find the blending level at which task performance is best. Then find the level at which YOU feel in control. Report the gap — it is the research problem.

6 CONCEPTUAL • • **Read the diff**

Using the 2013-versus-2024 table, pick the problem you judge to have advanced least and argue why, citing specific obstacles from Chapters 14–20 rather than general difficulty.

7 PRACTICAL • • • **Two Rustys**

Build a two-agent delivery gridworld and train CTDE PPO. Then train independent PPO with no centralized critic. Compare stability across five seeds — the independent version should be visibly worse, and you should be able to say why.

8 PRACTICAL • • • **VLM reward labelling**

Take 100 Reacher end-states, label success with a small vision model, and compare against ground truth. Measure precision and recall, then estimate how much reward noise a policy can tolerate before training degrades.

21.7 References

BASELINE REFERENCES

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.
§4.5 human–robot interaction, §4.6 multi-robot interaction, §5 general trends and open challenges, §6 conclusion — the 2024 column of §21.5.

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.
§8 discussion — open questions and practical challenges, the 2013 column of §21.5.

FURTHER READING AND MODERN SOURCES

Bernstein, D. S., Givan, R., Immerman, N. & Zilberstein, S. (2002). The Complexity of Decentralized Control of Markov Decision Processes *Mathematics of Operations Research 27(4)*.
Theorem 21.1.

Javdani, S., Admoni, H., Pellegrinelli, S., Srinivasa, S. S. & Bagnell, J. A. (2018). Shared autonomy via hindsight optimization for teleoperation and teaming *International Journal of Robotics Research 37(7)*.
The latent-goal formulation of §21.2.

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A. & Wu, Y. (2022). The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games *NeurIPS 35*.
MAPPO — CTDE with PPO, and why it is hard to beat.

Foerster, J., Farquhar, G., Afouras, T., Nardelli, N. & Whiteson, S. (2018). Counterfactual Multi-Agent Policy Gradients *AAAI 2018*.
The counterfactual baseline for multi-agent credit assignment.

Haarnoja, T. et al. (2024). Learning agile soccer skills for a bipedal robot with deep reinforcement learning *Science Robotics 9(89)*.
The robot-soccer result cited in §21.3.

Brohan, A. et al. (2023). RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control *CoRL 2023*.
The VLA approach of §21.4.

Ouyang, L. et al. (2022). Training language models to follow instructions with human feedback *NeurIPS 35*.
The KL-regularized RLHF objective whose closed form appears in §21.4.

CHAPTER 22

Capstone: An End-to-End Learned Robot in Rust

> *By analyzing a simple problem in some detail we demonstrate how reinforcement learning approaches may be profitably applied, and we note throughout open questions and the tremendous potential for future research.*
>
> **Jens Kober, J. Andrew Bagnell & Jan Peters** · On the discipline of a worked case study
> *Reinforcement Learning in Robotics — A Survey, IJRR 2013*

One project, using the whole book. Ferris must patrol a cluttered course, reach a sequence of waypoints, and recover from being pushed. We specify the task, formalize the MDP with every choice justified against evidence from Parts III and IV, build the randomized simulation, train teacher–student PPO, evaluate against the L0–L5 rubric with statistics that survive scrutiny, and ship both a native binary and a browser demo. The deliverable is not a trained policy — it is a reproducible pipeline you can fork for your own robot.

FOUNDATION

The assembled POMDP: a 48-dimensional observation with units, the full reward table, the randomization ranges, and the evaluation protocol.

CONCEPTUAL

Mission control — training curves, randomization draws, evaluation heatmaps and a failure-mode gallery with post-mortems.

PRACTICAL

The capstone crate: serde/clap config, rayon rollout farm, burn training, rerun telemetry, and a WASM deployment.

◎ AFTER THIS CHAPTER YOU CAN

- Write a complete formal problem statement for a real robot task, with every design choice justified
- Build a config-driven experiment runner whose results reproduce exactly from a seed
- Assemble the teacher–student pipeline end to end and know what each stage contributes
- Evaluate a policy against the L0–L5 rubric with confidence intervals that reflect the sample size
- Diagnose failures from telemetry rather than from guesswork
- State honestly what level this work reaches and what each further level would require

22.1 The specification, written before any code

Kober's survey closes with a case study, and the discipline it models is worth copying: **write the specification before touching the implementation.** Most robot-learning projects fail at this step rather than at training.

Task. Ferris patrols a course of randomly-generated rough terrain with scattered obstacles, visiting a sequence of waypoints in order. He must recover from external pushes and from stumbles, and complete the circuit within a time budget.

Success criteria, stated so a disinterested party could check them:

- Visits all four waypoints in order, within 3 minutes.
- Survives at least three lateral pushes of 150 N applied for 0.2 s at random times.
- No base-ground contact (a "fall") at any point.
- Mean cost of transport below 1.2.

Explicit non-goals. Not: manipulation of any kind, navigation of unmapped buildings, operation among people, or terrain types outside the generated distribution. Stating these matters — an unbounded task specification is how projects acquire scope that no evaluation can cover.

22.2 The formal problem

Now the MDP, with each choice traceable to a chapter.

Observation (48 dimensions; what the deployed student sees):

Component	Dim	Units	Justification
Base angular velocity	3	rad/s	IMU; directly measured (Ch 18)
Projected gravity	3	–	Yaw-invariant orientation (Ch 18)
Waypoint vector (body frame)	3	m	The task; body frame for translation invariance
Joint positions	12	rad	Encoders
Joint velocities	12	rad/s	Differentiated encoders
Previous action	12	rad	Enables smooth output (Ch 18)
Gait phase	2	–	$(\sin\phi, \cos\phi)$ clock (Ch 17)
Time remaining	1	–	Normalized; makes the finite horizon Markov (Ch 4)

Privileged observation (teacher only, +31 dimensions): terrain heights on a 4×4 grid under the body, true friction, per-foot contact states, applied external force, and true base linear velocity. Discarded after distillation (Ch 15).

Action. Twelve joint position targets at 50 Hz, tracked by a 1 kHz PD loop with $K_p = 40$, $K_d = 1$. Mid-level, per Chapter 17 and universal practice in Chapter 18.

> **WHY 'TIME REMAINING' IS IN THE OBSERVATION**
>
> A finite-horizon task is not Markov in the state alone: the optimal action near the deadline differs from the optimal action at the start, and a policy that cannot see the clock must average over both.
> Including normalized time remaining restores the Markov property. It is a small detail that produces a visible behaviour change — the policy learns to hurry — and it is the kind of thing that is obvious in Chapter 4's formalism and easy to forget in code.

Reward. The Chapter 18 locomotion terms, plus task terms:

Term	Formula	Units	Weight		
Waypoint progress	Δdist to next waypoint	m	+2.0		
Waypoint reached	indicator	–	+50.0		
Velocity tracking	$\exp(-\	v_{xy}-v^*\	^2/0.25)$	–	+1.0
Orientation	$\	g_{\text{proj},xy}\	^2$	–	−5.0
Effort	$\	\tau\	^2$	N^2m^2	$-2\times10^{-}$
Foot air time	$\sum_f (t_{\text{air}} - 0.5)$	s	+1.0		
Foot slip	$\sum_f \	v_f\	^2\mathbb{1}[\text{contact}]$	m^2/s^2	−0.1
Action rate	$\	a_t - a_{t-1}\	^2$	rad^2	−0.01

Term	Formula	Units	Weight
Fall	indicator	–	−100.0 (terminates)

Waypoint progress is **potential-based** — $\gamma\Phi(s') - \Phi(s)$ with $\Phi = -\text{dist}$ — so by Chapter 14's Theorem 14.1 it accelerates learning without changing the optimal policy. The waypoint bonus is the actual objective.

Randomization (Ch 15): link masses ±20%, friction 0.4–1.2, motor K_p ±30%, latency 0–30 ms, IMU noise, terrain roughness scheduled by curriculum, and random pushes of 50–200 N.

22.3 The pipeline

Five stages, each an earlier chapter cashed in.

1. **Environment** — Ferris in `rapier3d`, procedural terrain, 4096 parallel instances via `rayon` (Ch 13, 15).
2. **Teacher** — PPO with privileged observations, terrain curriculum with per-environment promotion (Ch 10, 18).
3. **Student** — GRU policy over a 15-step observation history, trained by DAgger-style distillation on its own state distribution (Ch 15, 16, 19).
4. **Evaluation** — held-out terrain seeds, the push protocol, L0–L5 rubric with Wilson intervals (Ch 14, 20).
5. **Deployment** — native binary plus a WASM build with live telemetry.

RUST **capstone/src/main.rs**

```rust
use clap::Parser;
use serde::{Deserialize, Serialize};

#[derive(Parser)]
#[command(name = "ferris-capstone")]
struct Cli {
    /// Path to the experiment config. Everything reproducible lives here.
    #[arg(short, long, default_value = "configs/patrol.toml")]
    config: String,

    #[arg(short, long)]
    stage: Stage,

    /// Overrides the config seed; recorded in the results either way.
    #[arg(long)]
    seed: Option<u64>,
}

#[derive(Clone, Copy, clap::ValueEnum)]
enum Stage { Teacher, Student, Evaluate, Export }

#[derive(Serialize, Deserialize, Clone)]
```

```rust
struct ExperimentConfig {
    seed: u64,
    env: EnvConfig,
    randomization: RandomizationRanges,
    curriculum: CurriculumConfig,
    reward: Vec<RewardTermConfig>,   // weights live in config, never in code
    ppo: PpoConfig,
    distillation: DistillConfig,
    evaluation: EvalConfig,
}

fn main() -> anyhow::Result<()> {
    let cli = Cli::parse();
    let raw = std::fs::read_to_string(&cli.config)?;
    let mut cfg: ExperimentConfig = toml::from_str(&raw)?;
    if let Some(s) = cli.seed { cfg.seed = s; }

    // Hash the exact config into every artifact, so a result can always be
    // traced back to the settings that produced it.
    let config_hash = blake3::hash(raw.as_bytes()).to_hex().to_string();
    tracing::info!(seed = cfg.seed, config = %config_hash, "starting");

    match cli.stage {
        Stage::Teacher => {
            let mut envs = FerrisVecEnv::new(&cfg.env, &cfg.randomization, cfg.seed)?;
            let mut teacher = PrivilegedPolicy::new(&cfg.ppo);
            let mut curriculum = TerrainCurriculum::new(&cfg.curriculum);
            train_ppo(&mut teacher, &mut envs, &mut curriculum, &cfg, &config_hash)?;
        }
        Stage::Student => {
            let teacher = PrivilegedPolicy::load("artifacts/teacher.mpk")?;
            let mut student = RecurrentPolicy::new(&cfg.distillation);
            // DAgger, not behaviour cloning: roll out the STUDENT, label with
            // the teacher. Chapter 16 explains why this distinction decides it.
            distill_on_policy(&teacher, &mut student, &cfg, &config_hash)?;
        }
        Stage::Evaluate => {
            let student = RecurrentPolicy::load("artifacts/student.mpk")?;
            let report = evaluate(&student, &cfg.evaluation, cfg.seed)?;
            report.print_with_wilson_intervals();
            report.write_json("artifacts/evaluation.json")?;
        }
        Stage::Export => export_wasm("artifacts/student.mpk", "dist/")?,
    }
    Ok(())
}
```

The config-driven runner. Every number that affects the result lives in one TOML file that is hashed into the results — which is what makes 'reproducible' a checkable claim rather than an aspiration.

22.4 Evaluation, and an honest verdict

Chapter 14 set the standard; here it is applied to our own work.

INTERACTIVE · ch22-mission-control

Mission control

The capstone's telemetry: training curves across five seeds, reward-term composition, evaluation with Wilson intervals, and the failure post-mortems.

Run this simulation in the web edition: /chapters/capstone #ch22-mission-control

Protocol. 200 episodes on terrain seeds disjoint from training, with the push protocol applied. Five independently trained policies (five training seeds), reported separately rather than pooled.

Results, as they would be reported:

Metric	Value	95% interval
Circuit completion	176 / 200	88.0% [82.7%, 91.9%]
No-fall rate	189 / 200	94.5% [90.4%, 96.9%]
Push recovery (3+ pushes survived)	171 / 200	85.5% [79.9%, 89.8%]
Mean cost of transport	0.94	±0.11 across seeds
Median completion time	108 s	budget 180 s

The intervals are Wilson score intervals, appropriate near the boundary where the normal approximation misleads. Across the five training seeds, completion ranged from 81% to 91% — a spread worth reporting, because a paper claiming 91% from one seed would be reporting the best of five.

THIS WORK IS L0, AND SAYING SO IS THE POINT

Chapter 1's rubric is unambiguous: **Level 0 — validated only in simulation environments.** No physical robot ran this policy. Claiming otherwise would be exactly the overstatement Chapter 14 warned about, and the rubric exists to make such claims checkable.

What each further level would require:

L1 (limited lab) — a physical quadruped, flat ground, controlled lighting, a person ready to catch it. Mostly hardware access and a safety procedure.

L2 (diverse lab) — multiple terrain types and surfaces indoors, tens of hours of operation. This is where the randomization ranges get validated, and where the first genuinely surprising failures appear.

L3 (confined real-world) — a defined outdoor site, weather within a stated range, unsupervised operation. Requires thermal management, battery behaviour under load, and sensor degradation — none of which is in our

simulator.
L4 (diverse real-world) — arbitrary terrain, unconstrained conditions. This is where the published quadruped results sit, and the gap from L2 to L4 has historically taken teams years.
Being explicit about this is not modesty. It is the difference between a result someone can build on and a claim someone must first debunk.

22.5 Failure modes, with post-mortems

The failures are more instructive than the successes, and each traces to a specific decision.

Stuck at a gap (11 of 24 failures). Ferris approaches a gap wider than his stride and oscillates at the edge. *Diagnosis:* the terrain curriculum generated gaps up to 40 cm, but the waypoint-progress reward rewards approaching the goal, so the policy is pulled toward an edge it cannot cross. *Fix:* either a lateral-detour skill (Ch 19's hierarchy) or a curriculum that pairs each gap with a traversable bypass so the policy learns detouring is acceptable.

Push recovery failure on low friction (7 of 24). Pushes at $\mu = 0.4$ cause a slide into a fall. *Diagnosis:* the randomization range covers $\mu \in [0.4, 1.2]$, but pushes were sampled uniformly over time rather than conditioned on friction, so the low-friction-plus-push corner is rare in training. *Fix:* stratify the randomization so hard combinations are sampled deliberately.

Timeout on rough terrain (4 of 24). Completes the circuit but past the budget. *Diagnosis:* the velocity-tracking weight is too low relative to the effort penalty on rough ground, so the policy trades speed for economy. *Fix:* condition the commanded velocity on remaining time, or reweight — a Chapter 18 reward-mixer decision.

Distillation gap (2 of 24). The student fails where the teacher succeeds. *Diagnosis:* the 15-step history is insufficient to infer terrain that the teacher observed directly. *Fix:* a longer history, or an explicit terrain-estimation auxiliary loss.

EVERY FIX IS A CHAPTER

Notice that none of these diagnoses is "the algorithm underperformed", and none of the fixes is "tune the learning rate". Each is a design decision from Parts III and IV: curriculum coverage, randomization stratification, reward weighting, observation history length.
That is the practical thesis of this book. Once the algorithm works — and PPO works — the remaining engineering is problem formulation, and the four curses are where it lives.

22.6 What you have

The repository is a template. Fork it, replace the URDF and the reward terms, and the pipeline carries over: config-driven experiments with hashed provenance, a parallel rollout farm, teacher–student training, evaluation with intervals, and a browser demo for showing people what you built.

Where to go next, in rough order of value: put it on hardware, because every assumption in §22.2 is a hypothesis until a physical robot tests it. Add a manipulator and confront Chapter 20's difficulties. Replace the hand-specified waypoint sequence with learned skills and take on Chapter 19's open question. Or take the pipeline to a task where simulation does not work, and find out what Chapter 16's methods can do.

22.7 A closing note

Twenty-two chapters ago, Rusty drove into a concave trap and stayed there, running a controller that was correct in the sense that mattered to its author and useless in the sense that mattered to the robot.

The distance from there to here is not a collection of algorithms. It is a way of thinking: state the problem formally, know which assumptions your robot violates, choose the representation before the optimizer, measure against a classical baseline, and report what you actually established rather than what you hoped.

The four curses are still there. They will be there for your project too. Knowing their names, and which technique manages each, is what this book was for.

Exercises

1 FOUNDATION •• **Justify every observation**

For each of the eight observation components in §22.2, state what fails if it is removed. Then propose one addition and argue whether it earns its dimensions.

2 FOUNDATION •• **Check the shaping**

Verify that the waypoint-progress term is potential-based and therefore policy-invariant. Then show that adding a raw $-$distance term instead would change the optimal policy, and describe the behaviour it would produce.

3 FOUNDATION •• **Horizon and discount**

The budget is 180 s at 50 Hz — 9000 steps. Choose γ so the effective horizon covers a waypoint leg but not the whole circuit, and justify the choice against the failure modes

in §22.5.

4 FOUNDATION •• **Read the intervals**

Compute the Wilson interval for 176/200 and confirm the reported figure. Then determine how many episodes would be needed to establish completion above 85% with 95% confidence.

5 CONCEPTUAL •• **Diagnose from telemetry**

From the mission-control dashboard, identify which reward term is being sacrificed during the timeout failures. Then predict the weight change that would fix it, and what it would cost elsewhere.

6 CONCEPTUAL •• **Grade another system**

Take a recent robot-RL paper and grade it on the L0–L5 rubric using only its reported experiments. Then compare with the level the abstract implies.

7 PRACTICAL ••• **Reproduce and perturb**

Run the capstone from the pinned config and confirm the results reproduce. Then change one reward weight by 20%, rerun, and report what changed. Small perturbations should produce visible behaviour changes — that sensitivity is the finding.

8 PRACTICAL ••• **Fix a failure mode**

Pick one failure from §22.5 and implement its proposed fix. Re-evaluate with the full protocol and report whether the fix helped, hurt elsewhere, or did neither — all three are legitimate results, and the third is the most common.

22.8 References

BASELINE REFERENCES

BASELINE Kober, J., Bagnell, J. A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey *International Journal of Robotics Research 32(11)*.

§7 the ball-in-a-cup case study — the model for this chapter's discipline of specifying before implementing.

BASELINE Tang, C. et al. (2024). Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes *Annual Review of Control, Robotics, and Autonomous Systems*.

§3.4 the L0–L5 rubric applied to our own work in §22.4, and §5 on principled system design.

BASELINE Sutton, R. S. & Barto, A. G. (2018). Reinforcement Learning: An Introduction *MIT Press, 2nd edition*.
The formalism underlying §22.2, from the MDP definition through policy gradients.

FURTHER READING AND MODERN SOURCES

Rudin, N., Hoeller, D., Reist, P. & Hutter, M. (2022). Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning *CoRL 2021*.
The parallel-training regime and curriculum structure this capstone follows.

Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V. & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain *Science Robotics 5(47)*.
The teacher–student pipeline of §22.3, and the L4 result our L0 work would need years to approach.

Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference *Journal of the American Statistical Association 22(158)*.
The score interval used throughout §22.4.

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. & Bellemare, M. G. (2021). Deep Reinforcement Learning at the Edge of the Statistical Precipice *NeurIPS 34*.
Why per-seed reporting matters, as practised in §22.4.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. & Meger, D. (2018). Deep Reinforcement Learning that Matters *AAAI 2018*.
The reproducibility problems this chapter's config-hashing and seed reporting are designed to avoid.