Robotics Reinforcement Learning in Action

Build reproducible goal-conditioned manipulation agents

Vlad Prytula

2026-02-28

Contents

\newpage

Part 1 -- Start Running, Start Measuring

\newpage

1 Proof of Life: A Reproducible Robotics RL Loop

This chapter covers:

- Setting up a containerized MuJoCo + Gymnasium-Robotics environment that works on Linux (NVIDIA GPU) and Mac (Apple Silicon)
- Verifying GPU access, physics simulation, and headless rendering through a structured test sequence
- Running a complete training loop and inspecting the artifacts it produces
- Establishing the experiment contract -- checkpoints, metadata, evaluation reports -- that you will reuse in every later chapter
- Introducing three diagnostic questions that prevent wasted compute

Reinforcement learning for robotics has an uncomfortable failure mode: everything runs, nothing learns. There are no compiler errors, no stack traces, no red text. The training loop finishes, the checkpoint saves, and when you evaluate the policy, the robot sits still -- or flails -- or does something that looks vaguely intelligent but succeeds 3% of the time.

This chapter helps you catch that failure mode early. By the end, you will have a working environment you can trust -- not because it ”seems fine,” but because you have concrete evidence: a rendered frame proving the physics engine works, a saved checkpoint proving the training loop completes, and a set of diagnostic habits that will serve you for the rest of the book.

We call this a ”proof of life” -- borrowed from hostage negotiations, where it means evidence that someone is still alive. Here, it means evidence that the computational environment is alive: capable of producing valid, reproducible results.

The chapter is structured in two parts. First, we inspect the environment by hand -- looking at observations, manually computing rewards, and checking success conditions (section 1.5, ”Build It”). Then we run the automated verification pipeline that checks the full stack from GPU to training loop (section 1.6, ”Run It”). This two-part pattern -- understand the pieces, then run the pipeline -- recurs in every chapter of the book.

> **Tip:** If you want to verify your setup immediately, jump to section 1.6 and run `bash docker/dev.sh python scripts/ch00_proof_of_life.py all`. If it passes, your environment works. Then come back here to understand what it checked and why each test matters.

1.1 Why robotics RL fails silently

Let's start with the problem that motivates everything in this chapter.

You train a policy for eight hours. The training loop runs without errors. You evaluate the policy and find a success rate of 0%. Zero. You check the reward curve -- it is flat. You check the logs -- nothing unusual. What went wrong?

The answer could be any of a dozen things: a rendering backend that silently fails, a CUDA driver mismatch that forces CPU training without telling you, a package version

incompatibility that changes the observation format, a reward function that returns the wrong sign. The code *ran*. It just did not *work*.

This is harder to debug than a typical software bug. A web server that crashes is easier to fix than a web server that serves wrong answers. A compiler error is easier to fix than silent data corruption. In RL, the equivalent of silent data corruption is a flat reward curve -- and it can waste days of compute before you notice.

The problem gets worse with robotics. Physics simulators like MuJoCo have system-level dependencies (shared libraries, GPU drivers, rendering backends) that can fail in platform-specific ways. A setup that works on your laptop may break on a remote server. A setup that works today may break after a system update. And because RL training is stochastic by nature -- different random seeds produce different trajectories, different gradient updates, different final policies -- it can be genuinely hard to tell whether a bad result is caused by a software bug, a configuration error, or just bad luck.

The ”runs but doesn't learn” problem

To make this concrete: in our experience developing the code for this book, we encountered a bug where the rendering backend (EGL) failed to initialize but the error was caught and silently swallowed. Training proceeded using a fallback code path that changed the observation normalization. The reward curve looked plausible -- it moved, it did not diverge -- but the success rate stayed at 0% for 500,000 steps. The fix was a two-line change to an environment variable. But finding the bug took an entire day, because nothing *crashed* and the training output looked superficially normal.

Henderson et al. (2018) documented a reproducibility crisis in deep RL that goes beyond individual bugs: many published results could not be replicated, even by the original authors, because the experimental conditions were underspecified. Their paper showed that seemingly minor details -- random seed, network initialization, even the number of parallel environments -- could change the outcome from ”learns successfully” to ”fails completely.”

In our experience, the single most useful thing you can do before training is verify that your environment, rendering stack, training loop, and evaluation protocol all work correctly. Once you have that confidence, a training failure tells you something interesting -- the problem lies in the algorithm, the hyperparameters, or the task itself, not in a broken setup.

This chapter gives you that confidence systematically. By the end, you will have verified the full stack, so you can focus on the interesting problems.

Three questions before you train

Before running any experiment in this book, we find it useful to ask three questions. These come from the mathematician Hadamard, who studied when problems have reliable solutions. The math is not important here -- they translate directly to engineering:

1. **Can this be solved?** Is there a policy that could achieve the goal, given the observation and action spaces? For some tasks, the answer is genuinely uncertain.

A policy with no access to the object's position cannot learn to push it. A 2-layer MLP may not have enough capacity for a high-dimensional visual task.

2. **Is the solution reliable?** Will different random seeds give similar results, or is success a fluke? If you train five times and succeed once, you probably got lucky -- the algorithm is not reliably solving the task.
3. **Is the solution stable?** Will small changes in hyperparameters or environment configuration break it? A solution that only works with a learning rate of exactly 3e-4 and falls apart at 5e-4 is fragile and hard to trust.

If the answer to any of these is ”no” or ”we don't know,” we have work to do before training.

For this chapter, the answers are reassuring. The task (FetchReachDense, where the robot moves its end-effector toward a target) is well within the capability of standard RL algorithms -- even random exploration occasionally reaches the goal. Results are consistent across seeds; the success rate does not vary wildly. And the environment is not sensitive to small configuration changes. But asking these questions explicitly -- even when the answers are easy -- builds a habit that prevents expensive mistakes later.

We will use these three questions as a lightweight checklist at the start of every chapter. In later chapters, where we tackle sparse rewards (Chapter 5) and contact-rich manipulation (Chapter 6), the answers become less obvious -- and the questions become more valuable.

1.2 The task family: goal-conditioned Fetch manipulation

The experiments in this book use the Fetch family of environments from Gymnasium-Robotics. These are simulated robotic manipulation tasks built on the MuJoCo physics engine -- a high-fidelity rigid-body dynamics simulator originally developed at the University of Washington and now maintained by Google DeepMind. A 7-degree-of-freedom (7-DOF) Fetch robotic arm sits on a table and must achieve goals: reaching a target position, pushing an object to a location, or picking up an object and placing it elsewhere.

Let's get a feel for what we are working with. The Fetch arm is a silver industrial manipulator about a meter tall, with seven visible joints connecting chunky links that taper down to a parallel-jaw gripper with rubber finger pads. It sits behind a low table, and its reachable workspace -- the volume where it can operate -- spans roughly 60 x 70 x 20 cm on and above the table surface. When the arm moves, it traces smooth, deliberate arcs; you issue Cartesian velocity commands (”move right, move up”), and an internal inverse-kinematics controller translates those into joint motions. In the simplest task, FetchReach, a small red sphere floats in the air near the table, and success means the fingertips arrive within 5 cm of that sphere -- roughly the width of two fingers side by side. It sounds easy, but getting a learning algorithm to do it reliably from scratch is where the interesting challenges begin.

What makes these tasks interesting for learning is that they are goal-conditioned: the robot receives a desired goal (where to move, where to push the object) that changes every episode. The policy must generalize across goals, not just memorize a single

target. This is a step toward real-world utility -- a robot that can only reach one fixed position is not very useful.

Why this environment family

A reasonable question at this point: why build an entire book around one robot in one simulator? Most RL textbooks rotate between CartPole, Atari, and MuJoCo locomotion -- a different environment each chapter. We take the opposite approach, and we want to be explicit about why.

We chose Fetch for three reasons:

1. **Goal conditioning enables the algorithms we care about.** The central challenge of this book is learning manipulation from sparse binary rewards -- "did you succeed or not?" -- rather than hand-designed distance signals. Solving this requires Hindsight Experience Replay (HER, Chapter 5), which needs an environment that separates `achieved_goal` from `desired_goal` so that failed trajectories can be relabeled as successes for different goals. Fetch environments provide this interface natively. Many popular RL benchmarks (CartPole, Ant, HalfCheetah) do not, which would force us to retrofit goal conditioning rather than teach it.
2. **The difficulty ladder lives inside one family.** Fetch provides five distinct challenges without changing the API:

   - Reaching a target (arm control only, continuous feedback)
   - Reaching with sparse reward (same task, binary signal -- the HER motivation)
   - Pushing an object (contact physics, multi-phase control)
   - Picking and placing (grasp coordination, goals in the air)
   - Pushing from camera images (same task, but raw pixels instead of state vectors)

   Each step adds exactly one new difficulty. When HER improves sparse Push from 0% to 99% success, you know the improvement comes from the algorithm, not from switching to an easier environment. This controlled progression is hard to achieve when hopping between unrelated benchmarks.
3. **Research comparability.** Fetch environments are widely used in the goal-conditioned RL literature (Plappert et al., 2018; Andrychowicz et al., 2017). Our results are directly comparable to published work. When we report 95% success on FetchPush-v4 with SAC+HER, a reader can check that number against the original HER paper and know we are in the right ballpark.

Depth over breadth

This is a deliberate pedagogical choice: we believe you learn more from understanding one task deeply than from running many tasks superficially. The cost is real -- you will not see locomotion, multi-agent coordination, or real hardware in the main text. We address portability in the appendices (Appendix C ports the methodology to PyBullet's PandaGym; Appendix E demonstrates it on NVIDIA Isaac). But the core curriculum stays with Fetch because depth enables something breadth cannot: controlled variable elimination.

When a training run fails in Chapter 5 (sparse rewards), is the problem the algorithm, the reward, the exploration, or the environment? Because you already solved the same environment with dense rewards in Chapters 3-4, you can rule out the environment. When pixel-based training struggles in Chapter 9, you already solved the same task from state vectors -- so you know the task is solvable, and the problem must be in the visual pipeline. With a new environment each chapter, every failure has five possible causes; with one environment family, you isolate variables the way a lab scientist would.

The Fetch environments form a natural difficulty ladder that we will climb throughout the book. Notice that the rightmost column spans all ten chapters -- same robot, same simulator, increasing challenge:

Environment	Task	What changes
`FetchReachDense-v4`	Move end-effector to target	Baseline: dense reward, state vectors
`FetchReach-v4`	Same task, sparse reward	Reward signal: continuous -> binary
`FetchPush-v4`	Push block to target position	Task complexity: contact, multi-phase
`FetchPickAndPlace-v4`	Pick up block, place at target	Coordination: grasp + lift + place
`FetchPush-v4` (pixels)	Push block, from camera images	Observation: 25D vectors -> 84x84 in

All five rows share the same robot, the same action space, and (except for pixels) the same observation structure. The "What changes" column is the key: each row adds exactly one new challenge. By keeping everything else constant, we isolate what matters -- when sparse-reward training fails, the culprit is the reward signal, not a new environment's quirks. When pixel training is slower, the culprit is the observation modality, not different dynamics.

Observations. Every Fetch environment returns a dictionary with three keys:

- `observation` -- the robot's proprioceptive state (joint positions, velocities, end-effector position). For FetchReach, this is a 10-dimensional vector. For environments with objects, it is larger (25D for FetchPush and FetchPickAndPlace) because it includes the object's position, rotation, and velocity.
- `desired_goal` -- where the robot should move its end-effector (or the object, for push/pick-and-place). A 3D position in Cartesian space (x, y, z coordinates in meters).
- `achieved_goal` -- where the end-effector (or object) currently is. Also a 3D position.

This three-part structure is a convention from the Gymnasium `GoalEnv` interface. It will show up in every chapter, so it is worth getting familiar with now. The separation of `achieved_goal` and `desired_goal` from the main observation is what makes goal conditioning explicit -- the environment literally tells the policy "here is where you are, here is where you should be."

Actions. Actions are 4-dimensional vectors: three components for the desired Cartesian velocity of the end-effector (dx, dy, dz) and one for gripper control (open/close), with each component bounded to [-1, 1]. For reaching tasks, the gripper dimension does not matter -- the robot is just moving its arm. For push and pick-and-place, the

gripper becomes essential: the robot must learn to close the gripper around an object and open it at the target location.

The Cartesian action space is an important design choice, because it means the learning algorithm does not need to figure out how joint torques map to end-effector motion -- an inverse kinematics controller handles that internally. The policy therefore operates at a higher level of abstraction: "move the hand right and close the gripper." This makes the learning problem tractable for the algorithms we use.

Rewards. Fetch environments come in two flavors:

- *Dense reward* (e.g., `FetchReachDense-v4`): the reward is the negative Euclidean distance between `achieved_goal` and `desired_goal`. Closer is better. The reward is always negative or zero, with 0 meaning the goal is perfectly achieved.
- *Sparse reward* (e.g., `FetchReach-v4`): the reward is -1 if the goal is not achieved and 0 if it is. "Achieved" means the distance is below a threshold (5 cm for Reach).

Dense rewards give the learning algorithm continuous feedback -- "you're getting warmer." Sparse rewards give a binary signal -- "success or failure." Sparse rewards are harder to learn from but more natural: in real robotics, you often know only whether the task succeeded, not by how much you missed. Learning from sparse rewards is one of the central challenges of this book, and we will tackle it directly in Chapter 5 using Hindsight Experience Replay (HER -- an algorithm that turns failed attempts into useful training data by asking "what goal would this attempt have achieved?").

For this chapter, we use `FetchReachDense-v4` -- the easiest variant. We are not trying to learn anything interesting yet. We are verifying that the machinery works.

> **NOTE:** We formalize dense and sparse rewards mathematically in Chapter 2. For now, the intuition is sufficient: dense = continuous distance feedback, sparse = binary success/failure signal.

1.3 The experiment contract

Every chapter in this book produces concrete artifacts. Not screenshots, not "it looked like it was working," not a training curve in TensorBoard (a web-based visualization dashboard for monitoring training runs) that you squint at and decide looks "good enough" -- files on disk that you can inspect, share, and reproduce.

Here is the contract:

Artifact	Format	Example
Checkpoints	Stable Baselines 3 (SB3) `.zip` + `.meta.json`	`checkpoints/ppo_FetchReach`
Evaluation reports	JSON	`results/ch02_ppo_fetchreac`
TensorBoard logs	Event files	`runs/ppo/FetchReachDense-v`
Videos	MP4	`videos/ppo_FetchReachDense`

The `ppo` in the example filenames refers to PPO (Proximal Policy Optimization), the algorithm used for training -- we introduce PPO in Chapter 3. The two most important columns are "Format" and "Example," because they make the contract concrete:

checkpoints are files with known paths and machine-readable metadata, and evaluation reports are JSON -- not prose, not impressions, but structured data with success rates, episode returns, goal distances, and seed counts. When this book says "94% success rate," there is a JSON file that contains that number, and you can verify it yourself.

Why provenance matters. Henderson et al. (2018) documented a reproducibility crisis in deep RL: many published results could not be replicated, even by the original authors, because the experimental conditions were underspecified. Random seeds, hyperparameters, library versions, and hardware all matter. A single random seed can be the difference between 95% success and 40% success on the same algorithm with the same hyperparameters. A result with provenance -- one you can trace back to a specific command, seed, and environment -- is worth far more than one without it.

Our defense is simple: every experiment in this book is defined by a single command, produces versioned artifacts, and records the conditions under which it ran. The `.meta.json` file alongside each checkpoint captures the algorithm, environment, seed, step count, and library versions. The evaluation JSON records exactly which checkpoint was evaluated, on which environment, with which seeds, and whether the policy was deterministic. If you want to know how we got a number, you can find the exact command in the chapter, run it yourself, and compare.

We find it helpful to hold ourselves to a simple standard: every number we report should trace back to a file on disk. If we can point to the JSON, the result is real. If we cannot, we treat it as unverified.

The training and evaluation CLIs. Every chapter uses the same two entry points:

- `train.py` -- trains a policy. Key flags: `--algo` (PPO, SAC, or TD3 -- algorithm names we introduce in Chapters 3-4), `--env` (environment ID), `--seed`, `--total-steps`, and `--her` (enables Hindsight Experience Replay for off-policy algorithms). Produces a checkpoint `.zip` and a `.meta.json`.
- `eval.py` -- evaluates a saved checkpoint. Key flags: `--ckpt` (path to the checkpoint), `--env`, `--n-episodes`, `--seeds`, `--deterministic`, and `--json-out`. Produces a JSON evaluation report.

You do not need to remember these flags now -- each chapter provides the exact commands. The point is that the interface is consistent: same CLI, same artifact format, same evaluation protocol. When you learn to read evaluation JSON in Chapter 3, that skill applies to every chapter that follows.

This chapter's contract is lighter than later chapters (we are smoke-testing, not training a real policy), but the pattern starts here. You will see it in every chapter that follows.

1.4 Setting up the environment

Prerequisites

Before you start, you need two things on your host machine:

1. **Docker.** Verify with `docker --version`. Any recent version (20+) works.

2. **NVIDIA Container Toolkit** (Linux with GPU only). Verify with `docker run --rm --gpus all nvidia/cuda:12.0-base nvidia-smi`. You should see your GPU listed.

On Mac, Docker Desktop is all you need -- there is no GPU passthrough, and the workflow uses CPU inside a Linux container.

If either check fails, consult your system administrator or Docker's installation documentation. This book does not cover Docker installation, but the above two commands are a fast way to confirm you are ready.

Docker as environment specification

We use Docker containers for all experiments, primarily for reproducibility.

Here is the scenario we are trying to prevent: you train a policy, achieve 94% success, and send the code to a colleague. They install the dependencies, run the same command, and get 12% success -- or a crash. The difference? Their MuJoCo version is slightly different. Or their CUDA toolkit does not match. Or a transitive dependency upgraded since you pinned yours. These problems are invisible and maddening.

A Docker container packages your code, its dependencies, the operating system libraries, and the configuration into a single artifact. The container is identified by a content-addressable hash, so "I ran image abc123" is unambiguous. If the container runs on your machine, it runs on your colleague's machine, on a cloud server, or on a DGX workstation -- with the same behavior.

The two-layer architecture

Our container has two layers, and the separation is deliberate:

- **Base layer.** The NVIDIA PyTorch image (`nvcr.io/nvidia/pytorch:25.12-py3`) provides CUDA, cuDNN, and PyTorch pre-configured and tested by NVIDIA. This is the heavyweight layer -- several gigabytes, carefully version-matched. It rarely changes between experiments.
- **Project layer.** On top of the base, our `docker/Dockerfile` adds the project-specific dependencies: MuJoCo rendering libraries (`libegl1`, `libosmesa6` for headless rendering), `gymnasium`, `gymnasium-robotics`, `stable-baselines3`, `tensorboard`, and `imageio`. These are lighter, and they change when we update our experiment code.

The benefit of this separation is practical: if you change a Python dependency, only the lightweight project layer rebuilds. You do not need to re-download the multi-gigabyte NVIDIA base.

> **NOTE:** For strict reproducibility, the Docker image digest (the content hash) is the true specification -- it freezes the entire dependency tree. If you need to capture exact versions for a paper or report, run `pip freeze` inside the container and save the output. The `requirements.txt` file specifies minimum version constraints (e.g., `stable-baselines3>=2.4.0`), which is sufficient for the experiments in this book.

The dev.sh entry point

The `docker/dev.sh` script is the single entry point for everything in this book. It handles building the image, launching the container, mounting your code, setting up a Python virtual environment, and dropping you into a shell. You do not need to learn Docker commands -- `docker/dev.sh` wraps them all.

```
# Enter an interactive development shell
bash docker/dev.sh

# Or run a single command
bash docker/dev.sh python scripts/ch00_proof_of_life.py all
```

When you run `docker/dev.sh`, it:

1. Checks for the `robotics-rl:latest` image and builds it if missing (this takes a few minutes on first run)
2. Launches a container with GPU access (`--gpus all` on NVIDIA systems)
3. Mounts your repository at `/workspace` inside the container, so changes you make to code on the host are instantly visible inside the container and vice versa
4. Runs as your host user ID (so files created inside have your ownership, not root's)
5. Creates a `.venv` virtual environment and installs dependencies from `requirements.txt`
6. Activates the environment and runs your command (or starts a shell)

Every command in this book that starts with `bash docker/dev.sh ...` follows this pattern. You will type it many times. If you prefer, you can enter an interactive shell with `bash docker/dev.sh` and then run commands directly inside the container.

> **TIP:** Three environment variables control headless rendering: `MUJOCO_GL=egl` selects the hardware-accelerated EGL backend, `PYOPENGL_PLATFORM=egl` configures PyOpenGL to match, and `NVIDIA_DRIVER_CAPABILITIES=al` enables full GPU access in the container. The `docker/dev.sh` script sets all of these automatically. You should not need to set them yourself unless you are debugging rendering issues (see section 1.7).

The virtual environment inside the container

You might wonder why we need a virtual environment inside an already-isolated container. There are practical reasons: it lets us do editable installs (`pip install -e .`) for rapid development, lets project dependencies shadow container defaults when needed, and keeps the dependency specification explicit in `requirements.txt` rather than scattered across the Dockerfile.

On NVIDIA systems, the venv is created with `--system-site-packages`, which lets it inherit PyTorch from the base container. This avoids reinstalling the large, CUDA-specific PyTorch package. On Mac, where there is no CUDA, the venv installs CPU PyTorch from scratch.

The first time you run `docker/dev.sh`, it installs all dependencies -- this adds a minute

or two. Subsequent runs skip installation (it hashes `requirements.txt` and only reinstalls when the file changes).

SIDEBAR: Running on Mac (Apple Silicon)

The same `docker/dev.sh` command works on Mac. The script auto-detects your platform (`uname -s` = Darwin for Mac) and uses an appropriate Docker image (`robotics-rl:mac` built from `docker/Dockerfile.mac`, which uses `python:3.11-slim` as its base instead of the NVIDIA image).

Key differences from Linux/NVIDIA:

	Linux/NVIDIA	Mac
Compute	CUDA GPU	CPU
Rendering	EGL (hardware-accelerated)	OSMesa (software)
Throughput	~600 fps	~60-100 fps
Docker image	`robotics-rl:latest`	`robotics-rl:mac`
Base image	`nvcr.io/nvidia/pytorch`	`python:3.11-slim`

The 6-10x throughput gap is mostly about faster CPUs and more cores on Linux machines, not GPU acceleration. For state-based RL (Chapters 1-8), both platforms are CPU-bound: MuJoCo physics dominates, and a 256x256 MLP processing a 25D observation vector completes in microseconds regardless of device. Mac is fully viable through Chapter 8 -- training runs that take seconds on DGX take tens of minutes on Mac, which is fine for learning and iteration. The GPU becomes important at Chapter 9 (pixel observations, where a CNN processes 84x84 images every step) and essential at Appendix E (Isaac Lab, where physics itself runs on the GPU). For Chapter 9, RAM is usually the binding constraint rather than GPU speed -- see that chapter for buffer-size guidance.

Apple's MPS (Metal Performance Shaders) backend exists for PyTorch but cannot work inside Docker, since Docker on Mac runs a Linux VM that has no access to the Metal framework. CPU is the reliable default.

If you encounter out-of-memory errors on Mac, increase Docker Desktop's memory allocation to at least 8 GB (Settings -> Resources -> Memory -> Apply & restart).

1.5 Build It: Inspecting what the robot sees

Before running the full pipeline, let's look under the hood. This section teaches you to inspect the environment directly -- the observation structure, the reward computation, and the success signal. These are the building blocks that every later chapter depends on.

In later chapters, Build It sections will have you implementing algorithms from scratch -- writing loss functions, replay buffers, and update loops in raw PyTorch. This chapter's Build It is lighter because there is no algorithm to implement yet. Instead, you will learn to talk to the environment: query its observations, manually compute rewards,

and check success conditions. These are diagnostic skills that come in handy whenever something goes wrong in a later chapter.

1.5.1 The observation dictionary

The first thing to understand about any RL environment is what the agent sees. In Fetch environments, the agent sees a dictionary, not a flat vector. This is unusual compared to classic RL environments like CartPole, where the observation is a simple array.

Create a Fetch environment and inspect what it gives you:

```
import gymnasium as gym
import gymnasium_robotics  # registers Fetch envs

env = gym.make("FetchReachDense-v4")
obs, info = env.reset(seed=42)

print(type(obs))           # <class 'dict'>
print(sorted(obs.keys()))  # ['achieved_goal', 'desired_goal', 'observation']

print(obs["observation"].shape)    # (10,)
print(obs["achieved_goal"].shape)  # (3,)
print(obs["desired_goal"].shape)   # (3,)
```

The observation dictionary has three arrays:

- `obs["observation"]` (shape `(10,)`) -- the robot's proprioceptive state: gripper position (3D), gripper linear velocity (3D), finger positions (2D), and finger velocities (2D).
- `obs["achieved_goal"]` (shape `(3,)`) -- where the end-effector currently is in 3D Cartesian space.
- `obs["desired_goal"]` (shape `(3,)`) -- where we want the end-effector to be.

Notice that `achieved_goal` is redundant with the first three elements of `observation` (both report the end-effector position). This redundancy is by design -- it lets the goal-conditioned interface work uniformly across environments. For pushing and pick-and-place tasks, `achieved_goal` will be the *object* position, not the end-effector position, while the robot's own position stays in `observation`.

What the values mean physically. The numbers in these arrays are not abstract -- they correspond to real physical quantities in the simulated world. The goal positions are in meters relative to the MuJoCo world frame. As Figure 1.1 shows, the Fetch arm sits on a table with a workspace spanning roughly x in [1.0, 1.6], y in [0.4, 1.1], and z in [0.4, 0.6] -- a roughly 60 cm x 70 cm x 20 cm box on and above the table. The velocities in `observation` are in meters per second. Understanding these scales will help you debug later: if you ever see goal positions at (0, 0, 0) or velocities in the thousands, something is wrong with the environment state.

> **Checkpoint.** Run the code above inside the container (`bash docker/dev.sh python -c "..."`). Verify that you get three keys, that shapes match `(10,)`,

(3,), and (3,), and that values are finite floating-point numbers (not NaN, not inf). Check that the goal positions are within the workspace bounds described above. If shapes differ, check your gymnasium-robotics version.

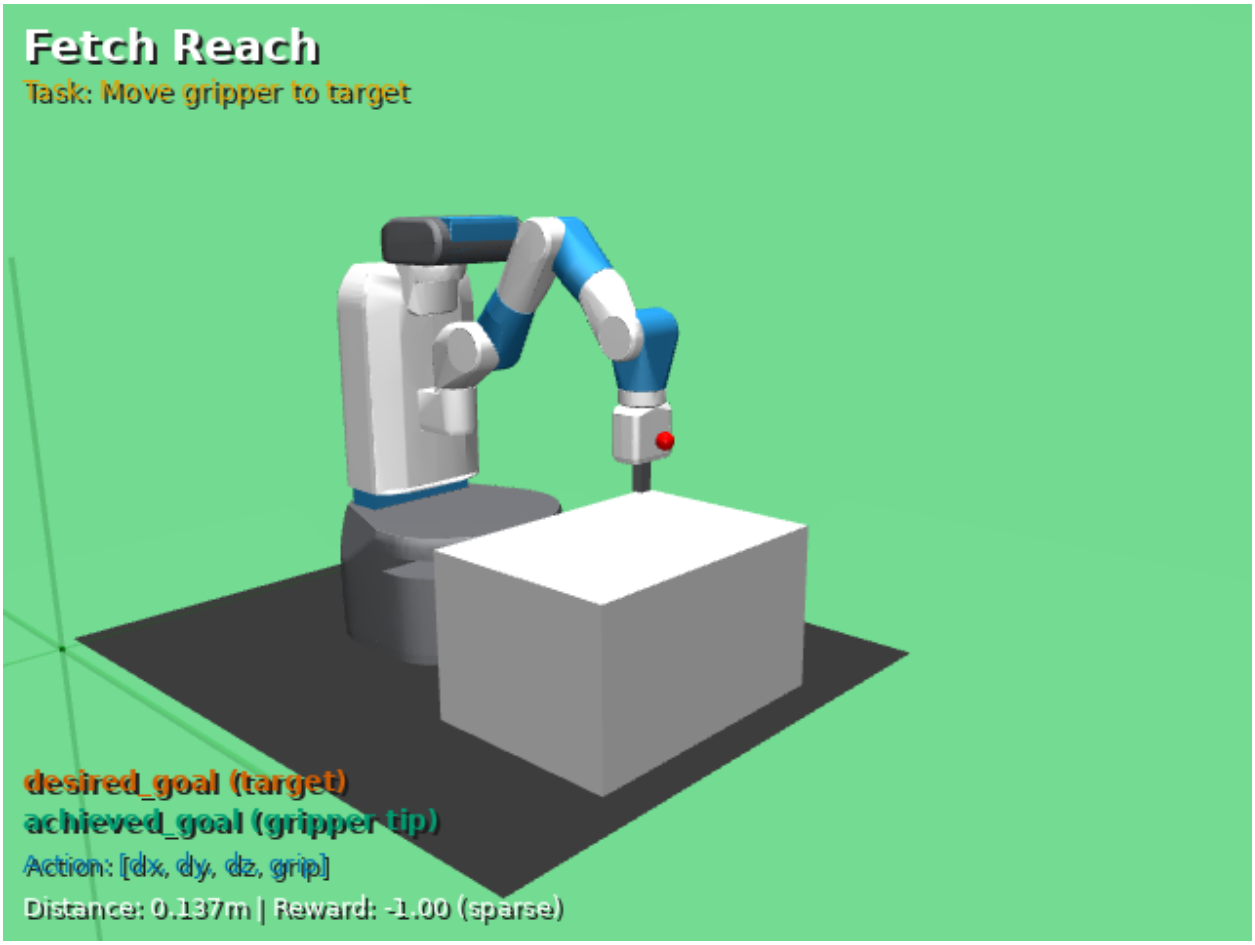

Figure 1.1: The FetchReach environment after `env.reset()`. The red sphere marks the desired goal -- the 3D position the end-effector must reach. The gripper's current position is the achieved goal. The observation dictionary separates these into `desired_goal` and `achieved_goal`, making goal conditioning explicit. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchReach-v4`. FetchReachDense-v4 is visually identical; only the reward differs.)

1.5.2 Manual compute_reward: the critical invariant

Every Fetch environment exposes a `compute_reward` method that takes an achieved goal, a desired goal, and an info dictionary, and returns a reward. We access it via `env.unwrapped.compute_reward(...)` because `gym.make()` wraps the environment in several layers (time limits, order enforcement) that do not expose `compute_reward` directly -- `.unwrapped` reaches through to the base Fetch environment. This method is the foundation of Hindsight Experience Replay (HER), which we introduce in Chapter 5. HER works by asking: ”what goal would this trajectory have achieved?” and

recomputing rewards accordingly. If `compute_reward` does not match the reward that `env.step()` returns, HER learns from incorrect data.

Let's verify this invariant directly:

```
import numpy as np

env = gym.make("FetchReachDense-v4")
obs, info = env.reset(seed=42)

# Take a random action
action = env.action_space.sample()
next_obs, step_reward, terminated, truncated, info = env.step(action)

# Recompute the reward manually
manual_reward = env.unwrapped.compute_reward(
    next_obs["achieved_goal"],
    next_obs["desired_goal"],
    info,
)

print(f"Step reward:   {step_reward:.6f}")
print(f"Manual reward: {manual_reward:.6f}")
print(f"Match: {np.isclose(step_reward, manual_reward)}")
```

Expected output (your exact values will differ -- the important result is `Match: True`):

```
Step reward:   -1.058329
Manual reward: -1.058329
Match: True
```

The specific reward value depends on the random action sampled, which varies across library versions and seeds. What matters is that the two values match exactly -- we call this the critical invariant: `compute_reward(achieved_goal, desired_goal, info)` must equal the reward from `env.step()`, since every chapter that uses HER depends on it.

Why does this matter so much? When HER relabels a failed trajectory -- "you did not reach the goal at position (1.3, 0.7, 0.5), but you did reach position (1.2, 0.6, 0.42)" -- it needs to recompute the reward as if that alternate position had been the goal all along. It does this by calling `compute_reward(achieved_goal=actual_position, desired_goal=relabeled_goal, info)`. If this function returns a different value than `env.step()` would have returned for the same state, then HER is training on incorrect reward labels. The policy learns from corrupted data, and training may silently fail or converge to a bad policy.

> **Checkpoint.** Run this check with several different seeds and actions. The rewards should always match exactly (not approximately -- exactly). If they do not, there is a version incompatibility between gymnasium and gymnasium-robotics that may cause problems in Chapter 5.

For dense rewards, the manual reward equals the negative Euclidean distance between the achieved goal and the desired goal:

```
distance = np.linalg.norm(
    next_obs["achieved_goal"] - next_obs["desired_goal"]
)
print(f"Negative distance: {-distance:.6f}")
print(f"Dense reward:      {step_reward:.6f}")
```

These should also match exactly. The dense reward equals `-distance` -- nothing more.

You can also verify the sparse reward variant. If you create `FetchReach-v4` (without "Dense") and perform the same check, the reward will be either -1.0 (goal not achieved) or 0.0 (goal achieved). The same `compute_reward` invariant holds for sparse rewards -- the function just returns a different value:

```
sparse_env = gym.make("FetchReach-v4")
sparse_obs, _ = sparse_env.reset(seed=42)
action = sparse_env.action_space.sample()
next_obs, reward, _, _, info = sparse_env.step(action)
print(f"Sparse reward: {reward}")  # -1.0 (most likely)
```

1.5.3 Parsing the success signal

The `info` dictionary returned by `env.step()` contains an `is_success` field that indicates whether the goal was achieved. For FetchReach, "achieved" means the distance between the end-effector and the target is below 5 cm (0.05 meters):

```
distance = np.linalg.norm(
    next_obs["achieved_goal"] - next_obs["desired_goal"]
)
print(f"Distance: {distance:.4f} m")
print(f"Success:  {bool(info['is_success'])}")
print(f"Threshold: 0.05 m")
print(f"Below threshold: {distance < 0.05}")
```

The success signal is what we measure during evaluation. When later chapters report "94% success rate," they mean that across many episodes, `info['is_success']` was True at the end of 94% of them. This is the metric that matters most -- not the reward, not the return, not the loss value, but the fraction of episodes where the robot actually achieved the goal. A high return with a low success rate means the robot is getting close but not close enough; a high success rate with a modest return means the robot succeeds but takes a roundabout path.

Notice how the three things we have inspected fit together: observations tell the policy where it is and where it should be, rewards give the policy a training signal based on that information (dense: how close? sparse: success or failure?), and the success metric is the binary outcome we ultimately care about. In other words, the reward drives learning, while the success signal measures whether learning worked. In dense-reward environments, a well-trained policy will have both high reward (close to 0) and

high success rate (close to 100%). In sparse-reward environments, the relationship is starker: the reward is -1 until the moment of success, then 0. This makes dense rewards more informative for learning but sparse rewards more honest about what we actually want.

> **Checkpoint.** On a random action immediately after reset, the distance is typically 0.05-0.15 meters and `is_success` is usually False (the robot's initial position is rarely on top of the goal). If `is_success` is True on the first step with a random action, something is unusual -- check that the environment is creating diverse goal positions.

1.5.4 The bridge: Build It meets Run It

We have now verified three things by hand:

1. The observation dictionary has the expected structure and shapes
2. `compute_reward(ag, dg, info)` matches the reward from `env.step()`
3. The success signal corresponds to goal distance below a threshold

Each of these checks connects to the production pipeline in specific ways.

When SB3 creates a PPO or SAC model with `MultiInputPolicy`, it reads the observation space to determine the input structure -- the same `Dict` with three `Box` entries that you inspected in section 1.5.1. If the shapes were wrong or the keys were missing, model creation would fail silently or produce a network with the wrong architecture. By inspecting the observation yourself, you know exactly what the model will see.

When HER (Chapter 5) relabels goals, it calls `compute_reward` with different goal values and trusts the returned reward to be consistent with what `env.step()` would have returned. The check in section 1.5.2 verifies that this trust is warranted. If you ever upgrade gymnasium-robotics and want to confirm that nothing broke, this is the check to run.

When `eval.py` reports a success rate, it counts how many episodes ended with `info['is_success'] == True` -- the same signal you examined in section 1.5.3. If the threshold were wrong, or if `is_success` did not correspond to the distance you measured, the evaluation numbers would be meaningless.

In later chapters, the Build It sections are more substantial -- you will implement entire algorithms from scratch. But the principle is the same: understand the pieces by hand before trusting the production code to assemble them correctly.

1.6 Run It: The proof-of-life pipeline

Now we run the full verification sequence. The script `scripts/ch00_proof_of_life.py` implements four tests, each verifying a necessary condition for the experiments that follow.

```
------------------------------------------------------------
EXPERIMENT CARD: Proof of Life
------------------------------------------------------------
```

```
Environment:  FetchReachDense-v4 (smoke test only)
Fast path:    ~5 min (GPU) / ~10 min (CPU)

Run command:
  bash docker/dev.sh python scripts/ch00_proof_of_life.py all

Artifacts:
  smoke_frame.png         (headless render validation)
  ppo_smoke.zip           (training loop validation)
  ppo_smoke.meta.json     (smoke run metadata: versions, seed, device)

Success criteria:
  smoke_frame.png exists, non-empty, shows Fetch robot
  ppo_smoke.zip exists, loadable by PPO.load()
  ppo_smoke.meta.json exists, contains env_id and versions
  All 4 subtests pass (gpu-check, list-envs, render, ppo-smoke)
------------------------------------------------------------
```

The four-test verification sequence

The `all` subcommand runs these tests in order. If a test fails, diagnose and fix it before proceeding -- later tests depend on earlier ones.

Test 1: GPU check (`gpu-check`)

Verifies that PyTorch can see the GPU via `torch.cuda.is_available()`. On a Linux system with NVIDIA hardware, this should report CUDA available with the device name and count. You should see something like:

```
OK: CUDA available -- 1 device(s), primary: NVIDIA A100-SXM4-80GB
```

On Mac, this correctly reports "CUDA not available" -- training proceeds on CPU, which is expected and functional:

```
WARN: CUDA not available; training will use CPU (this is expected on Mac)
```

This test always passes (it warns rather than fails on Mac or CPU-only systems) because CPU-only operation is valid. But on a system where you expect a GPU, treat a "not available" warning as a real problem: check that Docker was invoked with `--gpus all` and that the NVIDIA Container Toolkit is installed.

We verify GPU access early because later chapters need it -- but not all chapters, and not for the same reasons:

Chapters	Workload	CPU viable?
1-8	State-based RL (MuJoCo + small MLPs on 25D vectors)	Yes (~60-100 fps Mac, ~600 fp
9	Pixel-based RL (CNN on 84x84x12 images)	Slow but possible
App. E	Isaac Lab (GPU-parallel PhysX)	No

For Chapters 1-8, the bottleneck is MuJoCo physics simulation, which runs on CPU

regardless of platform. Training runs that take minutes on a Linux GPU machine take tens of minutes on a Mac laptop -- not days.

Test 2: Fetch environment registry (`list-envs`)

Imports `gymnasium_robotics` and lists all registered Fetch environments. You should see a list that includes:

```
FetchPickAndPlace-v3
FetchPickAndPlaceDense-v3
FetchPush-v3
FetchPushDense-v3
FetchReach-v3
FetchReachDense-v3
FetchReach-v4
FetchReachDense-v4
...
```

The `-v3` and `-v4` variants are both present; we use `-v4` throughout this book (the latest version at time of writing). If no Fetch environments appear, `gymnasium-robotics` is not installed correctly.

Test 3: Headless rendering (`render`)

Creates a Fetch environment with `render_mode="rgb_array"`, calls `env.render()`, and saves the frame as `smoke_frame.png`. This tests the entire rendering pipeline: MuJoCo physics initialization, scene construction, camera setup, and offscreen rendering via EGL or OSMesa.

Why is rendering non-trivial? On a typical workstation or laptop, rendering uses the display server (X11 on Linux, the window system on Mac) to manage the OpenGL context. But DGX systems and cloud servers are headless -- they have no monitor attached and no display server running. Rendering must happen entirely offscreen, which requires either EGL (using the GPU's rendering capability directly, without a display) or OSMesa (a software-only renderer that produces images using the CPU). Both approaches have system library requirements that can fail silently.

The script implements a fallback chain: it first tries EGL (hardware-accelerated, preferred on NVIDIA systems), then OSMesa (software rendering, slower but more compatible). The fallback works by re-executing the entire script process with different environment variables -- so if EGL fails, you may see the script's startup output appear twice in your terminal. This is the re-exec mechanism switching backends, not an error.

On success, you see:

```
OK: wrote /workspace/smoke_frame.png
```

Test 4: PPO smoke training (`ppo-smoke`)

Runs PPO (Proximal Policy Optimization, a policy gradient method we derive and implement from scratch in Chapter 3) for 50,000 timesteps on `FetchReachDense-v4` with 8 parallel environments and saves a checkpoint as `ppo_smoke.zip`. This is enough to verify that the entire training pipeline (environment interaction, gradient computation,

parameter updates, checkpoint saving) works end-to-end, though far too short to learn a good policy.

Why 50,000 steps and not 1,000,000? Because this is a smoke test, not a training run. We want to verify the machinery in under 5 minutes, not produce a useful policy. A full training run for FetchReachDense takes about 500,000 steps (Chapter 3). For now, we just need the loop to execute without crashing.

We use PPO for this smoke test rather than SAC (Soft Actor-Critic, an off-policy method we introduce in Chapter 4) because PPO is simpler and surfaces errors faster. It has fewer moving parts -- no replay buffer, no twin critics, no entropy tuning. If PPO runs, we can be confident the core infrastructure is sound. SAC-specific issues can be debugged separately when we get to Chapter 4.

> **WARNING:** If the PPO smoke test hangs or takes much longer than expected, check whether you are accidentally running on CPU when you expected GPU. The script's GPU check output at the top tells you which device is in use. On CPU, 50,000 steps may take 5-10 minutes; on GPU, about 1-2 minutes.
>
> **NOTE:** Low GPU utilization (~5-10%) during training is expected and normal. The bottleneck is CPU-bound MuJoCo physics simulation, not GPU-bound neural network operations. With small networks and small batch sizes, the GPU finishes its work in microseconds and waits for the CPU. This is the nature of RL with physics simulators, not a problem to solve.

Interpreting the artifacts

After `all` completes, you should have three new files in your repository root:

`smoke_frame.png` -- Open this file. You should see the Fetch robot arm on a table with a target marker (a small red sphere floating in the air near the arm). The table surface, the robot's silver links, and the red target should all be clearly visible. If the image exists but is black or empty, the rendering pipeline has a partial failure -- check the rendering backend output in the terminal. If it looks correct, headless rendering works.

This image is your visual confirmation that MuJoCo is simulating the robot correctly and that the rendering pipeline can turn the physics state into pixels. In later chapters, you will generate evaluation videos using the same pipeline.

`ppo_smoke.zip` -- This is a Stable Baselines 3 checkpoint containing the neural network weights and optimizer state, along with the observation/action space schema needed to load the model. You can verify it is loadable:

```
from stable_baselines3 import PPO
model = PPO.load("ppo_smoke.zip")
print(f"Policy type: {type(model.policy).__name__}")
print(f"Observation space: {model.observation_space}")
print(f"Action space: {model.action_space}")
```

The output should look like:

```
Policy type: MultiInputPolicy
```

```
Observation space: Dict('achieved_goal': Box(-inf, inf, (3,), ...),
  'desired_goal': Box(-inf, inf, (3,), ...),
  'observation': Box(-inf, inf, (10,), ...))
Action space: Box(-1.0, 1.0, (4,), float32)
```

A few things to notice here. The policy is a `MultiInputPolicy` -- not a `MlpPolicy` -- because observations are dictionaries, not flat vectors. SB3 handles the dictionary structure internally by processing each key separately and concatenating them before feeding into the neural network. The observation space is a `Dict` with three `Box` entries matching the shapes we saw in section 1.5. The action space is a `Box` with shape `(4,)` and bounds `[-1, 1]`, matching the Cartesian velocity + gripper action we described in section 1.2.

This checkpoint is not worth evaluating for performance -- it trained for only 50,000 steps, far too few to learn useful behavior on any task. Its purpose is to prove the loop runs, not that it learns; actual learning starts in Chapter 3.

`ppo_smoke.meta.json` -- A small metadata file capturing the environment ID, seed, training step count, device, and library versions used for the smoke run. Later chapters produce the same `.meta.json` alongside checkpoints in `checkpoints/`.

The dependency chain

The four tests form a logical dependency chain:

`GPU check -> Fetch env registry -> Headless rendering -> Training loop`

Each test assumes the previous ones work. Rendering depends on MuJoCo initializing correctly (Test 2). Training depends on rendering being at least attempted (the training test disables rendering, but it still needs MuJoCo and Gymnasium working). And training performance depends on the compute environment (Test 1) -- for state-based chapters (2-8), CPU is adequate; for pixel chapters (9+), GPU helps but RAM is often the binding constraint.

When something fails, diagnose in order: if rendering fails, check Test 2 first, since MuJoCo itself may not be initializing; if training is unexpectedly slow, check Test 1, since CUDA may not be available. The `all` subcommand runs the tests sequentially, so the output makes it clear which test failed.

What ”proof of life” means

After these four tests pass, you know that the container has access to the GPU (or is correctly falling back to CPU), that MuJoCo and Gymnasium-Robotics are installed and functional, that headless rendering produces valid images, and that the full training loop (env -> policy -> training -> checkpoint) executes without error.

Together with the Build It checks from section 1.5, you also know that observations have the expected dictionary structure and shapes, that `compute_reward` matches `env.step()` (the critical invariant), and that the success signal correctly reflects goal distance.

This is what "alive" means: the environment can produce valid results. Producing good results -- that is what the rest of the book is for.

SIDEBAR: Regenerating figures and videos

Every figure in this book is generated from code -- no hand-drawn diagrams, no screenshots from external tools. This means you can regenerate any figure yourself. The figures you see (annotated environment screenshots, reward diagrams, learning curves) are produced by:

```
bash docker/dev.sh python scripts/capture_proposal_figures.py all
```

This creates annotated PNGs in the `figures/` directory. Individual subcommands are available: `env-setup` for environment screenshots, `reward-diagram` for reward function plots, `ppo-clipping` for the PPO clipping diagram, and `ppo-demo-curve` for learning curves.

For evaluation videos showing trained policies in action, use:

```
bash docker/dev.sh python scripts/generate_demo_videos.py \
  --ckpt checkpoints/ppo_FetchReachDense-v4_seed0.zip \
  --out videos/ppo_FetchReachDense-v4_seed0 \
  --gif
```

This produces `videos/ppo_FetchReachDense-v4_seed0.mp4` and `videos/ppo_FetchReachDense-v4_seed0.gif`. We use these to verify that trained policies behave as expected -- not just that the numbers look right, but that the robot actually moves to the target. Later chapters provide specific video commands alongside their Experiment Cards.

1.7 What can go wrong

Here are the most common failures and how to fix them. We have encountered all of these during development. The list is roughly ordered by how early in the pipeline the failure occurs -- Docker issues first, then rendering, then training. If you hit a problem not listed here, the most productive diagnostic approach is to run the four tests individually (`gpu-check`, `list-envs`, `render`, `ppo-smoke`) and identify which one fails.

"Permission denied" when running Docker

Symptom. `docker: Got permission denied while trying to connect to the Docker daemon socket.`

Cause. Your user is not in the `docker` group.

Fix. Run `sudo usermod -aG docker $USER`, then log out and back in. Alternatively, prefix Docker commands with `sudo` (but this can cause file ownership issues).

”I have no name!” in the container shell

Symptom. The shell prompt shows `I have no name!@<container-id>`.

Cause. The container runs as your numeric UID (to match file ownership), and that UID has no entry in the container's `/etc/passwd`.

Impact. None. This is cosmetic. File permissions, training, and everything else work correctly. You can safely ignore it.

EGL initialization failure

Symptom. Errors mentioning `gladLoadGL`, `eglQueryString`, or `libEGL.so`.

Cause. The EGL rendering library is missing or the GPU driver does not expose EGL support.

Fix. The proof-of-life script automatically falls back to OSMesa. If you see the script output appear twice, that is the fallback mechanism -- not an error. If both EGL and OSMesa fail, check that `libegl1` and `libosmesa6` are installed in the container image. Rebuilding the image with `bash docker/build.sh` usually resolves this.

No Fetch environments found

Symptom. The `list-envs` test shows no output, or `gym.make("FetchReachDense-v4")` raises `EnvironmentNameNotFound`.

Cause. The `gymnasium-robotics` package is not installed.

Fix. Inside the container, check with `pip list | grep gymnasium`. You should see both `gymnasium` and `gymnasium-robotics`. If either is missing, run `pip install gymnasium-robotics` or rebuild the image.

CUDA not available (on a system with a GPU)

Symptom. The `gpu-check` test reports ”CUDA not available” on a machine that has an NVIDIA GPU.

Cause. Either Docker was not invoked with `--gpus all`, or the NVIDIA Container Toolkit is not installed, or there is a driver mismatch between the host and the container.

Fix. First, verify the NVIDIA runtime works at all:

```
docker run --rm --gpus all nvidia/cuda:12.0-base nvidia-smi
```

If that command fails, the NVIDIA Container Toolkit is not installed or not configured. Follow the installation instructions from NVIDIA's documentation for your distribution. If `nvidia-smi` works but PyTorch still cannot see the GPU, there may be a CUDA version mismatch -- the container's CUDA toolkit version must be compatible with the host driver version. Run `nvidia-smi` on the host to check the driver version and CUDA compatibility.

On Mac, ”CUDA not available” is expected and correct -- training uses CPU. This is not a problem to fix.

PPO smoke training crashes with shape/dtype errors

Symptom. Errors about tensor shapes or dtypes during training.

Cause. Usually a version mismatch between `stable-baselines3`, `gymnasium`, and `gymnasium-robotics`. The observation space handling changed between major versions.

Fix. Delete `.venv` and re-run `bash docker/dev.sh` to recreate the environment with correct versions. Check that `requirements.txt` specifies compatible versions.

Dependencies reinstall every time

Symptom. `docker/dev.sh` reinstalls packages on every run, adding several minutes of overhead.

Cause. The hash file `.venv/.requirements.sha256` is missing or corrupted. The script uses this hash to detect when `requirements.txt` has changed; if the file is missing, it assumes dependencies need updating.

Fix. Delete `.venv` entirely (`rm -rf .venv`) and re-run `docker/dev.sh`. The venv will be recreated from scratch and the hash file will be written correctly. Subsequent runs should skip installation.

Docker build fails or times out

Symptom. `docker/build.sh` (or the automatic build inside `docker/dev.sh`) fails with network errors, timeout errors, or ”unable to pull” messages.

Cause. The base image (`nvcr.io/nvidia/pytorch:25.12-py3`) is large (several GB) and hosted on NVIDIA's container registry, which can be slow or require authentication for some images. Alternatively, `pip install` during the build may fail if your network blocks PyPI.

Fix. For network issues, retry the build -- transient failures are common with large downloads. If the NVIDIA registry requires authentication, run `docker login nvcr.io` first (a free NVIDIA developer account is sufficient). If pip install fails, check that your network allows HTTPS traffic to pypi.org. On corporate networks with proxy servers, you may need to configure Docker's proxy settings.

> **TIP:** If the build succeeds once, the image is cached locally and you will not need to download it again unless you delete the image. You can verify your images with `docker images | grep robotics-rl`.

1.8 Summary

You now have a working laboratory. Specifically, you can trust four things:

- **The physics engine works.** MuJoCo initializes, Gymnasium-Robotics registers the Fetch environments, and observations have the expected structure: a dictionary with `observation` (10D), `achieved_goal` (3D), and `desired_goal` (3D).
- **Headless rendering works.** You can generate images of the robot without a display, using EGL or OSMesa as the rendering backend. This is the pipeline that will produce evaluation videos in later chapters.
- **The training loop works.** A complete cycle -- environment interaction, policy forward pass, gradient computation, parameter update, checkpoint save -- executes without error. The checkpoint is loadable by SB3.
- **The reward invariant holds.** `compute_reward(achieved_goal, desired_goal, info)` matches the reward returned by `env.step()`. This is the foundation for Hindsight Experience Replay (HER) in Chapter 5.

You also have a set of habits that will carry through the entire book: the three diagnostic questions (can it be solved? is it reliable? is it stable?), the experiment contract (artifacts on disk that you can inspect and share), and the willingness to inspect the environment directly rather than trusting that "it probably works."

What comes next is where things get interesting. The PPO smoke test ran for only 50,000 steps -- enough to verify the loop, not enough to learn. The remaining chapters will show you how to read training diagnostics (what does a healthy reward curve look like?), how to evaluate policies rigorously (across how many seeds? with deterministic or stochastic actions?), and how to tell whether a flat reward curve means "broken" or "needs more time."

Chapter 2 takes a deeper look at what the robot actually sees -- inspecting all observation components, understanding what the action space means physically, exploring reward semantics for both dense and sparse variants, and establishing the metrics schema (success rate, mean return, goal distance, action smoothness) that you will use for evaluation in every later chapter. Where this chapter asked "does the environment work?", Chapter 2 asks "do we understand what the environment is telling us?"

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
The artifacts in this chapter come from this run:

  bash docker/dev.sh python scripts/ch00_proof_of_life.py all

Hardware:     Any machine with Docker (GPU optional)
Time:         ~5 min (GPU) / ~10 min (CPU)

Artifacts produced:
  smoke_frame.png
```

```
  ppo_smoke.zip
  ppo_smoke.meta.json

Results summary:
  gpu-check:    PASS (CUDA available on NVIDIA; CPU fallback on Mac)
  list-envs:    PASS (Fetch environments listed)
  render:       PASS (smoke_frame.png created, shows Fetch robot)
  ppo-smoke:    PASS (ppo_smoke.zip created, loadable by PPO.load())
                PASS (ppo_smoke.meta.json created)

This chapter's pipeline is fast enough to run in full
every time. No checkpoint track is needed.
------------------------------------------------------------
```

Exercises

1. Change the seed and confirm reproducibility.

Run the proof-of-life pipeline with a different seed:

```
bash docker/dev.sh python scripts/ch00_proof_of_life.py all --seed 7
```

Confirm that the artifacts (`smoke_frame.png`, `ppo_smoke.zip`, and `ppo_smoke.meta.json`) are still produced. The rendered frame should look slightly different (the goal position is randomized), but the robot and table should be identical.

2. Force a rendering backend.

By default, the script tries EGL first, then falls back to OSMesa. Force each backend explicitly and document what happens on your system. We set the variable inside the container (after `bash -c`) to ensure it takes effect, since `docker/dev.sh` sets its own `MUJOCO_GL` value:

```
# Force EGL
bash docker/dev.sh bash -c 'MUJOCO_GL=egl python scripts/ch00_proof_of_life.py ren

# Force OSMesa
bash docker/dev.sh bash -c 'MUJOCO_GL=osmesa python scripts/ch00_proof_of_life.py
```

Which backend works on your machine? If only one works, note which one and why (hint: EGL requires NVIDIA drivers; OSMesa is software-only). The rendered image should be identical regardless of backend.

3. Inspect the observation dictionary across multiple resets.

Write a short script that creates `FetchReachDense-v4`, resets it 100 times, and for each reset prints or accumulates:

- The shape and dtype of each observation component
- The min and max values across all resets for each array
- Whether any values are NaN or infinite

This gives you a feel for the value ranges that the policy network will need to handle. Expect all values to be finite and float64. The `observation` array contains positions in meters and velocities in meters/second -- typical ranges are roughly [-0.1, 0.1] for velocities and [1.0, 1.5] for positions. The `desired_goal` should cover the workspace bounds (roughly x in [1.05, 1.55], y in [0.40, 1.10], z in [0.42, 0.60]) with uniform-looking coverage. If the goals are clustered or identical across resets, the environment may not be randomizing correctly.

This is also a good opportunity to check that different seeds produce different goal positions (they should -- the goal is sampled uniformly within the workspace at each reset).

4. (Challenge) Modify the success threshold.

The default success threshold for FetchReach is 0.05 meters (5 cm). What happens if you change it? The threshold is defined in the environment's XML configuration. Explore:

- Can you find where the threshold is set? (Hint: look at the `distance_threshold` parameter in the Gymnasium-Robotics source code, or check `env.unwrapped.distance_thres` after creating the environment.)
- If you made the threshold larger (say, 0.10 m), would a random policy succeed more often? Estimate by running 100 random episodes and checking `is_success` at the final step.
- If you made it smaller (say, 0.01 m), what would happen to training difficulty? Consider: how precisely would the end-effector need to be positioned?

You do not need to actually modify the threshold to answer these questions -- running random episodes and measuring goal distances will give you the intuition. We will revisit this question when we discuss sparse rewards in Chapter 5, where the threshold directly determines how much of the goal space produces zero reward.

\newpage

2 Environment Anatomy: What the Robot Sees, Does, and Learns From

This chapter covers:

- Inspecting the dictionary observation structure that every Fetch environment returns -- what each number means physically and why observations are dictionaries, not flat vectors
- Understanding what the 4D action vector controls: Cartesian deltas for the end-effector and a gripper open/close command
- Verifying reward computation for both dense (distance-based) and sparse (binary success/failure) variants, and proving the critical invariant that makes Hindsight Experience Replay possible
- Simulating goal relabeling by hand -- calling `compute_reward` with goals the environment never intended -- to see why the Fetch interface enables HER

- Establishing a random-policy baseline (success rate, mean return, goal distance) that every trained agent in later chapters must beat

In Chapter 1, you verified that the computational stack works: Docker launches, MuJoCo simulates, the rendering pipeline produces frames, and a training loop runs to completion. You have a proof of life -- evidence that the environment is alive and capable of producing results.

But alive is not the same as understood. You know the environment *works*, but not what it *says*. What do the 10 numbers in the observation vector mean? What happens when the robot takes action `[1, 0, 0, 0]`? Why does the reward function return -0.073 instead of -1? How does the environment know the robot "succeeded"? Answering these questions gives you the foundation to debug training failures and choose appropriate algorithms.

This chapter provides a complete anatomy of Fetch environment observations, actions, rewards, and goals. You will inspect every component by hand, verify reward computation against the distance formula, simulate HER-style goal relabeling, and establish the random-policy baseline that every trained agent must beat. With the environment understood, Chapter 3 trains a real policy -- PPO on dense Reach -- where the observation shapes you document here determine the network architecture, and the random baseline you establish here is the floor that PPO must exceed.

2.1 WHY: Why environment anatomy matters

Imagine you train a policy for 10 hours. The training loop completes without errors. You evaluate the checkpoint and find a success rate of 0%. You check the reward curve -- flat. You check the loss -- it looks normal. Nothing crashed. What went wrong?

Understanding what the agent sees and what rewards mean is what makes this question answerable. The problem could be anywhere: the observations might have unexpected scales, the rewards might have the wrong sign, the success threshold might be different from what you assumed, or the goal structure might not match what the algorithm expects. Knowing the environment anatomy turns a 10-hour mystery into a 10-minute diagnostic.

In our experience, environment misunderstandings are the most common source of wasted compute in robotics RL -- more than bad hyperparameters or wrong algorithms. The root cause is usually wrong assumptions about what the environment is actually doing.

Three questions that anatomy answers

Here are three questions that come up in every training failure. Each one becomes straightforward once you have inspected the environment:

1. "Is the observation what the network expects?" When you create a policy with SB3's `MultiInputPolicy`, it reads the observation space to determine its input structure. If the observation is a dictionary with three keys, the network builds three separate encoders and concatenates them. If the observation were somehow a flat

vector (due to a wrapper or version mismatch), the network architecture would be completely different -- and wrong for the task. You need to know exactly what the observation contains to understand what the network receives.

2. "Is the reward on a reasonable scale?" Dense Fetch rewards are negative distances -- typical values in the range -0.01 to -0.3 for FetchReach. If you tuned your learning rate assuming rewards on the order of -1 to 0 (as with sparse rewards), your value function targets would be off by an order of magnitude. Reward scale affects everything: the value function's target range, the entropy coefficient in SAC, and the advantage normalization in PPO.

3. "Can this environment support HER?" Hindsight Experience Replay -- which we introduce in Chapter 5 -- requires two specific properties from the environment: an explicit `achieved_goal` in the observation, and a `compute_reward` function that accepts arbitrary goals. If either is missing, HER cannot work. Checking these properties upfront saves you from a frustrating debugging session where the problem is structural, not algorithmic -- a distinction that is much easier to spot before training than after.

What misunderstandings cost

To be concrete about the cost, here is a table of misunderstandings we have seen (some in our own work, some reported by others) and what they led to:

Misunderstanding	What happens
Assumed observations are flat vectors	Network architecture wrong
Did not know `achieved_goal` tracks object (not gripper) in FetchPush	Reward calculations wrong;
Assumed sparse reward is -1/+1 (it is -1/0)	Value function targets off; c
Did not verify `compute_reward` matches `env.step()`	HER learns from corrupted
Did not check success threshold	Evaluated with wrong succe

The common thread: every one of these is preventable by inspecting the environment before training. This chapter does that inspection systematically.

The anatomy as a debugging foundation

When you understand what the environment returns, debugging becomes tractable. A flat reward curve is no longer a mystery -- you can check: does the policy's output fall within the action space bounds? Is the achieved goal moving in response to actions? Is the reward decreasing (getting closer to zero) even if success rate has not budged? Is `compute_reward` returning the same values as `env.step()`?

Each of these diagnostic checks has a specific prerequisite in this chapter's anatomy. Checking action bounds requires knowing the action space shape and range (section 2.4), while checking goal movement requires knowing what `achieved_goal` tracks -- gripper position for Reach, object position for Push (section 2.3). Checking reward trends requires knowing the reward formula and its range, since dense rewards are negative distances and sparse rewards are 0 or -1 (section 2.5). And checking the

`compute_reward` invariant requires knowing the API signature and how it relates to `env.step()` (also section 2.5).

All of these will come up in practice. In Chapter 3, when PPO training stalls, the first thing to check is whether the reward is moving. In Chapter 5, when HER does not improve performance, the first thing to check is whether `compute_reward` handles relabeled goals correctly. The anatomy you build here is the diagnostic toolkit for every later chapter.

The rest of this chapter gives you that knowledge, piece by piece. We start with what the agent sees (observations and goals), move to what it can do (actions), then to how performance is measured (rewards), and finally to the key insight that ties it all together (goal relabeling).

The goal-conditioned MDP: seven pieces

The Fetch environments implement a specific mathematical structure called a *goal-conditioned Markov Decision Process* (GCMDP). Here is what we mean by that -- not as a formal theorem, but as a concrete description of the interface you will work with.

A goal-conditioned MDP has seven pieces:

- **States** ($\mathcal{S}$): the full physical state of the robot and any objects on the table
- **Actions** ($\mathcal{A}$): what the robot can do (4D Cartesian deltas plus gripper)
- **Goals** ($\mathcal{G}$): the target positions the robot must achieve (3D Cartesian coordinates)
- **Transitions** (P): the physics -- how the state changes when the robot acts
- **Rewards** (R): the feedback signal, which depends on the goal
- **Goal-achievement mapping** (ϕ): a function that extracts "what goal did we achieve?" from the current state
- **Discount factor** (γ): how much we value future rewards versus immediate ones

The critical insight is that the reward depends on the goal. The same state might yield reward -0.1 for one goal (you are close) and -0.5 for another (you are far away). And the goal-achievement mapping ϕ tells us what goal we *actually* achieved, regardless of what goal we were aiming for. These two properties -- goal-dependent rewards and an explicit achieved goal -- are what make Hindsight Experience Replay possible. We will see exactly why in section 2.6.

2.2 The Fetch task family

Chapter 1 introduced the four Fetch tasks: Reach, Push, PickAndPlace, and Slide. Here we add the physical details that matter for environment anatomy.

The simulated robot. The Fetch arm is a 7-degree-of-freedom manipulator modeled in MuJoCo, a rigid-body physics simulator. MuJoCo runs the physics at 500 Hz internally; the environment exposes control at 25 Hz (every 20 simulation steps). When you call `env.step(action)`, MuJoCo simulates 20 timesteps of physics, then returns the resulting state.

The workspace. The arm sits on a fixed base next to a table. The workspace -- the region where goals are sampled and the end-effector can reach -- spans roughly:

```
Fetch Workspace (approximate bounds in meters):

  z (height)
  0.6 |  +--------------------------+
      |  |   goal region (air)      |
  0.5 |  |                          |
      |  +--------------------------+ <-- table surface ~0.42
  0.4 |
      +----+----+----+----+----+----> x (forward)
         1.0  1.1  1.2  1.3  1.4  1.5

  y (sideways) spans roughly 0.5 to 1.0 (goal region)
```

When you render this scene, the coordinates come to life. The tan tabletop sits at z=0.42 -- about kitchen-counter height in the simulation. The silver arm links rise from a fixed base at the table's edge, and the gripper hovers at roughly z=0.53, just above where you would rest your elbow. A small red sphere marks the target goal, floating somewhere in the air above the table. The workspace numbers above map directly onto this visual: x runs forward away from the base, y spans the table side to side, and z measures height above the floor.

These numbers matter because they give you a sense of scale. The entire workspace is about 60 cm x 70 cm x 20 cm. The success threshold is 5 cm (0.05 m), which means the end-effector must be within about 8% of the workspace width to "succeed." Random flailing is unlikely to land within 5 cm of an arbitrary target -- which is why random baselines have near-zero success rates.

Dense vs. sparse rewards. Each task comes in two reward variants, and the environment ID encodes which: `FetchReachDense-v4` uses dense rewards (negative distance to goal), while `FetchReach-v4` uses sparse rewards (0 if goal reached, -1 otherwise). The observation structure and action space are identical between variants -- only the reward computation differs.

For this chapter, we work primarily with `FetchReachDense-v4` (dense Reach). It is the simplest variant: no objects, continuous feedback, and a workspace that the arm can easily cover. The anatomy principles transfer directly to Push, PickAndPlace, and Slide -- the interface is the same, and we verify that uniformity in section 2.7. Figures 2.1 through 2.3 show the three Fetch environments we use in this book.

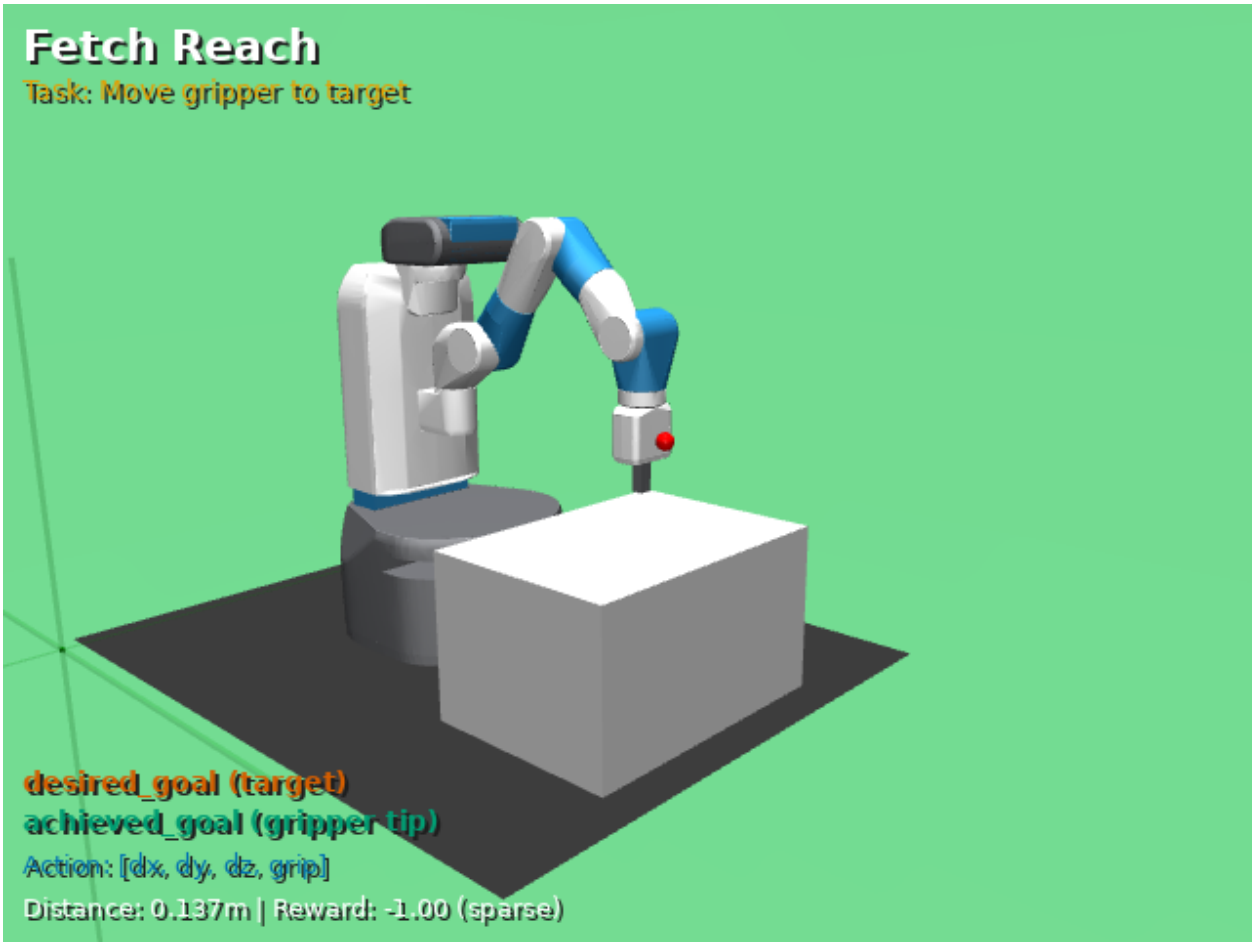

Figure 2.1: FetchReach-v4 after `env.reset()`. The silver arm extends over a tan tabletop, its parallel-jaw gripper (achieved_goal) hovering in the air. A small red sphere marks the target (desired_goal). No objects are involved -- this is pure arm positioning. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchReach-v4`.)

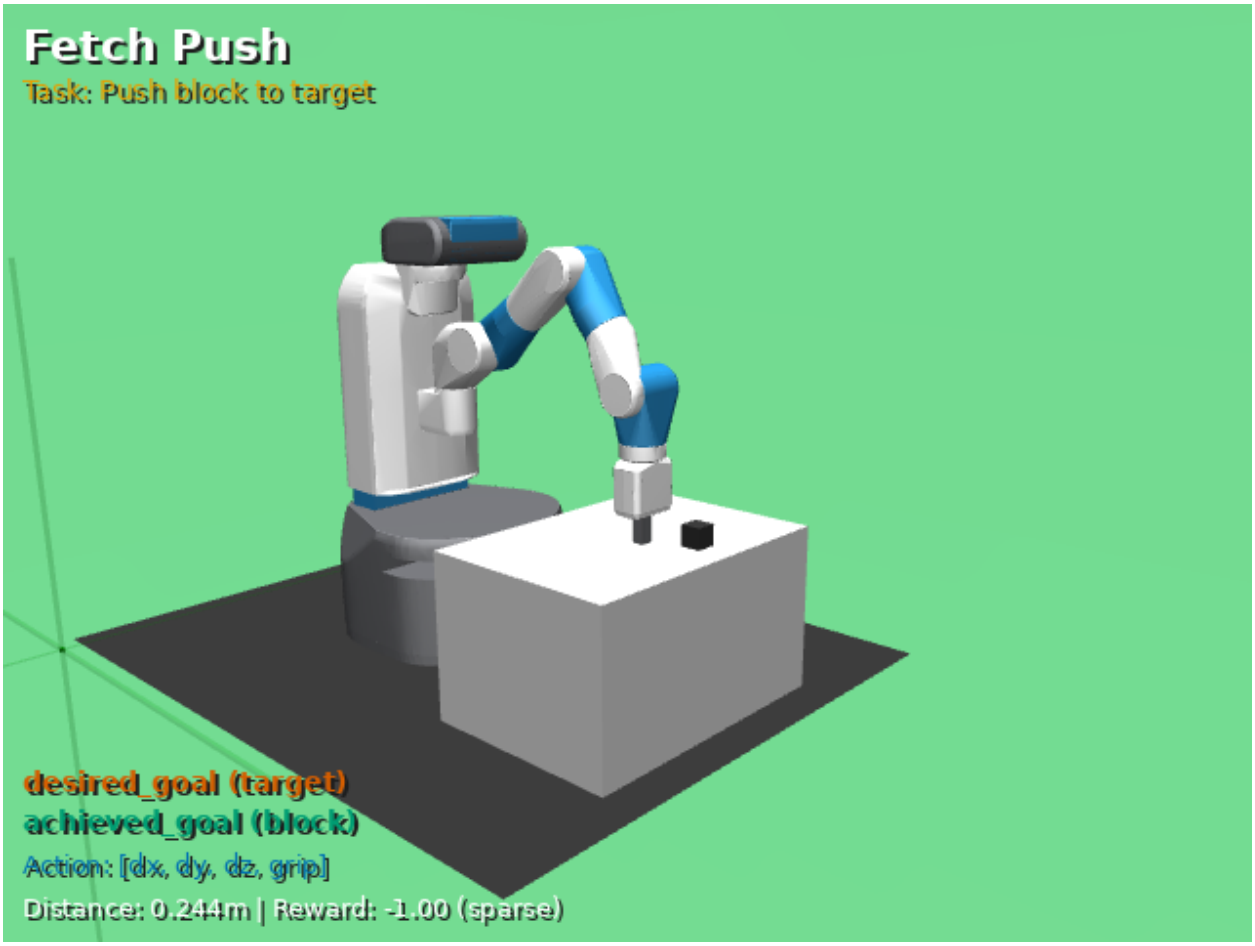

Figure 2.2: FetchPush-v4 after `env.reset()`. A dark cube rests on the tan tabletop; a red marker on the surface shows where it needs to go. The robot must slide the block across the table to that target position. Here, `achieved_goal` is the block's position (not the gripper's), and `desired_goal` is where the block should end up. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchPush-v4`.)

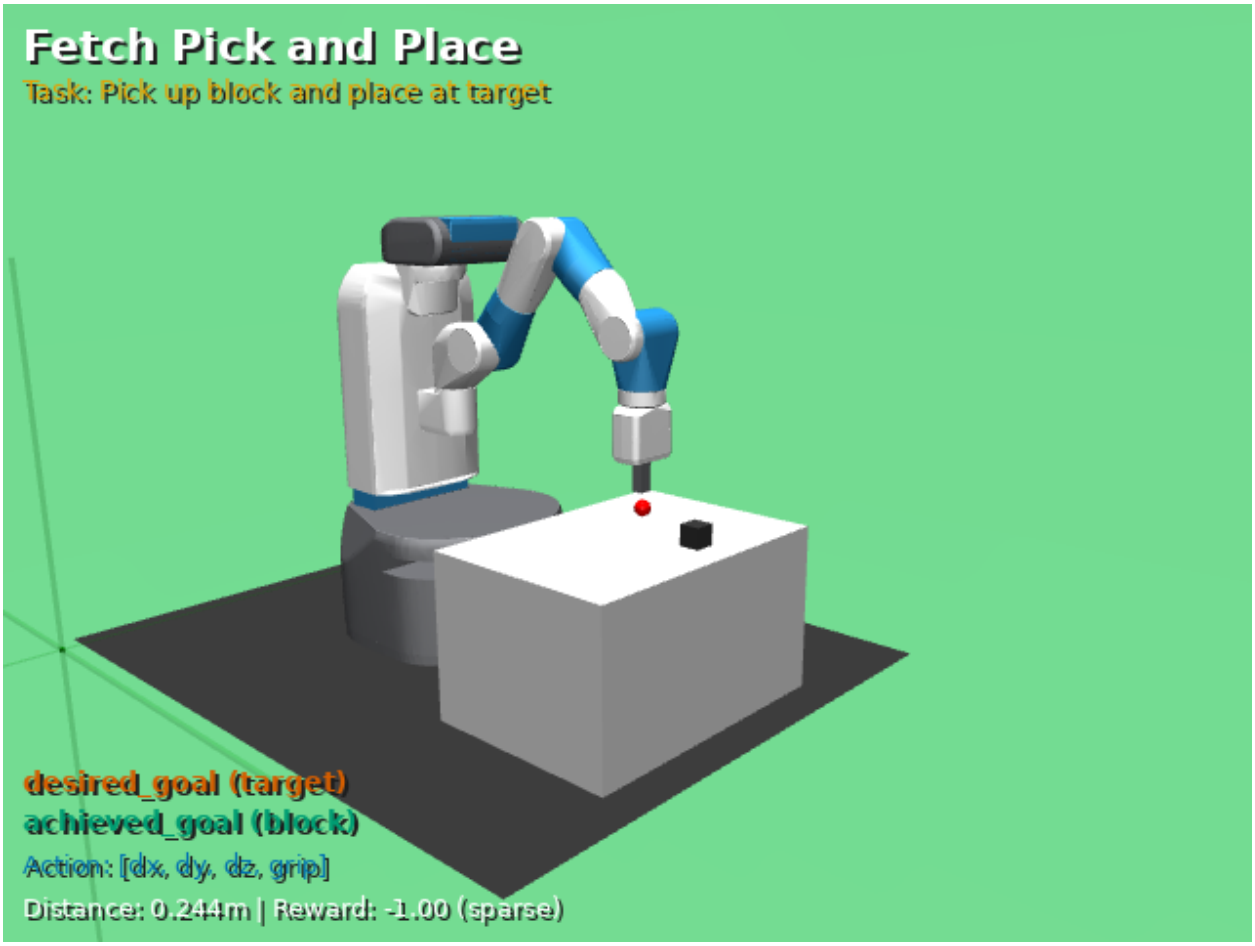

Figure 2.3: FetchPickAndPlace-v4 after `env.reset()`. The red target sphere floats in the air above the table -- visibly higher than in Push, signaling that the robot must lift the block off the surface. The robot must pick up the block and place it at that suspended target position, coordinating grasping, lifting, and releasing. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchPickAndPlace-v4`.)

Episode structure. Every episode lasts exactly 50 steps (the environment truncates at this limit). At each step, the agent observes the state, chooses an action, and receives a reward. The episode ends when the step limit is reached -- there is no early termination on success. This means that even a successful policy continues to act for the full 50 steps, which affects how you interpret returns: a policy that reaches the goal on step 10 still receives rewards for the remaining 40 steps. For dense rewards, those remaining steps contribute small negative values (the agent is near the goal). For sparse rewards, they contribute 0s (the agent has already succeeded).

2.3 Build It: The observation dictionary

The first thing to understand about any RL environment is what the agent perceives. In Fetch environments, the agent does not see a flat vector but rather a dictionary with three keys, each carrying a different kind of information. This structure is not just a

data format -- it is the interface through which goal-conditioned learning operates.

What the dictionary contains

The observation dictionary has three entries:

- `obs["observation"]` -- the robot's proprioceptive state. For FetchReach, this is a 10-dimensional vector containing the gripper's position, velocity, and finger state.
- `obs["achieved_goal"]` -- where the relevant thing (end-effector for Reach, object for Push) currently is. Always 3D Cartesian coordinates.
- `obs["desired_goal"]` -- where we want that thing to be. Also 3D Cartesian coordinates.

The separation of achieved and desired goals from the main observation is the key design decision. It makes the goal-conditioning explicit: the environment tells the agent "here is your state, here is where you are, here is where you need to be."

Why not a flat vector? Many RL environments (CartPole, Atari, MuJoCo locomotion) return a single array as the observation. Fetch environments *could* concatenate everything into a flat vector of length 16 (for Reach) or 31 (for Push). But the dictionary structure serves two purposes. First, it makes the goal-conditioned interface explicit -- the achieved and desired goals are labeled, not buried at arbitrary indices in a flat vector. Second, it enables SB3's `MultiInputPolicy` to process each component through a separate encoder before concatenation, which gives the network a natural way to compare "where I am" against "where I should be."

Inspecting the observation

> **Note:** All code listings in this chapter assume `import gymnasium as gym`, `import gymnasium_robotics`, and `import numpy as np`. The full runnable versions live in `scripts/labs/env_anatomy.py`.

Let's look at the code that inspects this structure:

```python
# Observation dictionary inspector
# (adapted from scripts/labs/env_anatomy.py:obs_inspector)

def obs_inspector(env_id="FetchReachDense-v4", seed=0):
    """Inspect obs dict: keys, shapes, dtypes, workspace bounds."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    summary = {}
    for key in ["observation", "achieved_goal", "desired_goal"]:
        arr = np.asarray(obs[key])
        summary[key] = {
            "shape": arr.shape, "dtype": str(arr.dtype),
            "min": float(arr.min()), "max": float(arr.max()),
            "finite": bool(np.all(np.isfinite(arr))),
        }
    # Check desired_goal is within workspace
```

```
    dg = np.asarray(obs["desired_goal"])
    summary["desired_goal_in_workspace"] = bool(
        1.1 <= dg[0] <= 1.5 and 0.5 <= dg[1] <= 1.0
        and 0.35 <= dg[2] <= 0.75
    )
    env.close()
    return summary
```

Checkpoint. Run `bash docker/dev.sh python scripts/labs/env_anatomy.py --verify` and check the first output section. You should see:

- `obs=(10,)`, `ag=(3,)`, `dg=(3,)`
- All values finite
- `desired_goal in workspace: True`

If the observation shape is not `(10,)`, check that you are using `FetchReachDense-v4` (not a Push or PickAndPlace variant, which have 25D observations).

The 10D observation breakdown

What do those 10 numbers actually represent? Here is the breakdown for FetchReach:

Index	Semantic	Units	Typical range
0-2	Gripper position (x, y, z)	meters	[1.1, 1.5], [0.5, 1.0], [0.35, 0.75]
3-4	Gripper finger positions (right, left)	meters	[0.0, 0.05]
5-7	Gripper linear velocity (dx, dy, dz)	m/s	[-0.1, 0.1]
8-9	Gripper finger velocities (right, left)	m/s	[-0.01, 0.01]

For FetchReach, the first three elements of `obs["observation"]` -- the gripper position -- are the same as `obs["achieved_goal"]`. This is because the goal-achievement mapping ϕ for reaching is: "where is the end-effector?" We can verify this:

The goal space: where goals live

The goal-achievement mapping ϕ extracts the "achieved goal" from the current state -- for Reach, $\phi(s)$ is the gripper position, while for Push, $\phi(s)$ is the object position. This mapping is what determines `achieved_goal` in the observation dictionary.

```
# Goal space verification
# (adapted from scripts/labs/env_anatomy.py:goal_space)

def goal_space(env_id="FetchReachDense-v4", n_resets=100, seed=0):
    """Sample goals via reset; check bounds; verify phi(s)=obs[:3]."""
    env = gym.make(env_id)
    desired_goals, phi_matches = [], []
    for i in range(n_resets):
        obs, _ = env.reset(seed=seed + i)
```

```
        desired_goals.append(np.asarray(obs["desired_goal"]))
        ag = np.asarray(obs["achieved_goal"])
        obs_vec = np.asarray(obs["observation"])
        # phi(s) = grip_pos = obs["observation"][:3] for FetchReach
        phi_matches.append(bool(np.allclose(ag, obs_vec[:3])))
    dg_arr = np.stack(desired_goals)
    env.close()
    return {
        "n_resets": n_resets,
        "goal_dim": int(dg_arr.shape[1]),
        "desired_goal_bounds": {
            "min": dg_arr.min(axis=0).tolist(),
            "max": dg_arr.max(axis=0).tolist(),
        },
        "all_phi_match": all(phi_matches),
    }
```

Checkpoint. The verification output should show:

- `goal_dim=3`
- `phi matches obs[:3]: True` (all 100 resets)
- `desired_goal range:` within the workspace (roughly x in [1.1, 1.5], y in [0.5, 1.0], z in [0.35, 0.75])

If `phi matches obs[:3]` is False for any reset, there is a version mismatch in gymnasium-robotics. If the goal range is very narrow (all coordinates nearly identical), the environment is not randomizing goals correctly -- check your seed handling.

What the policy network will see

When we create an SB3 model with `MultiInputPolicy` in Chapter 3, the network reads this dictionary observation space and builds the architecture shown in Figure 2.4:

Network input	Source key	Dimensions
State encoder	`observation`	10
Achieved goal encoder	`achieved_goal`	3
Desired goal encoder	`desired_goal`	3
Total input	(concatenated)	**16**

The encoders process each key through separate MLP layers, then concatenate the results before passing through shared layers. The total input dimension -- 16 for FetchReach, 31 for FetchPush -- determines the network's capacity requirements. This is why we document shapes carefully: they directly affect architecture.

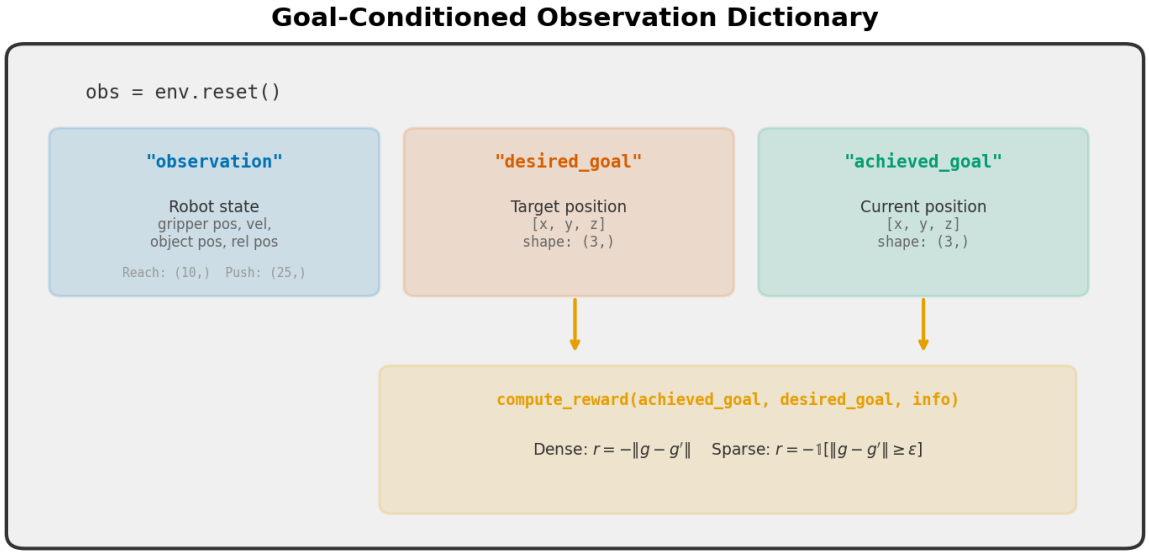

Figure 2.4: The observation dictionary structure and how it maps to the policy network. Each key in the dictionary (`observation`, `achieved_goal`, `desired_goal`) feeds a separate encoder in SB3's `MultiInputPolicy`. The outputs are concatenated into a single feature vector (16D for FetchReach, 31D for FetchPush) before passing through shared layers. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py reward-diagram`.)

2.4 Build It: Action semantics

The agent does not control joint torques. Instead, it outputs a 4-dimensional vector of Cartesian commands that an internal controller translates into joint movements. This is an important design choice: the learning algorithm does not need to solve inverse kinematics, which makes the learning problem more tractable.

What each action dimension controls

Index	Semantic	Range	Physical effect
0	dx	[-1, 1]	End-effector moves in the x direction (forward/backward)
1	dy	[-1, 1]	End-effector moves in the y direction (left/right)
2	dz	[-1, 1]	End-effector moves in the z direction (up/down)
3	gripper	[-1, 1]	< 0 opens, > 0 closes the parallel-jaw gripper

Actions are clipped to [-1, 1] and then scaled by the environment's internal controller before being applied as forces. The scaling means that action `[1, 0, 0, 0]` does not move the end-effector by exactly 1 meter -- it applies a maximal positive displacement command along x, and the actual movement depends on the physics simulation (friction, inertia, joint limits).

Verifying action-to-movement mapping

We can verify that each action axis produces movement in the expected direction:

```
# Action space explorer
# (adapted from scripts/labs/env_anatomy.py:action_explorer)

def action_explorer(env_id="FetchReachDense-v4", seed=0):
    """Step with axis-aligned actions, measure displacement."""
    env = gym.make(env_id)
    results = {
        "action_shape": env.action_space.shape,
        "action_low": env.action_space.low.tolist(),
        "action_high": env.action_space.high.tolist(),
        "displacements": {},
    }
    axis_names = ["x", "y", "z"]
    for axis_idx in range(3):
        obs, _ = env.reset(seed=seed)
        grip_before = np.asarray(obs["achieved_goal"]).copy()
        action = np.zeros(4, dtype=np.float32)
        action[axis_idx] = 1.0
        obs, _, _, _, _ = env.step(action)
        grip_after = np.asarray(obs["achieved_goal"])
        disp = grip_after - grip_before
        results["displacements"][axis_names[axis_idx]] = {
            "action": action.tolist(),
            "displacement": disp.tolist(),
            "moved_positive": bool(disp[axis_idx] > 0),
        }
    env.close()
    return results
```

Checkpoint. The verification output should show:

- `action_shape=(4,)`, bounds `[-1, 1]`
- Action `[1,0,0,0]` produces positive x displacement
- Action `[0,1,0,0]` produces positive y displacement
- Action `[0,0,1,0]` produces positive z displacement

The displacement magnitudes are small (typically 0.005-0.02 meters per step) because the action is applied for one 25 Hz control step. At 50 steps per episode, the end-effector can traverse roughly 0.25-1.0 meters total -- enough to cover the workspace.

For FetchReach, the gripper dimension (index 3) has no effect on the task -- there is no object to grasp. For Push and PickAndPlace, it becomes essential: the agent must learn to close the gripper around an object and open it at the target location.

Warning: A subtle point about action semantics: the policy outputs continuous values in [-1, 1], but the mapping from these values to physical displace-

ment is not linear and depends on MuJoCo's internal controller. An action of [0.5, 0, 0, 0] does not produce exactly half the displacement of [1.0, 0, 0, 0]. In practice, this does not matter for training -- the policy learns the mapping implicitly -- but it means you should not try to hand-code a controller using these action values as if they were calibrated velocity commands.

2.5 Build It: Reward computation -- dense and sparse

Now that we know what the agent sees and what it can do, we need to understand how the environment evaluates performance. Fetch environments provide two reward variants, and understanding both is essential for the rest of the book.

Dense reward: negative distance

The dense reward for any Fetch environment is:

$$R_{\text{dense}} = -\|g_a - g_d\|_2$$

where g_a is the achieved goal (a 3D position), g_d is the desired goal (also 3D), and $\|\cdot\|_2$ is the Euclidean norm. The reward is always negative or zero, with zero meaning perfect goal achievement. A reward of -0.1 means the relevant point (end-effector for Reach, object for Push) is 10 cm from the target.

This reward provides continuous gradient information: every action that moves closer to the goal produces a less negative reward. Standard policy gradient methods like PPO can learn effectively from this signal because there is always a direction to improve.

Verifying the dense reward formula

We verify that three things match: the manual distance formula, the `compute_reward` API, and the reward returned by `env.step()`.

```
# Dense reward check
# (adapted from scripts/labs/env_anatomy.py:dense_reward_check)

def dense_reward_check(env_id="FetchReachDense-v4", n_steps=100,
                       seed=0, atol=1e-10):
    """Verify R = -||ag - dg|| matches step_reward and compute_reward."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    mismatches = 0
    for t in range(n_steps):
        action = env.action_space.sample()
        obs, step_reward, terminated, truncated, info = env.step(action)
        ag = np.asarray(obs["achieved_goal"])
        dg = np.asarray(obs["desired_goal"])
        manual = -float(np.linalg.norm(ag - dg))
        cr = float(
```

```
            env.unwrapped.compute_reward(ag, dg, info))
        step_r = float(step_reward)
        if abs(manual - cr) > atol or abs(manual - step_r) > atol:
            mismatches += 1
        if terminated or truncated:
            obs, info = env.reset(seed=seed + t + 1)
    env.close()
    return {"n_steps": n_steps, "mismatches": mismatches}
```

This three-way comparison is the heart of the verification. We compute the reward ourselves (`manual`), ask the environment to compute it (`cr`), and compare both to what `env.step()` returned (`step_r`). All three must agree.

Why do we check all three? Because each catches a different failure mode. If `manual` differs from `cr`, our understanding of the reward formula is wrong -- maybe the reward is not just negative distance, but includes a scaling factor or an offset. If `cr` differs from `step_r`, there is a bug in the environment or a version mismatch between the wrapper and the base environment. If `manual` matches `cr` but not `step_r`, the wrapper is modifying the reward (perhaps adding a penalty term). Any of these mismatches would corrupt HER's relabeling in Chapter 5.

> **Checkpoint.** Over 100 steps: zero mismatches, tolerance 1e-10. The three values should match to floating-point precision. If they do not match, you likely have a version mismatch between gymnasium and gymnasium-robotics.

Sparse reward: binary success signal

As Figure 2.5 illustrates, the sparse reward works very differently from the dense variant: instead of a smooth gradient, the agent receives a binary signal determined by a distance threshold $\epsilon = 0.05$ meters (5 cm):

$$R_{\text{sparse}} = \begin{cases} 0 & \text{if } \|g_a - g_d\|_2 \leq \epsilon \\ -1 & \text{otherwise} \end{cases}$$

Note the values: 0 for success and -1 for failure -- not +1 for success, not -1/+1. The reward is always non-positive, which matters for value function initialization and for understanding what a "good" return looks like.

Let's trace through the numbers. With sparse rewards, an episode of 50 steps where the goal is never reached has a return of $\sum_{t=0}^{49}(-1) = -50$. An episode where the goal is reached on step 40 and maintained for the remaining 10 steps has a return of $40 \times (-1) + 10 \times 0 = -40$. A perfect policy that reaches the goal immediately would have a return of $-1 \times (\text{steps to reach goal}) + 0 \times (\text{remaining steps})$. The best possible return on a 50-step episode is 0 (goal reached on step 0), but in practice even good policies take a few steps to reach the goal, so returns of -2 to -5 indicate strong performance.

For dense rewards, the numbers are different but the reasoning is similar. The return is the sum of negative distances across all 50 steps, so a random policy averages about -25 to -10 per episode, while a well-trained policy that quickly reaches the goal and stays there might achieve -1 to -3.

Verifying the sparse reward formula

```
# Sparse reward check
# (adapted from scripts/labs/env_anatomy.py:sparse_reward_check)

def sparse_reward_check(env_id="FetchReach-v4", n_steps=100,
                        seed=0, atol=1e-10):
    """Verify R = 0 if ||ag-dg|| <= eps else -1; check threshold."""
    env = gym.make(env_id)
    eps = float(env.unwrapped.distance_threshold)
    obs, info = env.reset(seed=seed)
    mismatches, non_binary = 0, 0
    for t in range(n_steps):
        action = env.action_space.sample()
        obs, step_reward, terminated, truncated, info = env.step(action)
        ag = np.asarray(obs["achieved_goal"])
        dg = np.asarray(obs["desired_goal"])
        dist = float(np.linalg.norm(ag - dg))
        expected = 0.0 if dist <= eps else -1.0
        step_r = float(step_reward)
        cr_r = float(
            env.unwrapped.compute_reward(ag, dg, info))
        if step_r not in (0.0, -1.0):
            non_binary += 1
        if (abs(step_r - expected) > atol
                or abs(cr_r - expected) > atol):
            mismatches += 1
        if terminated or truncated:
            obs, info = env.reset(seed=seed + t + 1)
    env.close()
    return {"n_steps": n_steps, "threshold": eps,
            "mismatches": mismatches, "non_binary": non_binary}
```

Checkpoint. Over 100 steps:

- `threshold=0.05`
- `mismatches=0`
- `non_binary=0` (every sparse reward is exactly 0.0 or -1.0)

If the threshold is not 0.05, your gymnasium-robotics version may use a different default. If non-binary count is positive, something unexpected is happening with the reward computation.

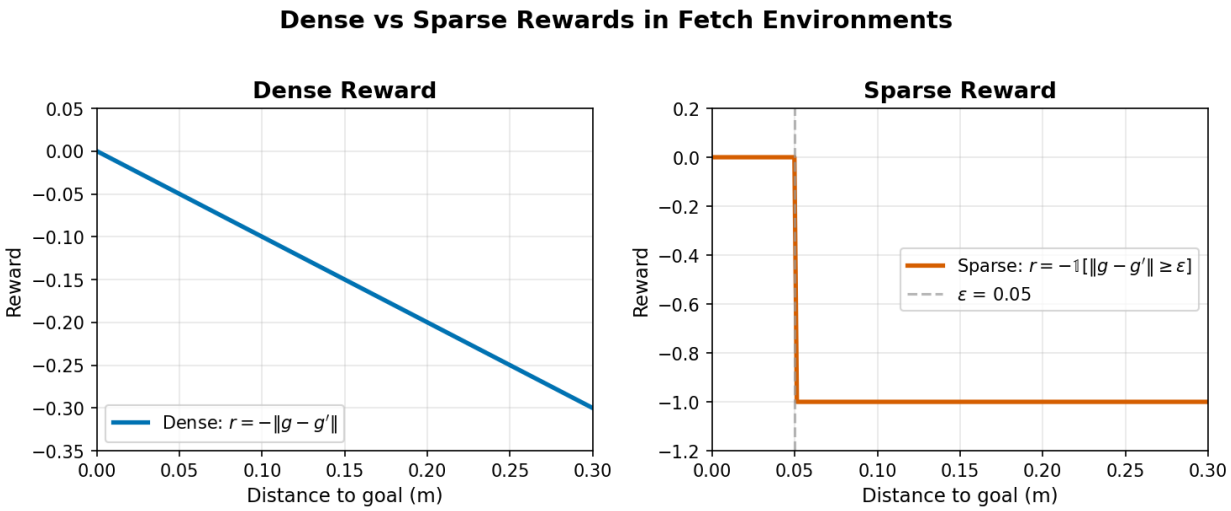

Figure 2.5: Dense vs sparse reward functions. Left: dense reward $R = -\|g_a - g_d\|$ provides a smooth gradient signal -- every action that reduces distance improves the reward. Right: sparse reward is a step function at the threshold $\epsilon = 0.05$ m -- the agent receives no useful gradient until it crosses the threshold. This is why sparse rewards create a needle-in-a-haystack exploration problem that standard algorithms like PPO cannot solve alone. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py reward-diagram`.)

The critical invariant

Both reward checks verify the same fundamental property -- the *critical invariant* for the entire book:

> The reward returned by `env.step()` must equal `env.unwrapped.compute_reward(achieved_g desired_goal, info)`.

We run these checks because different versions of gymnasium-robotics have had bugs affecting reward computation, and API changes have altered the signature of `compute_reward`. A quick verification ensures that your specific installation behaves correctly.

More importantly, this invariant is the foundation of Hindsight Experience Replay. When HER relabels a trajectory with a different goal, it calls `compute_reward` with the new goal to get the relabeled reward. If `compute_reward` disagrees with `env.step()`, HER trains on incorrect labels. The policy would learn from corrupted data, and training could fail silently -- no error, no crash, just a policy that does not work.

> **Tip:** If you are ever unsure whether the reward invariant holds after upgrading gymnasium-robotics, run the quick check: `bash docker/dev.sh python scripts/labs/env_anatomy.py --verify`. The dense and sparse reward checks (components 4 and 5) specifically verify this invariant. It takes less than a minute and can save you days of debugging corrupted HER training.

2.6 Build It: Goal relabeling simulation

Everything we have built up to -- the dictionary observations, the separate achieved and desired goals, the `compute_reward` function -- converges here. We are going to do something that seems odd at first: call `compute_reward` with goals that the environment never set.

Why relabeling matters

The problem with sparse rewards is simple: if the agent never reaches the goal, the reward is -1 at every step of every episode, which means there is no gradient signal pointing toward success. The agent has no way to distinguish between a near-miss (3 cm from the goal) and a complete failure (30 cm away), since both get reward -1.

Hindsight Experience Replay (HER), which we cover in depth in Chapter 5, solves this by asking: "You failed to reach the goal, but you *did* reach some position. What if that position had been the goal?" HER takes the `achieved_goal` from a failed trajectory and retroactively pretends it was the `desired_goal`. Then it recomputes the reward -- and because the agent actually reached that position, the recomputed reward is 0 (success).

For this to work, the environment must let us call `compute_reward` with arbitrary goals -- not just the goal that was originally set. Let's verify that this works:

Simulating relabeling by hand

```
# Goal relabeling check
# (adapted from scripts/labs/env_anatomy.py:relabel_check)

def relabel_check(env_id="FetchReachDense-v4", n_goals=10,
                  seed=0, atol=1e-10):
    """Call compute_reward with arbitrary goals (HER simulation)."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    goal_sp = env.observation_space.spaces["desired_goal"]
    # Take one step to get a non-trivial achieved_goal
    obs, _, _, _, info = env.step(env.action_space.sample())
    ag = np.asarray(obs["achieved_goal"])
    reward_type = getattr(
        env.unwrapped, "reward_type", "dense")
    eps = float(env.unwrapped.distance_threshold)
    mismatches = 0
    for _ in range(n_goals):
        random_goal = goal_sp.sample()
        cr = float(
            env.unwrapped.compute_reward(ag, random_goal, info))
        dist = float(np.linalg.norm(ag - random_goal))
        expected = (-dist if reward_type == "dense"
```

```
                    else (0.0 if dist <= eps else -1.0))
        if abs(cr - expected) > atol:
            mismatches += 1
    env.close()
    return {"n_goals": n_goals, "mismatches": mismatches}
```

Here is what this code does. It first takes a step to get a non-trivial achieved goal (the gripper moved somewhere), then samples 10 random goals from the goal space -- positions the environment never intended as targets. For each random goal, it calls `compute_reward` with the actual achieved goal and verifies that the returned reward matches the distance formula.

This is exactly what HER does, in miniature. The agent "failed" (it did not reach the original desired goal), but we can ask: "what would the reward have been if the goal were somewhere else?" And `compute_reward` gives us a correct answer.

> **Checkpoint.** Over 10 random goals: zero mismatches. `compute_reward` correctly handles arbitrary goals -- not just the one the environment set. No errors, no NaN, no crashes.
>
> This proves the key property for HER: **we can recompute rewards for any goal without re-running the simulation.**

Why this works (and why it would not work everywhere)

The reason `compute_reward` accepts arbitrary goals is that the Fetch reward depends on the goal only through the distance $\|g_a - g\|$. The function does not need to access the physics state, the action history, or any internal simulation data -- it takes two 3D vectors and computes a distance, which is why relabeling is both cheap and exact.

Not all environments have this property. An environment where the reward depends on the trajectory (not just the endpoint), or where "success" requires a specific sequence of actions, would not support this kind of relabeling. An environment that returns flat observations with no goal separation does not expose the structure HER needs -- you would not know what "achieved goal" to relabel with, and you would have no function to recompute rewards for a different goal.

The Fetch interface was designed with HER in mind -- the dictionary observations, the separate goals, and the standalone `compute_reward` are all part of that design. The original HER paper (Andrychowicz et al., 2017) introduced these environments alongside the algorithm, specifically to provide a test bed where the relabeling mechanism works cleanly.

> **Note:** For a full treatment of HER's mechanism and the conditions under which it applies, see Chapter 5. For now, the important takeaway is practical: you have verified with your own hands that the Fetch environment supports goal relabeling, and you know exactly what that means -- calling `compute_reward` with goals the environment never set, and getting correct rewards back.

2.7 Build It: Cross-environment comparison

So far we have focused on FetchReach. But the Fetch family includes Push, PickAndPlace, and Slide, and the interface generalizes across all of them -- with important differences in the details.

How observations change across tasks

The key difference: environments with objects have larger observation vectors because they include the object's state (position, rotation, velocity).

```
# Cross-environment comparison
# (adapted from scripts/labs/env_anatomy.py:cross_env_compare)

def cross_env_compare(seed=0):
    """Compare obs dims across Reach, Push, PickAndPlace."""
    env_configs = {
        "FetchReach-v4": {"expected_obs_dim": 10,
                          "ag_is_grip": True},
        "FetchPush-v4": {"expected_obs_dim": 25,
                         "ag_is_grip": False},
        "FetchPickAndPlace-v4": {"expected_obs_dim": 25,
                                 "ag_is_grip": False},
    }
    results = {}
    for env_id, cfg in env_configs.items():
        env = gym.make(env_id)
        obs, _ = env.reset(seed=seed)
        obs_vec = np.asarray(obs["observation"])
        ag = np.asarray(obs["achieved_goal"])
        grip_pos = obs_vec[:3]
        ag_matches_grip = bool(np.allclose(ag, grip_pos))
        results[env_id] = {
            "obs_dim": obs_vec.shape[0],
            "ag_matches_grip": ag_matches_grip,
        }
        env.close()
    return results
```

Checkpoint. Expected output:

- `FetchReach-v4`: `obs_dim=10`, `ag_matches_grip=True`
- `FetchPush-v4`: `obs_dim=25`, `ag_matches_grip=False`
- `FetchPickAndPlace-v4`: `obs_dim=25`, `ag_matches_grip=False`

For Push and PickAndPlace, `achieved_goal` is the **object position**, not the gripper position. This is because the task goal is about where the object ends up, not where the gripper is. The gripper position is still available in `obs["observation"][:3]`.

The 25D observation for manipulation tasks

Environments with objects (Push, PickAndPlace) include 15 additional dimensions:

Index	Semantic	Source
0-2	Gripper position	`grip_pos`
3-5	Object position	`object_pos`
6-8	Object relative position (object - gripper)	`object_rel_pos`
9-10	Gripper finger positions	`gripper_state`
11-13	Object rotation (Euler angles)	`object_rot`
14-16	Object linear velocity (relative to gripper)	`object_velp`
17-19	Object angular velocity	`object_velr`
20-22	Gripper linear velocity	`grip_velp`
23-24	Gripper finger velocities	`gripper_vel`

The crucial thing to notice is that when `achieved_goal` changes from tracking the gripper (Reach) to tracking the object (Push, PickAndPlace), the goal-achievement mapping ϕ changes accordingly -- $\phi(s)$ is the gripper position for Reach but the object position for Push. The environment handles this transparently, so you do not need to change your code. But you need to understand it, because it affects what "success" means: in Push, the agent succeeds when the *object* (not the gripper) is within 5 cm of the target.

Uniform interface, changing semantics

Despite the dimension differences, the *interface* is identical across all Fetch environments:

Property	FetchReach	FetchPush	FetchPickAndPlace
`obs["observation"]` dim	10	25	25
`obs["achieved_goal"]` dim	3	3	3
`obs["desired_goal"]` dim	3	3	3
Action dim	4	4	4
`compute_reward` API	same	same	same
`distance_threshold`	0.05	0.05	0.05

This uniformity is what lets us develop algorithms on Reach (fast iteration, easy debugging) and then apply them to harder tasks. The same `MultiInputPolicy`, the same `compute_reward` invariant, the same HER relabeling -- all transfer directly. What changes is the task difficulty, not the interface.

In many RL domains, moving to a harder task means rewriting the environment wrapper, changing the observation preprocessing, and adjusting the reward function. In Fetch, you change one string -- the environment ID -- and everything else stays the same. The code that trains PPO on FetchReachDense-v4 in Chapter 3 will train SAC on FetchPush-v4 in Chapter 5 with no structural changes. The only differences will

be algorithmic (SAC instead of PPO, HER for goal relabeling) and in hyperparameters (more training steps for harder tasks).

2.8 Bridge: Manual inspection meets the production script

We have now inspected every major component of the Fetch environment by hand:

1. **Observations**: dictionary with three keys, shapes (10,), (3,), (3,) for Reach
2. **Actions**: 4D Cartesian deltas in [-1, 1], each axis produces movement in the expected direction
3. **Goals**: 3D Cartesian positions within the workspace; ϕ maps states to achieved goals
4. **Dense reward**: $-\|g_a - g_d\|$, matches `compute_reward` and `env.step()`
5. **Sparse reward**: 0 if distance $\leq$ 0.05, else -1, all three sources agree
6. **Relabeling**: `compute_reward` accepts arbitrary goals and returns correct rewards
7. **Cross-environment**: interface is uniform, observation dimensions and ϕ change with task complexity

The production script `scripts/ch01_env_anatomy.py` automates the same core checks for a single Fetch environment (describe, reward-check with relabeling, and a random-policy baseline) and writes the results as JSON artifacts. For the cross-environment comparison and the side-by-side dense vs sparse walkthroughs, use the lab script (`scripts/labs/env_anatomy.py --verify`).

You can run the bridging proof to confirm that the manual lab code and the production script agree:

```
bash docker/dev.sh python scripts/labs/env_anatomy.py --bridge
```

Expected output:

```
Environment Anatomy -- Bridging Proof
============================================================
Env: FetchReachDense-v4, seed=42, steps=100
Relabeled goals per step: 5
Tolerance: atol=1e-10

Step reward checks:    100 steps, mismatches=0
Relabel reward checks: 500 checks, mismatches=0

[MATCH] All rewards match within atol=1e-10
  manual_reward == compute_reward == step_reward  (100 steps)
  relabel_reward == -||ag - random_goal||  (500 checks)

[BRIDGE OK] Bridging proof passed
```

That gives us 100 steps with 5 relabeled goals per step, for 500 total reward checks -- and zero mismatches. The manual computation and the environment's `compute_reward` agree to within 1e-10 on every single check.

This bridging proof serves the same purpose as unit tests in software engineering: it confirms that your understanding (the manual computation) matches the implementation (the environment's API). When you run the production script in the next section and it reports ”OK,” you know exactly what ”OK” means -- because you have done the same checks by hand.

In later chapters, the bridging proof will be more dramatic: in Chapter 3, you will compare your from-scratch PPO loss computation against SB3's internal implementation. In Chapter 4, you will compare SAC update targets. Here, the bridge is simpler -- reward computation is just a distance calculation -- but the principle is the same: we find it valuable to verify any pipeline by hand before relying on it.

2.9 Run It: The inspection pipeline

Now we run the full production inspection. The script `scripts/ch01_env_anatomy.py` automates these checks for a single environment, producing machine-readable JSON artifacts.

```
------------------------------------------------------------
EXPERIMENT CARD: Fetch Environment Inspection
------------------------------------------------------------
Algorithm:    None (inspection and verification only)
Environment:  FetchReachDense-v4 (default; override with --env-id)

Run command (full inspection):
  bash docker/dev.sh python scripts/ch01_env_anatomy.py all \
    --seed 0

Time:         < 2 min (CPU or GPU)

Checkpoint track:
  N/A (no training; all commands produce results in seconds)

Expected artifacts:
  results/ch01_env_describe.json
  results/ch01_random_metrics.json

Success criteria:
  ch01_env_describe.json exists, contains observation_space
    with keys: observation (shape [10]), achieved_goal (shape [3]),
    desired_goal (shape [3])
  reward-check prints "OK:" (zero mismatches across 500 steps)
  ch01_random_metrics.json exists, success_rate near 0.0-0.1,
    return_mean in [-25, -10] for dense, ep_len_mean == 50
------------------------------------------------------------
```

By default, `--env-id auto` chooses FetchReachDense-v4 if it is available in your Gym registry. To inspect another task, pass an explicit ID (for example, `--env-id FetchReach-v4` or `--env-id FetchPush-v4`). If you run multiple inspections, also

override `--describe-out` and `--random-out` to avoid overwriting the default artifact paths.

Running the pipeline

The `all` subcommand runs three inspection stages:

Stage 1: Describe (`describe`). Creates `results/ch01_env_describe.json` documenting the observation and action spaces. This is the machine-readable version of section 2.3.

Stage 2: Reward check (`reward-check`). Runs 500 steps of random actions, verifying the critical invariant at each step: `env.step()` reward matches `compute_reward()` matches the distance formula. Also tests relabeling with random goals. This is the automated version of sections 2.5 and 2.6.

Stage 3: Random episodes (`random-episodes`). Runs 10 complete episodes with a random policy and records baseline metrics: success rate, mean return, mean episode length, and final goal distance. This establishes the performance floor.

Interpreting the artifacts

`results/ch01_env_describe.json` contains the observation and action space schema. Open it and verify:

- `observation_space.observation.shape == [10]`
- `observation_space.achieved_goal.shape == [3]`
- `observation_space.desired_goal.shape == [3]`
- `action_space.shape == [4]`
- `action_space.low == [-1, -1, -1, -1]`
- `action_space.high == [1, 1, 1, 1]`

`results/ch01_random_metrics.json` contains the random baseline. Expected values for FetchReachDense-v4:

Metric	Expected range	Meaning
`success_rate`	0.0-0.1	Random actions very rarely reach the goal
`return_mean`	-25 to -10	Sum of negative distances over 50 steps
`ep_len_mean`	50	Episodes always run to the truncation limit
`final_distance_mean`	0.05-0.15	Average distance to goal at episode end

These baseline numbers are your floor. In Chapter 3, PPO must produce a success rate well above 0.1 and a return much closer to 0. If a trained policy's metrics are not clearly better than these random baseline values, something is wrong with training.

The random baseline as diagnostic tool

The random baseline is more useful than it might appear. It answers a concrete question: ”how hard is this task for an agent that does nothing intelligent?” If random

success rate is already 20-30%, the task might be too easy to test your algorithm. If it is 0.0% over hundreds of episodes, the task is genuinely hard -- the agent must learn something specific to succeed.

For FetchReachDense-v4 with a threshold of 0.05 meters, random success is typically 0-10%. The workspace is much larger than the success region, so stumbling into the goal by chance is rare but not impossible. This makes Reach a good development environment: hard enough that random does not solve it, easy enough that basic algorithms (PPO with dense rewards) reliably learn it.

Note the `ep_len_mean` of 50, which confirms something we mentioned in section 2.2: episodes are truncated at 50 steps, never terminated early, so even when the agent reaches the goal the episode continues. This is a design choice in the Fetch environments -- the agent is rewarded for staying at the goal, not just reaching it. For dense rewards, staying at the goal produces rewards near 0 (distance near 0), which is the maximum possible. For sparse rewards, staying at the goal produces reward 0 (success), which is also the maximum. Both reward structures incentivize reaching and holding the goal position.

2.10 What can go wrong

Here are the most common issues when inspecting Fetch environments, along with diagnostics. We have encountered all of these during development.

`obs` is a flat array, not a dictionary

Symptom. `type(obs)` is `ndarray`, not `dict`. Calling `obs["observation"]` raises `TypeError`.

Cause. Old version of gymnasium-robotics (before 1.0), wrong environment ID, or a wrapper that flattens the dictionary.

Fix. Run `pip show gymnasium-robotics` and check the version is >= 1.0. Verify the environment ID matches a known Fetch environment. If you are wrapping the environment with SB3's `FlattenObservation`, remove that wrapper -- goal-conditioned environments must keep the dictionary structure.

`obs["observation"]` shape is not (10,) for FetchReach

Symptom. The observation has 25 dimensions instead of 10 (or some other unexpected number).

Cause. You are using FetchPush or FetchPickAndPlace (which have 25D observations), or there is a version mismatch.

Fix. Print `env.spec.id` to confirm the environment ID. Check `env.observation_space` for the full structure.

`compute_reward` raises `AttributeError`

Symptom. `env.compute_reward(...)` fails with `AttributeError: 'TimeLimit' object has no attribute 'compute_reward'`.

Cause. Calling on the wrapped environment instead of the unwrapped base environment. `gym.make()` wraps environments in `TimeLimit` and other wrappers that do not expose `compute_reward`.

Fix. Use `env.unwrapped.compute_reward(ag, dg, info)`.

Step reward and `compute_reward` disagree

Symptom. The three-way comparison shows mismatches. Manual reward, `compute_reward`, and `env.step()` return different values.

Cause. Version mismatch between gymnasium and gymnasium-robotics. Some versions changed the reward computation or the `compute_reward` API.

Fix. Upgrade both packages: `pip install --upgrade gymnasium gymnasium-robotics`. Then re-run the verification.

`desired_goal` is identical across resets

Symptom. Every call to `env.reset()` returns the same `desired_goal`.

Cause. Not passing different seeds, or passing the same seed every time.

Fix. Pass unique seeds: `env.reset(seed=42 + episode_number)`. If you want truly random goals, omit the seed argument.

`achieved_goal` does not match `obs["observation"][:3]` for FetchPush

Symptom. For FetchPush, `np.allclose(obs["achieved_goal"], obs["observation"][:3])` returns False.

Cause. This is correct behavior, not a bug. For Push and PickAndPlace, `achieved_goal` is the **object** position, not the gripper position. The gripper position is `obs["observation"][:3]`, but the goal is about where the object ends up.

Fix. No fix needed -- this is by design. See section 2.7 for the explanation.

`EnvironmentNameNotFound` for FetchReachDense-v4

Symptom. `gym.make("FetchReachDense-v4")` raises `gymnasium.error.NameNotFound`.

Cause. gymnasium-robotics is not installed, or the version does not include v4 environments.

Fix. Install with `pip install gymnasium-robotics`. If installed but v4 is not found, check the version -- v4 environments were introduced in gymnasium-robotics 1.2.0.

Random success rate much higher than 0.1

Symptom. Running 100 random episodes gives success_rate of 0.3 or higher.

Cause. The distance threshold might be larger than expected, or the goal space might be very small.

Fix. Check `env.unwrapped.distance_threshold` -- it should be 0.05. If it is different, your environment version uses a different default.

`is_success` is True immediately after reset

Symptom. The very first step of an episode reports `is_success: True` in the info dict, or the sparse reward is 0 right after reset.

Cause. The desired goal happened to be sampled at (or very near) the gripper's initial position. This is rare but normal -- it occurs in fewer than 5% of episodes.

Fix. No fix needed. Run multiple episodes and verify that success at reset is rare. If it happens consistently (every episode), there is likely a bug in goal sampling -- check that you are passing different seeds to `env.reset()`.

Workspace bounds look wrong

Symptom. Goal positions are near (0, 0, 0) or have very large values.

Cause. Different MuJoCo model version or coordinate frame issue.

Fix. Check `env.unwrapped.initial_gripper_xpos` -- typical values are around [1.34, 0.75, 0.53]. If these are very different, the MuJoCo model may have been modified or replaced.

2.11 Summary

You now understand the Fetch environment interface at the level needed to train and debug policies. Specifically:

- **Observations** are dictionaries with three keys: `observation` (proprioceptive state, 10D for Reach, 25D for Push/PickAndPlace), `achieved_goal` (where you are, 3D), and `desired_goal` (where you should be, 3D). This structure is what makes goal-conditioned learning explicit.
- **Actions** are 4D Cartesian deltas in [-1, 1]: three components for end-effector movement (dx, dy, dz) and one for gripper control. The agent operates in Cartesian space; an internal controller handles inverse kinematics.
- **Dense rewards** equal $-\|g_a - g_d\|$ -- negative distance. **Sparse rewards** are 0 if distance $\leq$ 0.05, else -1. Both can be recomputed for arbitrary goals via `compute_reward`.
- **The critical invariant** -- `env.step()` reward equals `compute_reward(ag, dg, info)` -- holds for all Fetch environments and is the foundation of HER.

- **Goal relabeling works**: calling `compute_reward` with goals the environment never set produces correct rewards, enabling the "what if that had been the goal?" trick at the core of HER.
- **The interface is uniform across Fetch tasks**: same dictionary structure, same action space, same `compute_reward` API. What changes is the observation dimension and what `achieved_goal` tracks (gripper for Reach, object for Push/PickAndPlace).
- **The random baseline** (success_rate 0.0-0.1, return_mean in [-25, -10] for dense Reach) is the performance floor that any trained agent must beat.

With this anatomy understood, Chapter 3 trains a real policy. PPO on FetchReachDense-v4 will use the observation shapes you documented here to build its network, the reward signal you verified to drive learning, and the random baseline you established as the metric it must surpass.

Verify It

```
VERIFY IT (Chapter 2)

Run command:
  bash docker/dev.sh python scripts/ch01_env_anatomy.py all --seed 0

Hardware:  Any machine with Docker (no GPU required)
Time:      < 2 min

Artifacts:
  results/ch01_env_describe.json
  results/ch01_random_metrics.json

Expected highlights (FetchReachDense-v4):
  describe: observation [10], achieved_goal [3], desired_goal [3], action [4], bou
  reward-check: OK: reward checks passed (atol=1e-6, n_steps=500, n_random_goals=3
  random-episodes (10): success_rate 0.0-0.1, return_mean -25 to -10, ep_len_mean 5

Optional lab checks:
  bash docker/dev.sh python scripts/labs/env_anatomy.py --verify
  bash docker/dev.sh python scripts/labs/env_anatomy.py --bridge
```

Exercises

1. (Verify) Confirm observation structure across seeds.

Reset `FetchReachDense-v4` with 5 different seeds. For each reset, verify that the observation dictionary has the same three keys and the same shapes. Record the range of `desired_goal` values across resets.

Expected: goals span the workspace (roughly x 1.1-1.5, y 0.5-1.0, z 0.35-0.75). If all goals are identical, something is wrong with the seed handling -- check that you are passing different seeds to each `env.reset()` call.

2. (Tweak) Compare dense and sparse rewards on the same trajectory.

Create both `FetchReachDense-v4` and `FetchReach-v4` with the same seed. Take the same sequence of 50 random actions in both. Compare the reward sequences side by side. Questions:

- (a) Is the dense return always more negative than the sparse return?
- (b) What fraction of steps have sparse reward = 0?
- (c) At what distance does the sparse reward switch from -1 to 0?

Expected: sparse reward is 0 only when distance < 0.05. Dense return is typically -25 to -10; sparse return is typically -50 (all -1s, since random actions rarely reach the goal).

3. (Extend) Observation breakdown for FetchPush.

Create `FetchPushDense-v4` and inspect the 25D observation vector. Using the observation component table from section 2.7, identify:

- (a) Which indices correspond to the object position
- (b) What `achieved_goal` represents (hint: it is the object position, not the gripper position)
- (c) What happens to the object position when you take action `[0, 0, 0, 0]` for 50 steps -- does the object move?

Expected: the object stays roughly in place (gravity keeps it on the table); `achieved_goal` tracks the object, not the gripper.

4. (Challenge) Estimate success rate as a function of distance threshold.

Run 100 random episodes on `FetchReachDense-v4`. For each episode, record the final distance between `achieved_goal` and `desired_goal`. Then tabulate what the success rate *would be* at thresholds of 0.01, 0.02, 0.05, 0.10, and 0.20 meters. How does the ”difficulty” of the task change with the threshold?

Expected: at threshold 0.01, random success is roughly 0%; at 0.20, it may be 5-15%. The default threshold of 0.05 makes random success very rare (0-5%). This exercise gives you intuition for how the success threshold determines task difficulty -- a concept we revisit when discussing sparse rewards in Chapter 5.

\newpage

Part 2 -- Baselines That Debug Your Pipeline

\newpage

3 PPO on Dense Reach: Your First Trained Policy

This chapter covers:

- Deriving the PPO clipped surrogate objective from the policy gradient theorem -- why constraining the likelihood ratio prevents the catastrophic updates that plague vanilla policy gradient
- Implementing PPO from scratch: actor-critic network, Generalized Advantage Estimation (GAE), clipped policy loss, value loss, and the full update loop
- Verifying each component with concrete checks (tensor shapes, expected values, learning curves) before assembling the complete algorithm
- Bridging from-scratch code to Stable Baselines 3 (SB3): confirming that both implementations compute the same GAE advantages, and mapping SB3 TensorBoard metrics to the code you wrote
- Training PPO on FetchReachDense-v4 to 100% success rate, establishing the pipeline baseline that every future chapter builds on

In Chapter 2, you dissected the Fetch environment -- observation dictionaries, action semantics, dense and sparse reward computation, goal relabeling, and the random-policy baseline (0% success, mean return around -20). You now understand what the agent sees and what the numbers mean.

But understanding the environment is necessary and not sufficient. A random policy achieves 0% success, so you need an algorithm that converts observations into intelligent actions -- one that improves through experience. The question is: which algorithm, and how do you verify it is working?

This chapter introduces PPO (Proximal Policy Optimization), an on-policy algorithm that learns by clipping likelihood ratios to prevent destructive updates. You will derive the PPO objective, implement it from scratch (actor-critic network, GAE, clipped loss, value loss), verify each component, bridge to SB3, and train a policy that reaches 100% success on FetchReachDense-v4 (see Figure 3.1), thereby validating your entire training pipeline.

One note before we begin: PPO works here because dense rewards provide continuous gradient signal, but PPO is on-policy -- it discards all data after each update, wasting expensive simulation time. Chapter 4 introduces SAC, an off-policy algorithm that stores and reuses experience in a replay buffer, and that off-policy machinery is what Chapter 5 (Hindsight Experience Replay) requires when we tackle sparse rewards.

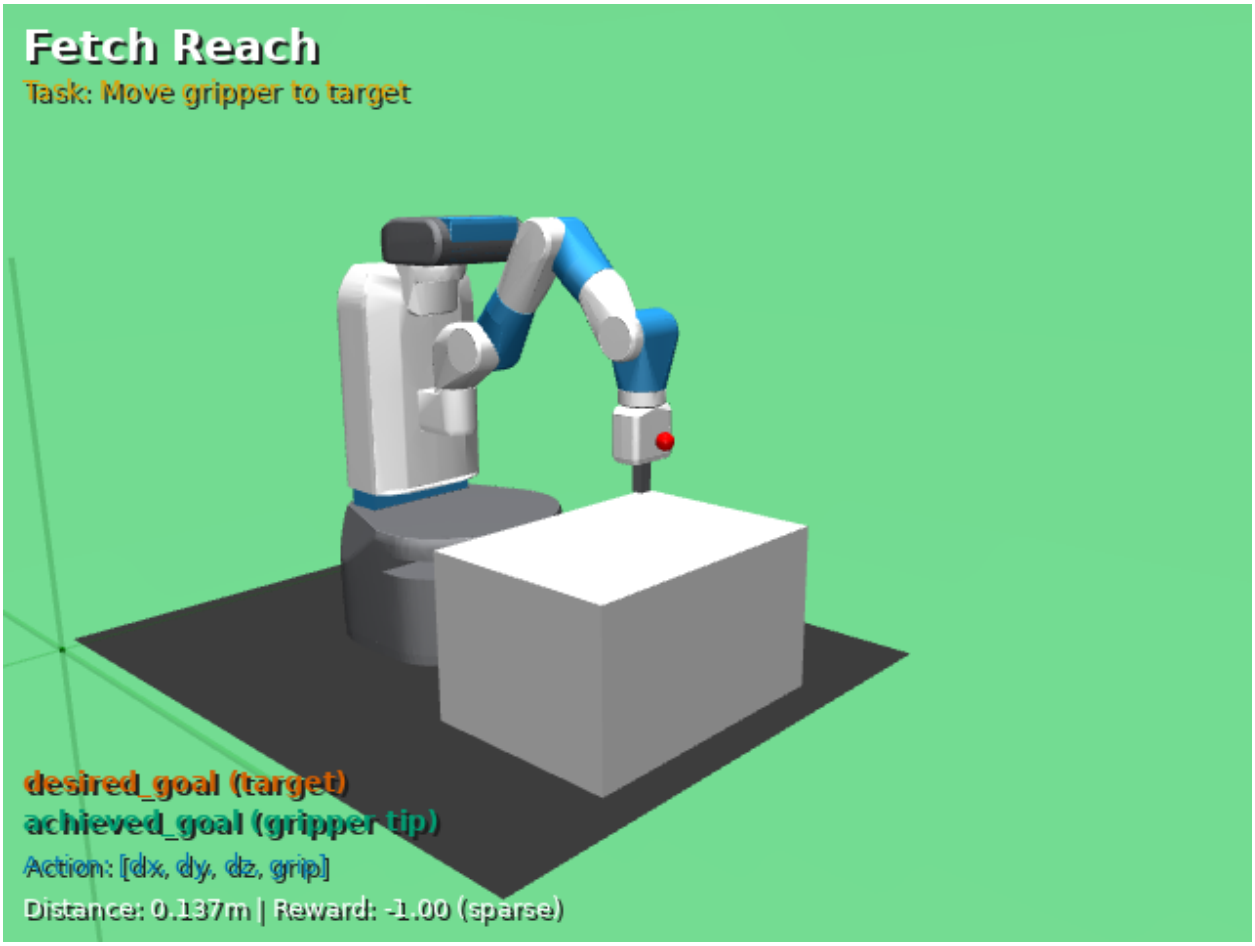

Figure 3.1: FetchReachDense-v4 -- the environment for this chapter. The robot arm must move its end-effector to the red target sphere. Dense rewards provide continuous feedback proportional to the distance between them, making this task well-suited for PPO as a first training baseline. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchReach-v4`. FetchReachDense-v4 is visually identical; only the reward differs.)

3.1 WHY: The learning problem

What are we optimizing?

Let's build up the math from intuition, defining each symbol as we introduce it.

At each timestep t, our **policy** π -- a neural network with parameters θ -- sees the current state s_t and the goal g. We bundle these into a single goal-conditioned input:

$$x_t := (s_t, g)$$

The policy outputs an action $a_t \sim \pi_\theta(\cdot \mid x_t)$. The environment responds with a new state s_{t+1} and a **reward** r_t -- a single number indicating how good that transition was.

Before stating the objective, we need three definitions.

Reward. The reward $r_t \in \mathbb{R}$ is the immediate feedback signal at timestep t. In FetchReachDense-v4, $r_t = -\|p_t - g\|_2$ where p_t is the gripper position and g is the goal. More negative means farther from the goal; zero means perfect.

Discount factor. The discount factor $\gamma \in [0, 1)$ determines how much we value future rewards relative to immediate rewards. A reward of magnitude r received k steps in the future contributes $\gamma^k r$ to our objective. With $\gamma = 0.99$, a reward 100 steps away is worth $0.99^{100} \approx 0.37$ as much as an immediate reward. This captures two things: sooner is better than later, and distant rewards are more uncertain.

Time horizon. The horizon T is the maximum number of timesteps in an episode. For FetchReach, $T = 50$ steps (we index $t = 0, \ldots, T-1$).

Now the objective. We want to find policy parameters θ that maximize the **expected discounted return**:

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

Here $J(\theta)$ measures how good a policy with parameters θ is, averaged over many episodes. The sum $G_t = \sum_{t=0}^{T-1} \gamma^t r_t$ is called the **return** -- the total reward accumulated over an episode, with future rewards discounted by γ.

The expectation is over trajectories -- different runs give different outcomes because actions sample from the policy distribution and the environment may be stochastic.

The challenge is: how do you take a gradient of this? The expectation depends on θ in a complicated way, since θ determines the policy, which determines the actions, which determines the states visited, which in turn determines the rewards.

The policy gradient theorem

Here is the key insight, stated informally:

> To improve the policy, increase the probability of actions that led to better-than-expected outcomes, and decrease the probability of actions that led to worse-than-expected outcomes.

The "better-than-expected" part is crucial. An action that got reward +10 is not necessarily good -- if you typically get +15 from that state, it was actually a bad choice.

This is captured by the **advantage function**:

$$A(x, a) = Q(x, a) - V(x)$$

where:

- $Q(x, a)$ is the **Q-function**: expected return if you take action a in goal-conditioned state x, then follow your policy

- $V(x)$ is the **value function**: expected return if you follow your policy from goal-conditioned state x

So $A(x, a) > 0$ means action a was better than average; $A(x, a) < 0$ means it was worse.

A concrete example helps here. If $V(x) = -0.3$ and $Q(x, a_{\text{left}}) = -0.1$, then $A(x, a_{\text{left}}) = +0.2$ -- moving left is better than the policy's average from this state. But notice: a positive advantage does NOT mean the action leads to a good outcome in absolute terms. The expected return is still $Q = -0.1$, which is negative. The advantage tells you the action is better *relative to what you normally do*, not that it is good in any absolute sense.

The **policy gradient theorem** (Sutton & Barto, 2018, Ch13.1) tells us how to differentiate the objective:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) \cdot A(x_t, a_t)\right]$$

Read this as: "Adjust θ to make good actions more likely and bad actions less likely, weighted by how good or bad they were."

The instability problem

In theory, you can follow this gradient and improve. In practice, vanilla policy gradient is notoriously unstable (Henderson et al., 2018), and two things go wrong.

Advantage estimates are noisy. We do not know the true advantage -- we estimate it from sampled trajectories. With a finite batch, these estimates have high variance, which means they are sometimes way off, leading to bad updates.

Big updates can be destructive. Suppose we estimate that some action is great ($A \gg 0$) and crank up its probability. If our estimate was wrong, we have committed to a bad action. Worse still, the new policy visits different states, making our old advantage estimates invalid, so the whole thing can spiral into a collapse that the policy never recovers from.

In our experience, unclipped policy gradient on Fetch tasks fails roughly half the time -- some seeds converge and others crash to zero and never recover. PPO's clipping mechanism was designed to make training reliably stable across seeds.

PPO's solution: constrained updates

PPO's key idea is: do not change the policy too much in one update (Schulman et al., 2017). But "too much" in what sense? Not in parameter space -- a small parameter change can cause large behavior change. Instead, PPO constrains change in probability space.

Define the **probability ratio** (also called the likelihood ratio):

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid x_t)}{\pi_{\theta_{\text{old}}}(a_t \mid x_t)}$$

This measures how the action likelihood changed:

- $\rho = 1$: same likelihood as before
- $\rho = 2$: action is now twice as likely
- $\rho = 0.5$: action is now half as likely

  Note (continuous actions). For Fetch, actions are continuous, so $\pi_\theta(a \mid x)$ is a probability density. The ratio is still well-defined as a likelihood ratio, and implementations compute it via log-probabilities: $\rho_t = \exp(\log \pi_\theta(a_t \mid x_t) - \log \pi_{\text{old}}(a_t \mid x_t))$.

PPO clips this ratio to stay in $[1 - \epsilon, 1 + \epsilon]$ (typically $\epsilon = 0.2$):

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min\left(\rho_t A_t,\ \text{clip}(\rho_t, 1 - \epsilon, 1 + \epsilon) \cdot A_t\right)\right]$$

What this does (also illustrated in Figure 3.2):

Advantage	Gradient wants to...	Clipping effect
$A > 0$ (good action)	Increase ρ (make action more likely)	Stops at $\rho = 1.2$
$A < 0$ (bad action)	Decrease ρ (make action less likely)	Stops at $\rho = 0.8$

The policy can improve, but only within a trust region around its current behavior. This prevents the catastrophic updates that kill vanilla policy gradient.

A concrete example

Let's trace through one update to make this tangible.

Imagine a discrete action space. The old policy assigns probability 0.3 to action a in state x. We estimate the advantage is $A = +2$ (this was a good action).

Naive approach. The gradient says ”make this action more likely!” So we update and now $\pi_\theta(a \mid x) = 0.6$. The ratio $\rho = 0.6/0.3 = 2.0$. We doubled the probability in one update. If our advantage estimate was wrong, we have made a big mistake.

PPO's approach. The clipped objective computes:

- Unclipped: $\rho \cdot A = 2.0 \times 2 = 4.0$
- Clipped: $\text{clip}(2.0, 0.8, 1.2) \times 2 = 1.2 \times 2 = 2.4$
- Objective: $\min(4.0, 2.4) = 2.4$

The gradient flows through the clipped version. We still increase the action probability, but the update is bounded. We cannot go from 0.3 to 0.6 in one step -- we would need multiple updates, each constrained.

Why dense rewards matter here

FetchReachDense-v4 gives reward $r_t = -\|g_a - g_d\|_2$ at every step -- the negative distance to the goal.

This helps PPO in three related ways. Every action provides signal -- ”you got 2cm closer” or ”you drifted 1cm away” -- so the algorithm always has gradient information. Since even random actions produce useful data (every distance tells you something), exploration is not a bottleneck. Most importantly, dense rewards decouple the exploration problem from the learning problem, which means any training issues on dense Reach point to the implementation rather than insufficient exploration, making debugging much more straightforward.

Compare this to sparse rewards ($R = 0$ if success, -1 otherwise), where most of your data carries no information about which direction to improve. We address that challenge in Chapter 5 with HER.

The well-posedness check

Before we train, it is worth asking the three well-posedness questions from Chapter 1.

First, **can this be solved?** The policy network has 16 inputs and 4 outputs, with ~5,600 parameters, and the mapping from ”see goal, move toward it” is well within the capacity of a small MLP. A hand-coded controller solves this task with a few lines of code, so a learned one certainly can.

Second, **is the solution reliable?** We expect yes, because the task is low-dimensional, the reward is smooth, and the goal distribution is bounded. Different seeds should therefore converge to qualitatively similar policies (move toward the goal), even if the exact parameters differ.

Third, **is the solution stable?** With dense rewards and PPO's clipping, small hyperparameter changes should not break convergence. We verify this empirically: three seeds with the same hyperparameters all reach 100% success (see Reproduce It).

These questions become more interesting for harder tasks. For dense Reach, the answers are reassuring -- which is precisely the point. We start here so that success validates the full pipeline, and any issues can be traced directly to implementation bugs rather than task difficulty.

3.2 HOW: The actor-critic architecture and training loop

The actor-critic architecture

PPO maintains two neural networks (or two heads of one network):

Actor $\pi_\theta(a \mid x)$: Given the goal-conditioned input $x = (s, g)$, output a probability distribution over actions. For continuous actions, this is a Gaussian with learned mean and standard deviation.

Critic $V_\phi(x)$: Given the same input, estimate the expected return. This is what we use to compute advantages.

Why two networks, not one? The actor maximizes expected return (finding good actions) while the critic minimizes prediction error (producing accurate value estimates), and these gradients can conflict so that improving one hurts the other. They also produce different output types -- the actor outputs a probability distribution (mean and variance for continuous actions), whereas the critic outputs a single scalar -- so forcing both through the same final layers creates unnecessary coupling. There is also a stability concern: the critic's value estimates feed into advantages, which train the actor, so if actor updates destabilize the critic, advantages become noisy, which destabilizes the actor further in a vicious cycle.

In practice, implementations often share early layers (a ”backbone”) with separate final layers (”heads”), capturing shared features while keeping the objectives separate. SB3 uses this approach by default.

The training loop

Here is the PPO training loop in pseudocode:

```
repeat:
    1. Collect N steps using current policy
    2. Compute advantages using critic (GAE)
    3. Update actor using clipped objective (multiple epochs)
    4. Update critic using MSE loss on returns
    5. Discard data, go to 1
```

Step 1: Collect data. Run the policy for `n_steps` in each of `n_envs` parallel environments, giving us `n_steps * n_envs` transitions to learn from.

Step 2: Compute advantages. We use Generalized Advantage Estimation (GAE), which balances bias and variance via a parameter λ. The full equation appears in Section 3.4, where we implement it.

Steps 3-4: Update networks. Unlike supervised learning, we do multiple passes over the same data -- `n_epochs = 10` is typical for PPO, with each pass using mini-batches of size `batch_size`. This reuses our expensive-to-collect trajectory data while the clipping prevents us from overfitting to it.

Step 5: Discard and repeat. This step reveals PPO's key tradeoff.

> **Definition (on-policy learning).** An algorithm is **on-policy** if it can only learn from data collected by the current policy π_θ. Every time you update θ, all transitions collected with the old parameters become invalid for computing unbiased gradients. You must throw away the data and collect fresh transitions with the new policy.

Here is what this means concretely. Each update cycle in PPO produces `n_steps * n_envs` transitions (8,192 with our settings), and you use these transitions for `n_epochs` gradient steps before discarding all of them -- even though many transitions contain useful information that could improve the policy further. This is sample-inefficient, since millions of simulation steps are thrown away after a single use.

Why does this matter for what comes next? The on-policy constraint is the main reason we move to SAC in Chapter 4. Off-policy methods store transitions in a replay buffer and reuse them across many updates, so every simulation step contributes to learning not once but repeatedly. More importantly, the replay buffer is what makes Hindsight Experience Replay (Chapter 5) possible, because you cannot relabel goals in data you have already discarded.

Key hyperparameters

Parameter	Our setting	What it controls
`n_steps`	1024	Trajectory length before update
`n_envs`	8	Parallel environments (throughput)
`batch_size`	256	Minibatch size for gradient updates
`n_epochs`	10	Passes over data per update
`learning_rate`	3e-4	Gradient step size
`clip_range`	0.2	PPO clipping parameter (ϵ)
`gae_lambda`	0.95	Advantage estimation bias-variance
`gamma`	0.99	Discount factor
`ent_coef`	0.0	Entropy bonus (exploration incentive)

For FetchReachDense-v4, SB3 defaults work well, and we find it helpful to verify the baseline with these default values first before experimenting with hyperparameter changes.

Compact equation summary

For reference, here are all the PPO equations in one place. Each appears again in the Build It sections alongside its implementation.

$$x_t := (s_t, g) \qquad J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

$$\delta_t = r_t + \gamma(1 - d_t)V_{\text{rollout}}(x_{t+1}) - V_{\text{rollout}}(x_t)$$

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l} \qquad \hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$$

$$\rho_t(\theta) = \pi_\theta(a_t \mid x_t) \;/\; \pi_{\theta_{\text{old}}}(a_t \mid x_t)$$

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(\rho_t \hat{A}_t,\ \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

$$L_{\text{value}} = \frac{1}{2}\mathbb{E}[(V_\phi(x_t) - \hat{G}_t)^2]$$

$$\mathcal{L} = -L^{\text{CLIP}} + c_1 L_{\text{value}} - c_2 \mathcal{H}[\pi]$$

3.3 Build It: The actor-critic network

Before we can compute losses, we need the network they operate on. We build PPO piece by piece, verifying each component before moving to the next. The implementations live in `scripts/labs/ppo_from_scratch.py` -- these are for understanding, not production.

The actor-critic network has a shared backbone (two hidden layers with tanh activations) and separate heads for the actor and critic:

```
# Actor-critic network with shared backbone
# (from scripts/labs/ppo_from_scratch.py:actor_critic_network)

class ActorCritic(nn.Module):
    def __init__(self, obs_dim: int, act_dim: int,
                 hidden_dim: int = 64):
        super().__init__()
        self.backbone = nn.Sequential(
            nn.Linear(obs_dim, hidden_dim), nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim), nn.Tanh(),
        )
        # Actor: mean of Gaussian policy
        self.actor_mean = nn.Linear(hidden_dim, act_dim)
        # Learnable log std (state-independent)
        self.actor_log_std = nn.Parameter(torch.zeros(act_dim))
        # Critic: scalar value estimate
        self.critic = nn.Linear(hidden_dim, 1)

    def forward(self, obs):
        features = self.backbone(obs)
        mean = self.actor_mean(features)
        std = self.actor_log_std.exp()
        dist = Normal(mean, std)
        value = self.critic(features).squeeze(-1)
        return dist, value
```

The actor outputs a Gaussian distribution parameterized by a learned mean and a state-independent log standard deviation, while the critic outputs a single scalar -- the estimated value $V(x)$. Both share the backbone features but have independent output layers.

The network is deliberately small, since Fetch tasks use MLPs rather than CNNs. For FetchReach, the input dimension is 16 (10D observation + 3D achieved goal + 3D

desired goal, concatenated by SB3's `MultiInputPolicy`), and the output is 4D (dx, dy, dz, gripper).

> **Checkpoint.** Instantiate the network with `obs_dim=16, act_dim=4` and run a forward pass with a random input. You should see: action mean shape `(1, 4)`, value shape `(1,)`, total parameters around 5,577. All outputs should be finite. If you get a shape error, check that `obs_dim` matches your concatenated observation size.

3.4 Build It: GAE computation

The advantage formula from Section 3.2 tells us how much better an action was compared to the policy's average behavior. Generalized Advantage Estimation (Schulman et al., 2015) computes this efficiently using a parameter λ that trades off bias and variance:

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}$$

where the **TD residual** (with termination masking) is:

$$\delta_t = r_t + \gamma(1 - d_t)V_{\text{rollout}}(x_{t+1}) - V_{\text{rollout}}(x_t)$$

Here $d_t \in \{0, 1\}$ indicates whether the episode terminated at timestep t. If it did, we do not bootstrap from x_{t+1} -- there are no future rewards to estimate. The value terms V_{rollout} are computed when collecting the rollout and treated as constants during optimization.

The λ parameter controls the bias-variance tradeoff:

- $\lambda = 0$: One-step TD. High bias (only looks one step ahead), low variance.
- $\lambda = 1$: Monte Carlo. Low bias (uses full trajectory), high variance.
- $\lambda = 0.95$: The typical default, and what we use.

In code, we compute GAE backwards through the trajectory:

```
# GAE computation (backward pass through trajectory)
# (from scripts/labs/ppo_from_scratch.py:gae_computation)

def compute_gae(rewards, values, next_value, dones,
                gamma=0.99, gae_lambda=0.95):
    T = len(rewards)
    advantages = torch.zeros(T, device=rewards.device)
    last_gae = 0.0

    for t in reversed(range(T)):
        if t == T - 1:
            next_val = next_value
```

```
        else:
            next_val = values[t + 1]
        next_val = next_val * (1.0 - dones[t])

        # TD residual: was this transition better than expected?
        delta = rewards[t] + gamma * next_val - values[t]

        # GAE recursion with episode boundary masking
        last_gae = (delta + gamma * gae_lambda
                    * (1.0 - dones[t]) * last_gae)
        advantages[t] = last_gae

    returns = advantages + values
    return advantages, returns
```

The key line is the GAE recursion: `last_gae = delta + gamma * gae_lambda * (1 - done) * last_gae`. This accumulates TD residuals backwards, decaying each by $\gamma\lambda$, and the `(1 - done)` term resets the accumulation at episode boundaries so that future advantages from a different episode do not bleed in.

The **returns** are then computed as `advantages + values`, which serve as the target for the value function: $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$.

Math	Code	Meaning
δ_t	`delta`	TD residual: was this transition better than expected?
γ	`gamma`	Discount factor (0.99)
λ	`gae_lambda`	Bias-variance tradeoff (0.95)
$\hat{A}_t$	`advantages[t]`	How much better was this action vs. average?
d_t	`dones[t]`	Episode terminated at this step?

> **Checkpoint.** Test with a trajectory where reward arrives only at the end: `rewards[-1] = 1.0`, all other rewards zero, `dones[-1] = 1.0`. The last advantage should be positive because the agent received a reward the value function did not fully predict. All advantages and returns should be finite. If the last advantage is negative or zero, check that the done mask is applied correctly -- the bootstrap value should be zeroed when the episode terminates.

3.5 Build It: The clipped surrogate loss

The PPO objective from Section 3.1:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(\rho_t A_t,\ \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon) \cdot A_t \right) \right]$$

In code, this maps directly to ratio computation, clipping, and a pessimistic (minimum) bound:

```python
# PPO clipped surrogate loss (part 1: ratio and clipping)
# (from scripts/labs/ppo_from_scratch.py:ppo_loss)

def compute_ppo_loss(dist, old_log_probs, actions,
                     advantages, clip_range=0.2):
    # Log prob under CURRENT policy
    new_log_probs = dist.log_prob(actions).sum(dim=-1)

    # Probability ratio via logs (numerically stable)
    log_ratio = new_log_probs - old_log_probs
    ratio = log_ratio.exp()

    # Clipped ratio: keep within [1-eps, 1+eps]
    clipped_ratio = torch.clamp(
        ratio, 1.0 - clip_range, 1.0 + clip_range)

    # Pessimistic bound: take the minimum
    surr1 = ratio * advantages
    surr2 = clipped_ratio * advantages
    policy_loss = -torch.min(surr1, surr2).mean()
```

The ratio is computed in log-space (`exp(log_new - log_old)`) for numerical stability, and the `min` operation is the pessimistic bound -- it takes the more conservative of the clipped and unclipped surrogate, ensuring we never overestimate the benefit of a policy change.

The function also returns diagnostics that track training health:

```python
    # PPO loss diagnostics (part 2)
    with torch.no_grad():
        approx_kl = ((ratio - 1) - log_ratio).mean()
        clip_fraction = (
            (ratio - 1.0).abs() > clip_range
        ).float().mean()

    info = {"policy_loss": policy_loss.item(),
            "approx_kl": approx_kl.item(),
            "clip_fraction": clip_fraction.item(),
            "ratio_mean": ratio.mean().item()}
    return policy_loss, info
```

The `clip_fraction` tells you what fraction of updates were clipped -- this is the same metric you will see in TensorBoard under `train/clip_fraction`. The `approx_kl` measures how much the policy diverged from the old policy in this update step.

Math	Code	Meaning
$\rho_t = \pi_\theta(a \mid x)/\pi_{\theta_{\text{old}}}(a \mid x)$	`ratio`	How much did action likelihood change?
ϵ	`clip_range`	Maximum allowed ratio change (0.2)

Math	Code	Meaning
A_t	`advantages`	Advantage estimates from GAE

Checkpoint. When the policy has not changed yet (same model, same parameters), all ratios should be 1.0 and no clipping should occur. Create a model, compute `old_log_probs` with `torch.no_grad()`, then immediately compute the loss with the same model. You should see: `clip_fraction = 0.000`, `ratio_mean = 1.000`, `approx_kl` near 0.000. If `clip_fraction` is nonzero, something is wrong -- the policy should not have changed between the two forward passes.

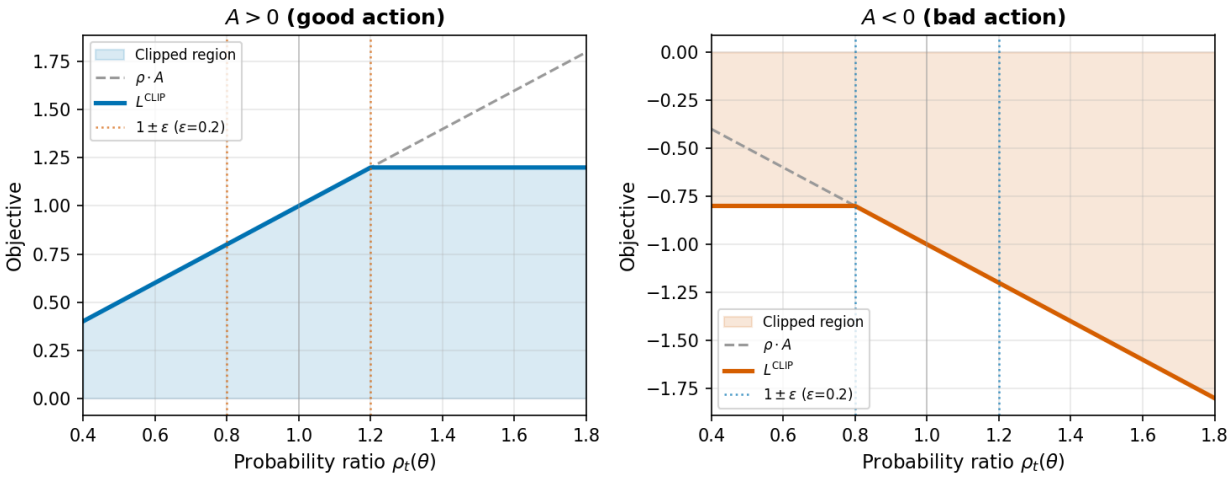

Figure 3.2: The PPO clipping mechanism. For a good action ($A > 0$, upper curve), the objective increases with the ratio ρ until the clip boundary at $1 + \epsilon = 1.2$, where the gradient is zeroed. For a bad action ($A < 0$, lower curve), the objective decreases with ρ until the clip boundary at $1 - \epsilon = 0.8$. The shaded region shows where clipping is active -- the policy can improve, but only within a trust region around its current behavior. (Generated by matplotlib in `scripts/labs/ppo_from_scratch.py`.)

3.6 Build It: The value loss

The critic learns to predict expected returns. We minimize the mean squared error between the critic's predictions $V_\phi(x_t)$ and the computed return targets $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$:

$$L_{\text{value}} = \frac{1}{2}\mathbb{E}\left[\left(V_\phi(x_t) - \hat{G}_t\right)^2\right]$$

```python
# Value function loss (critic update)
# (adapted from scripts/labs/ppo_from_scratch.py:value_loss)

def compute_value_loss(values, returns):
    value_loss = 0.5 * (values - returns).pow(2).mean()
```

```
    # Explained variance: how much of the return variance
    # does the critic explain?
    with torch.no_grad():
        var_target = returns.var()
        if var_target < 1e-8:
            ev = 0.0
        else:
            ev = (1.0 - (returns - values).var()
                  / var_target).item()

    info = {"value_loss": value_loss.item(),
            "explained_variance": ev}
    return value_loss, info
```

The **explained variance** is a useful diagnostic. It measures how much of the return variance the critic captures:

- EV = 1: perfect predictions
- EV = 0: predictions are no better than predicting the mean
- EV < 0: predictions are worse than predicting the mean

At initialization the critic predicts near-zero for everything, so explained variance starts near 0, but as training progresses it should climb toward 0.5-0.9. If it stays at 0 or goes negative, the critic is not learning -- check that the optimizer is attached to the critic's parameters.

> **Checkpoint.** Create near-zero value predictions and random returns. You should see `value_loss` around 0.5 (the predictions are wrong, so the squared error is roughly the variance of the returns) and `explained_variance` near 0.0 (the critic has no prediction skill yet). If `value_loss` is exactly 0, the critic and return tensors may be the same object -- check that you are using `.detach()` or separate computations.

3.7 Build It: PPO update (wiring)

The individual components above are combined into a single update step. PPO minimizes a combined loss:

$$\mathcal{L} = -L^{\text{CLIP}} + c_1 \cdot L_{\text{value}} - c_2 \cdot \mathcal{H}[\pi]$$

where $c_1 = 0.5$ (value coefficient), c_2 is the entropy coefficient (0.0 for Fetch tasks -- exploration is not the bottleneck with dense rewards), and $\mathcal{H}[\pi]$ is the entropy of the policy distribution (higher entropy means more exploration).

```
# PPO update step (part 1: compute losses)
# (from scripts/labs/ppo_from_scratch.py:ppo_update)

def ppo_update(model, optimizer, batch,
```

```
            clip_range=0.2, value_coef=0.5,
            entropy_coef=0.0, max_grad_norm=0.5):
    dist, values = model(batch.observations)

    # Normalize advantages (reduces variance)
    adv = batch.advantages
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)

    # Individual losses
    policy_loss, p_info = compute_ppo_loss(
        dist, batch.old_log_probs, batch.actions,
        adv, clip_range)
    value_loss, v_info = compute_value_loss(
        values, batch.returns)
    entropy = dist.entropy().mean()
    entropy_loss = -entropy
```

This computes all three loss components independently. The advantage normalization (subtracting mean, dividing by standard deviation) is standard practice -- it reduces sensitivity to reward scale.

The combined loss and gradient step:

```
    # PPO update step (part 2: backprop and step)
    total_loss = (policy_loss
                  + value_coef * value_loss
                  + entropy_coef * entropy_loss)

    optimizer.zero_grad()
    total_loss.backward()
    grad_norm = nn.utils.clip_grad_norm_(
        model.parameters(), max_grad_norm)
    optimizer.step()

    return {**p_info, **v_info,
            "entropy": entropy.item(),
            "total_loss": total_loss.item(),
            "grad_norm": grad_norm.item()}
```

A few things to notice. Before computing the policy loss, we normalize advantages by subtracting the mean and dividing by the standard deviation -- this is standard practice that reduces sensitivity to reward scale and makes gradient magnitudes more consistent across batches. We also clip the gradient norm at `max_grad_norm=0.5`, which prevents a single bad batch from causing an explosively large update. This gradient clipping is separate from PPO's ratio clipping: gradient clipping limits the step size in parameter space, while PPO clipping limits the step size in probability space. Finally, we negate the entropy because we minimize the total loss but want to *maximize* entropy. With `entropy_coef=0.0`, this term has no effect -- for FetchReachDense, the

dense reward signal is enough to drive learning without an explicit exploration bonus.

> **Checkpoint.** Run 10 updates on a mock batch (random observations, actions, advantages, returns). The value loss should decrease from its initial value -- the critic is learning to predict returns. The `approx_kl` should stay small (below 0.05) -- the clipping is preventing overly large policy changes. If the value loss does not decrease, verify that `optimizer.step()` is being called and that the model parameters are actually changing.

3.8 Build It: The training loop

The training loop wires together data collection, GAE computation, and the PPO update. The `collect_rollout` function runs the policy in the environment, and `transitions_to_batch` assembles the collected data into a training batch:

```python
# Rollout collection and batch assembly
# (from scripts/labs/ppo_from_scratch.py:ppo_training_loop)

def collect_rollout(env, model, n_steps, device):
    transitions, episode_returns = [], []
    obs, _ = env.reset()
    obs_t = torch.FloatTensor(obs).unsqueeze(0).to(device)
    ep_return = 0.0

    for _ in range(n_steps):
        with torch.no_grad():
            dist, value = model(obs_t)
            action = dist.sample()
            log_prob = dist.log_prob(action).sum(-1)
        # ... step env, store transition, handle resets
        # (full code in scripts/labs/ppo_from_scratch.py)

    with torch.no_grad():
        _, next_value = model(obs_t)
    return transitions, next_value.squeeze(), episode_returns
```

The full implementation handles environment resets on episode boundaries, stores all tensors needed for the PPO update, and computes the bootstrap value for the final state. The outer training loop looks like:

```
for each iteration:
    transitions, next_value, ep_returns = collect_rollout(...)
    batch = transitions_to_batch(transitions, next_value)
    for epoch in range(n_epochs):
        shuffle indices
        for each minibatch:
            info = ppo_update(model, optimizer, minibatch)
    log metrics
```

You can see this in action by running the demo mode, which trains PPO from scratch

on CartPole-v1 (~30 seconds on CPU). Figure 3.3 shows the resulting learning curve:

```
bash docker/dev.sh python scripts/labs/ppo_from_scratch.py --demo
```

Here are the results we got (your numbers may vary slightly with different seeds):

Iteration	Steps	Avg Return	Value Loss	What's happening
1	2k	22	7.6	Random behavior
5	10k	45	25.9	Starting to balance
10	20k	64	57.6	Improving steadily
15	31k	129	32.0	Getting close
18	37k	254	14.2	Solved (threshold: 195)

CartPole is a much easier task than FetchReach, but it demonstrates that the algorithm works end-to-end: the policy improves, the value loss eventually decreases, and the KL divergence stays bounded (typically below 0.05). This is the same algorithm SB3 uses; the from-scratch version just makes every step explicit.

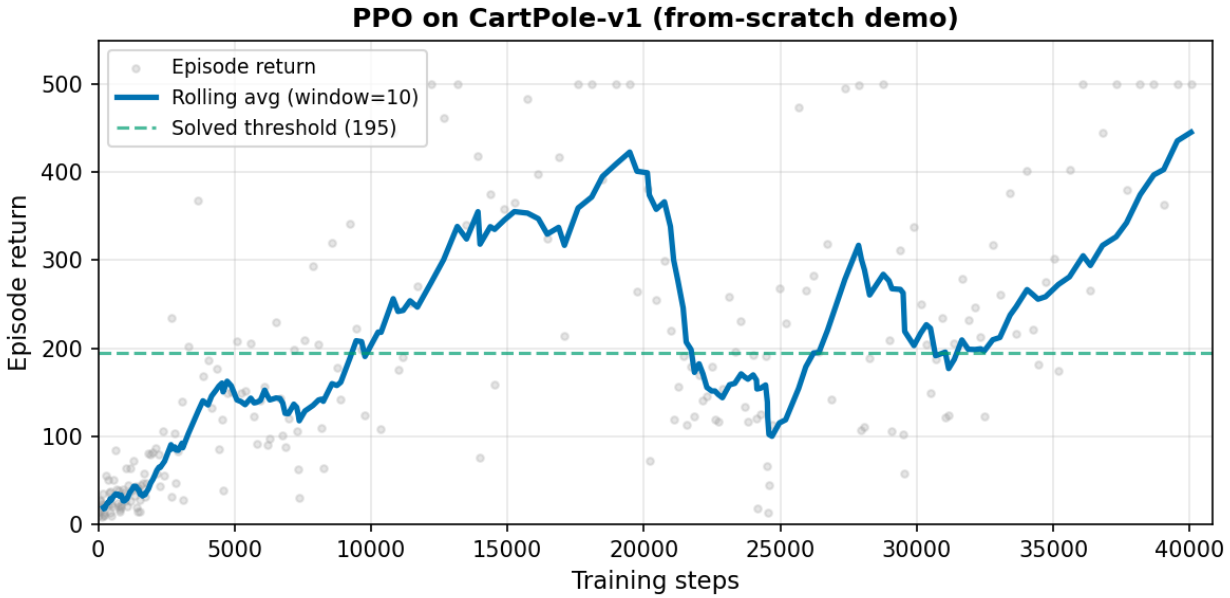

Figure 3.3: Learning curve from the from-scratch PPO implementation on CartPole-v1. The average episode return rises from ~22 (random behavior) to above 250 (solved) within 40k steps. The curve is noisy because on-policy methods discard data after each update, but the upward trend is clear. This validates that our implementation computes correct gradients. (Generated by `python scripts/labs/ppo_from_scratch.py --demo`.)

Checkpoint. Run `bash docker/dev.sh python scripts/labs/ppo_from_scratch.py --verify` to exercise all components end-to-end. Expected output:

```
============================================================
PPO From Scratch -- Verification
============================================================
```

```
Verifying actor-critic network...
  [PASS] Actor-critic network OK
Verifying GAE computation...
  [PASS] GAE computation OK
Verifying PPO loss...
  [PASS] PPO loss OK
Verifying value loss...
  [PASS] Value loss OK
Verifying PPO update...
  [PASS] PPO update OK
============================================================
[ALL PASS] PPO implementation verified
============================================================
```

This runs on CPU in under 2 minutes. If any check fails, the error message tells you which component and what went wrong.

3.9 Bridge: From-scratch to SB3

We have built PPO from scratch and verified it component by component. SB3 implements the same math in a production-grade library, so before we use SB3 for the real training run, let us confirm that the two implementations agree on the same computation.

The bridging proof feeds the same random data (rewards, values, dones) through our `compute_gae` and through SB3's `RolloutBuffer.compute_returns_and_advantage`, with both using gamma=0.99, gae_lambda=0.95, and the same seed:

```
bash docker/dev.sh python scripts/labs/ppo_from_scratch.py --compare-sb3
```

Expected output:

```
============================================================
PPO From Scratch -- SB3 Comparison
============================================================
Max abs advantage diff: ~0
Max abs returns diff:   ~0
[PASS] Our GAE matches SB3 RolloutBuffer
```

The two implementations produce identical advantages and returns within floating-point precision (tolerance 1e-6), which confirms that the same math drives both codebases.

What SB3 adds beyond our from-scratch code

Our implementation handles one environment at a time. SB3 adds engineering features that matter for real training:

- **Vectorized environments.** n_envs=8 parallel environments collect data simultaneously, increasing throughput by roughly 8x without algorithmic changes.

- **Learning rate scheduling.** SB3 can anneal the learning rate over training. We used a fixed 3e-4; SB3 defaults to the same but supports schedules.
- **Multi-epoch minibatch shuffling.** SB3 shuffles indices each epoch and handles batching efficiently. Our implementation does the same thing in a more explicit loop.
- **MultiInputPolicy.** SB3 handles dictionary observations automatically -- it builds separate encoders for each key (observation, achieved_goal, desired_goal) and concatenates them. Our from-scratch code assumed a flat observation vector.

Mapping SB3 TensorBoard metrics to our code

When you train with SB3 and open TensorBoard, the logged metrics correspond directly to the functions we just implemented:

SB3 TensorBoard key	Our function	What it measur
`train/value_loss`	`compute_value_loss`	Critic prediction
`train/clip_fraction`	`compute_ppo_loss` -> `info["clip_fraction"]`	Fraction of upd
`train/approx_kl`	`compute_ppo_loss` -> `info["approx_kl"]`	Policy divergen
`train/entropy_loss`	`dist.entropy().mean()`	Policy randomn
`train/policy_gradient_loss`	`compute_ppo_loss` -> `info["policy_loss"]`	Clipped surroga
`rollout/ep_rew_mean`	(environment)	Mean episode r

This mapping is worth internalizing. When you see `train/clip_fraction = 0.15` in TensorBoard, you know that 15% of the probability ratios exceeded the $[0.8, 1.2]$ clip range and those updates were constrained -- the number comes from the same calculation as the `clip_fraction` in our `compute_ppo_loss`. Having built these components yourself, you can read every TensorBoard metric as a quantity you understand from the inside.

3.10 Run It: Training PPO on FetchReachDense-v4

> **A note on script naming.** The production script is called `ch02_ppo_dense_reach.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, PPO on dense Reach is chapter 2; in this book, it is chapter 3. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch02`. This is intentional -- renaming would break the tutorial infrastructure. When you see `ch02` in a command or file path, you are running the right script for this chapter.

```
------------------------------------------------------------
EXPERIMENT CARD: PPO on FetchReachDense-v4
------------------------------------------------------------
Algorithm:    PPO (clipped surrogate, on-policy)
Environment:  FetchReachDense-v4
Fast path:    500,000 steps, seed 0
Time:         ~5 min (GPU) / ~30 min (CPU)
```

```
Run command (fast path):
  bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all \
    --seed 0 --total-steps 500000

Checkpoint track (skip training):
  checkpoints/ppo_FetchReachDense-v4_seed0.zip

Expected artifacts:
  checkpoints/ppo_FetchReachDense-v4_seed0.zip
  checkpoints/ppo_FetchReachDense-v4_seed0.meta.json
  results/ch02_ppo_fetchreachdense-v4_seed0_eval.json
  runs/ppo/FetchReachDense-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.90
  mean_return > -10.0
  final_distance_mean < 0.02

Full multi-seed results: see REPRODUCE IT at end of chapter.
------------------------------------------------------------
```

Running the experiment

The one-command version:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all --seed 0
```

This runs training, evaluation, and generates all artifacts in about 5-10 minutes on a GPU. For a quick sanity check that finishes in about 1 minute:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py train --total-steps 5000
```

If you are using the checkpoint track (no training), the pretrained checkpoint is at `checkpoints/ppo_FetchReachDense-v4_seed0.zip`. You can evaluate it directly:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py eval \
  --ckpt checkpoints/ppo_FetchReachDense-v4_seed0.zip
```

Training milestones

Watch for these milestones during training:

Timesteps	Success Rate	What's happening
0-50k	5-10%	Random exploration, policy is not yet useful
50k-100k	30-50%	Policy starting to move toward goals
100k-200k	70-90%	Rapid improvement phase
200k-500k	95-100%	Fine-tuning, convergence

Our test run achieved 100% success rate after 500k steps, with an average goal distance of 4.6mm (the success threshold is 50mm), and throughput of approximately 1,300 steps/second on an NVIDIA GB10.

Reading TensorBoard

Launch TensorBoard to watch training in real time:

```
bash docker/dev.sh tensorboard --logdir runs --bind_all
```

Then open http://localhost:6006 in your browser. Here is what healthy training looks like:

Metric	Expected behavior	What to watch for
`rollout/ep_rew_mean`	Steadily increasing (less negative)	Should move from around -20 tow
`rollout/success_rate`	0 -> 1 over training	The primary success metric
`train/value_loss`	High initially, then decreasing	Critic is learning to predict returns
`train/approx_kl`	Small (< 0.03), occasional spikes OK	Measures how much the policy ch
`train/clip_fraction`	0.1-0.3	Some updates clipped, not all
`train/entropy_loss`	Slowly moves toward 0	Policy becoming more determinist

Remember that these are the same quantities you implemented in the Build It sections -- `train/value_loss` is the output of your `compute_value_loss`, and `train/clip_fraction` comes from your `compute_ppo_loss`. You know exactly what these numbers mean because you have computed them yourself.

Verifying results

After training completes, check the evaluation JSON:

```
cat results/ch02_ppo_fetchreachdense-v4_seed0_eval.json | python -m json.tool | hea
```

Key fields to verify:

```
{
  "aggregate": {
    "success_rate": 1.0,
    "return_mean": -0.40,
    "final_distance_mean": 0.0046
  }
}
```

The passing criteria are success rate above 90%, mean return above -10, and final distance below 0.02 meters. Our runs consistently exceed these thresholds; if yours do not, see What Can Go Wrong below.

What the trained policy does

The trained network maps the 16D concatenated observation (10D proprioceptive + 3D achieved goal + 3D desired goal) to 4D actions (dx, dy, dz, gripper). It has learned that to reach a goal, it should output velocities that point toward the goal position -- effectively subtracting its current position from the desired position and scaling appropriately. The network discovered this purely from trial and error, using the dense reward signal as guidance.

In practice, the robot arm starts at a default position, and at each timestep the policy sees the current gripper position (in `achieved_goal`) and the target position (in `desired_goal`), then outputs a 4D action that moves the gripper toward the target. Within about 10-15 steps (out of 50 per episode) the gripper reaches the target and holds position for the remaining steps. The gripper dimension (index 3) is largely irrelevant for Reach since there is nothing to grasp, so the policy typically outputs near-zero values for it.

The final distance of 4.6mm (0.0046 meters) means the policy overshoots the target by less than 5mm on average. The success threshold is 50mm (0.05 meters), so the policy is roughly 10x more precise than required. This margin gives us confidence that the solution is robust, not barely passing.

Why this validates your pipeline

If PPO succeeds on dense Reach, you know that the environment is configured correctly (observations and actions have the right shapes and semantics), that the network architecture works (`MultiInputPolicy` correctly processes dictionary observations), that GPU acceleration works (training completes in reasonable time), that the evaluation protocol is sound (you can load checkpoints and run deterministic rollouts), and that metrics are computed correctly (success rate matches what you observe).

This is the pipeline baseline. Every future chapter builds on this infrastructure, so when something goes wrong with SAC in Chapter 4 or HER in Chapter 5, you can always come back here and verify that the foundation still works.

3.11 What can go wrong

Here are the failure modes we have encountered, organized by symptom. For each, we give the likely cause and a specific diagnostic.

`ep_rew_mean` flatlines near -20 for the entire run

Likely cause. The environment is misconfigured -- wrong observation or action shapes, or the policy network is not receiving goal information.

Diagnostic. Print the observation structure:

```
obs, _ = env.reset()
print({k: v.shape for k, v in obs.items()})
```

You should see `observation (10,)`, `achieved_goal (3,)`, `desired_goal (3,)`. Also verify that SB3 is using `MultiInputPolicy`, not `MlpPolicy` -- the latter cannot handle dictionary observations and will fail silently.

Success rate stays at 0% after 200k steps

Likely cause. You are using the wrong environment ID -- `FetchReach-v4` (sparse) instead of `FetchReachDense-v4` (dense). PPO cannot learn effectively from sparse rewards alone on this task.

Diagnostic. Print the reward from a random step. Dense rewards should be in the range [-1, 0] (negative distance). Sparse rewards are exactly 0 or -1. If you see only 0s and -1s, you have the sparse variant.

`value_loss` explodes (above 100) early in training

Likely cause. The reward scale is unexpected, or the GAE returns are not computed correctly.

Diagnostic. Check the reward range with a random policy. FetchReachDense rewards should be in [-1, 0]. If you see rewards on the order of -1000, something is misconfigured. Also check that GAE returns fall in a reasonable range (roughly [-50, 0] for FetchReachDense).

`approx_kl` consistently above 0.05

Likely cause. The learning rate is too high -- the policy is changing too fast per update.

Diagnostic. Reduce `learning_rate` from 3e-4 to 1e-4, or reduce `n_epochs` from 10 to 5. Either change limits how much the policy can move per update cycle.

`clip_fraction` near 1.0 every update

Likely cause. Updates are too aggressive. Nearly all actions are being clipped.

Diagnostic. Reduce the learning rate. If that does not help, reduce `clip_range` from 0.2 to 0.1. Also verify that advantages are normalized (subtracting mean, dividing by standard deviation) -- unnormalized advantages with large magnitudes can push ratios far from 1.0.

`clip_fraction` always 0.0

Likely cause. The policy is not learning. The learning rate may be too low, or the optimizer may not be attached to the model parameters.

Diagnostic. Check that `grad_norm` is nonzero in the training logs. If it is zero, gradients are not flowing -- verify that the loss tensor is connected to the model parameters (no `.detach()` in the wrong place).

`entropy_loss` immediately goes to 0

Likely cause. The policy collapsed to deterministic -- the `log_std` parameter went to negative infinity.

Diagnostic. Add an entropy coefficient: `ent_coef=0.01`. This penalizes overly deterministic policies and keeps the standard deviation from collapsing.

Training very slow (below 300 fps on GPU)

Likely cause. The GPU may not be in use, or `n_envs` is too low.

Diagnostic. Run `nvidia-smi` inside the container and check for a python process using GPU memory. If the GPU is being used but throughput is still 500-1300 fps, that is actually normal -- RL training on Fetch is CPU-bound on MuJoCo simulation rather than GPU-bound on neural network operations, since with small networks (5k parameters) and batch sizes (256), GPU operations complete in microseconds while the CPU runs physics.

`--compare-sb3` shows mismatch above 1e-6

Likely cause. Episode boundary handling differs between our GAE and SB3's implementation.

Diagnostic. Check that the done mask is applied to both `next_value` and `last_gae` in the backward loop. SB3 uses an `episode_starts` convention that may align slightly differently with `dones`. If the mismatch is small (1e-5 to 1e-4), it is likely a floating-point precision issue and not a cause for concern.

Build It `--verify` fails on "Value loss should decrease"

Likely cause. The random seed produced an adversarial batch where 10 updates are not enough to improve predictions.

Diagnostic. Re-run -- the test uses random data and very occasionally hits an edge case. If the failure is persistent across multiple runs, check that `optimizer.step()` is being called and that model parameters are changing between iterations. You can verify this by printing the sum of parameters before and after: `sum(p.sum() for p in model.parameters())`.

3.12 Summary

This chapter derived PPO from first principles and built it from the ground up. Here is what you accomplished along the way:

- **The objective.** We want to maximize expected discounted return $J(\theta)$. The policy gradient theorem tells us how to differentiate this, and the advantage function tells us which actions are better than average.
- **The instability problem.** Vanilla policy gradient is unstable because noisy advantage estimates can cause destructively large policy updates. PPO's clipped

surrogate objective constrains the probability ratio ρ_t to $[1-\epsilon, 1+\epsilon]$, preventing catastrophic updates while still allowing improvement.

- **From-scratch implementation.** You built six components: the actor-critic network, GAE advantage computation, the clipped policy loss, the value loss, the combined update step, and the training loop. Each was verified with concrete checks before moving to the next.
- **Bridge to SB3.** The bridging proof confirmed that our GAE implementation and SB3's `RolloutBuffer` produce identical advantages. SB3 adds engineering features (vectorized environments, dict-observation handling, learning rate scheduling) but computes the same underlying math.
- **Pipeline validation.** PPO achieved 100% success rate on FetchReachDense-v4, establishing that the environment, network architecture, training loop, evaluation protocol, and GPU setup all work correctly.
- **On-policy limitation.** PPO discards all data after each update because it is on-policy. This is sample-inefficient -- millions of simulation steps are thrown away after a single use.

That last point is the gap that Chapter 4 addresses. SAC (Soft Actor-Critic) is off-policy: it stores transitions in a replay buffer and reuses them across many updates, so every simulation step contributes to learning not once but repeatedly. On FetchReachDense, you will see SAC converge faster than PPO using less total simulation. More importantly, the replay buffer is a prerequisite for Chapter 5's Hindsight Experience Replay (HER), which turns failed trajectories into learning signal by relabeling goals -- a technique that requires stored transitions to relabel.

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
The results and pretrained checkpoints in this chapter
come from these runs:

  for seed in 0 1 2; do
    bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all \
      --seed $seed --total-steps 1000000
  done

Hardware:    Any machine with Docker (GPU optional; tested on NVIDIA GB10)
Time:         ~8 min per seed (Linux GPU), ~45 min per seed (Mac/CPU)
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/ppo_FetchReachDense-v4_seed{0,1,2}.zip
```

```
  checkpoints/ppo_FetchReachDense-v4_seed{0,1,2}.meta.json
  results/ch02_ppo_fetchreachdense-v4_seed{0,1,2}_eval.json
  runs/ppo/FetchReachDense-v4/seed{0,1,2}/

Results summary (what we got):
  success_rate: 1.00 +/- 0.00  (3 seeds x 100 episodes)
  return_mean:  -0.40 +/- 0.05
  final_distance_mean: 0.005 +/- 0.001

If your numbers differ by more than ~5%, check the
"What Can Go Wrong" section above.

The pretrained checkpoints are available in the book's
companion repository for readers using the checkpoint track.
------------------------------------------------------------
```

Exercises

1. (Verify) Reproduce the single-seed baseline.

Run the fast path command and verify your results match the experiment card:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all \
  --seed 0 --total-steps 500000
```

Check the eval JSON: `success_rate` should be >= 0.90, `mean_return` > -10. Record your training time and steps per second -- you will compare these against SAC in Chapter 4.

2. (Tweak) GAE lambda ablation.

In the lab verification, modify `gae_lambda` and observe the effect on advantage magnitudes:

- `gae_lambda = 0.0` (one-step TD, high bias)
- `gae_lambda = 0.5` (midpoint)
- `gae_lambda = 1.0` (Monte Carlo, high variance)

Question: How do the advantage magnitudes change? Why does `lambda=0` produce smaller magnitude advantages?

Expected: `lambda=0` advantages are dominated by single TD residuals (one step of reward minus one step of value change). `lambda=1` advantages accumulate over the full trajectory, producing larger magnitudes and more variance between samples.

3. (Tweak) Clip range ablation.

Train PPO with different `clip_range` values (0.1, 0.2, 0.4) for 500k steps each. Compare:

- (a) Final success rate
- (b) Training stability (watch `approx_kl` in TensorBoard)
- (c) `clip_fraction` values

Expected: `clip_range=0.1` is more conservative -- the policy changes slowly per update, which may require more iterations but is stabler. `clip_range=0.4` allows larger changes per update, which risks instability but may converge faster on easy tasks. The default of 0.2 is a compromise that works reliably across many tasks.

4. (Extend) Add wall-clock time tracking.

Modify the eval report to include wall-clock training time and compute steps per second. Compare GPU versus CPU performance. Expected: GPU is roughly 2-5x faster, but the speedup is modest because the bottleneck is CPU-bound MuJoCo simulation, not GPU-bound neural network operations.

5. (Challenge) Train the from-scratch implementation on FetchReachDense.

Extend the `--demo` mode in `ppo_from_scratch.py` to work with `FetchReachDense-v4` instead of CartPole. You will need to handle dictionary observations (concatenate `observation` + `desired_goal` as the network input, giving a 13D vector) and continuous actions (remove the discrete threshold used for CartPole). Does it learn? How does it compare to SB3 in sample efficiency?

Expected: it should learn but more slowly than SB3 -- SB3 uses vectorized environments (8 parallel), optimized rollout storage, and proper observation preprocessing. With a single environment, your from-scratch implementation may reach 50-80% success rate in 500k steps. The performance gap is engineering, not algorithmic -- both implementations compute the same math.

\newpage

4 SAC on Dense Reach: Off-Policy Learning with Maximum Entropy

This chapter covers:

- Why deterministic policies are brittle -- and how the maximum entropy objective keeps exploration alive by rewarding high-entropy action distributions alongside high reward
- Implementing SAC from scratch: replay buffer, twin Q-networks with clipped double Q-learning, squashed Gaussian policy with tanh bounds, automatic temperature tuning, and the full update loop
- Verifying each component with concrete checks (tensor shapes, Q-value ranges, log-probability correctness) before assembling the complete algorithm
- Bridging from-scratch code to Stable Baselines 3 (SB3): confirming that both implementations compute the same squashed Gaussian log-probabilities, and mapping SB3 TensorBoard metrics to the code you wrote
- Training SAC on FetchReachDense-v4 to 100% success rate, matching PPO's performance while establishing the off-policy replay machinery that Chapter 5 (HER) requires

In Chapter 3, you derived the PPO clipped surrogate objective, implemented it from scratch, bridged to SB3, and trained a policy to 100% success on FetchReachDense-v4. The entire pipeline -- environment, network, training loop, evaluation -- is validated and working.

But PPO has a structural limitation that matters for what comes next. PPO is on-policy: every transition is used for a handful of gradient steps, then discarded. For FetchReachDense, where dense rewards provide continuous feedback at every timestep, this wastefulness is tolerable -- PPO still reaches 100% success in about 5 minutes. But each MuJoCo simulation step costs real CPU time. PPO achieves roughly 1,300 steps per second yet uses each frame only once. For harder tasks with sparser signal (coming in Chapter 5), reusing data becomes essential.

This chapter introduces SAC (Soft Actor-Critic), an off-policy algorithm that stores every transition in a replay buffer and reuses it across many updates. SAC adds a maximum entropy bonus that keeps the policy exploratory early in training and lets it become deterministic as it converges. You will derive the maximum entropy objective, implement SAC from scratch (replay buffer, twin Q-networks, squashed Gaussian policy, automatic temperature tuning), verify each component, bridge to SB3, and match PPO's 100% success on FetchReachDense-v4 -- validating the off-policy stack.

One note on why this matters beyond sample efficiency: SAC's replay buffer also enables a technique we will need in Chapter 5. HER (Hindsight Experience Replay) relabels failed transitions with alternative goals -- manufacturing success signal from failure. HER requires off-policy learning because relabeled data did not come from the current policy. The off-policy machinery you build in this chapter is the foundation HER needs.

4.1 WHY: The sample efficiency problem

The standard RL objective (review)

Recall from Chapter 3 that standard RL maximizes expected return:

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

where π_θ is the policy with parameters θ, r_t is the reward at timestep t, $\gamma \in [0, 1)$ is the discount factor (0.99 in our experiments), and $T = 50$ is the episode horizon. This finds a policy that accumulates high reward. But the optimal policy under this objective is deterministic -- once you know the best action for each state, there is no reason to do anything else.

Why determinism is a problem

In theory, a deterministic optimal policy is fine. In practice, it causes three problems that matter for robotics.

Problem 1: exploration dies. A deterministic policy exploits what it knows. If the current best action gets reward -0.1, the policy commits to it. But what if there is an action that would get reward -0.01 which the policy has never tried because it stopped exploring? With continuous action spaces (4D in Fetch), the chance of stumbling onto a good action by noise alone is small. The policy gets stuck in a local optimum.

Problem 2: brittleness. A policy that commits fully to one action per state is fragile. Small perturbations -- observation noise, model mismatch between simulation and hardware -- can push it into unfamiliar states where it has no idea what to do. In robotics, sim-to-real transfer regularly exposes this failure mode: a policy that works perfectly in simulation often fails on real hardware.

Problem 3: training instability. When the policy is nearly deterministic, small changes in value estimates cause large behavioral changes (the "winning" action flips). This amplifies noise in the training process and can lead to oscillating or diverging training curves.

The maximum entropy objective

SAC addresses these problems by modifying the objective. Instead of maximizing reward alone, we maximize reward plus an entropy bonus:

Definition (Maximum entropy objective).

Motivating problem. The standard RL objective produces deterministic policies that stop exploring, are brittle to perturbations, and cause training instability. We need a way to keep the policy stochastic -- preferring a spread of actions -- while still pursuing high reward.

Intuitive description. Instead of asking "which single action is best?", we ask "which distribution over actions gives the best tradeoff between reward and keeping our options open?" The policy prefers actions proportionally to how good they are, rather than committing entirely to the single best one.

Formal definition. The maximum entropy objective augments the standard return with an entropy bonus:

$$J_{\text{MaxEnt}}(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \left(r_t + \alpha \mathcal{H}(\pi_\theta(\cdot \mid s_t))\right)\right]$$

where $\mathcal{H}(\pi_\theta(\cdot \mid s_t)) = -\mathbb{E}_{a \sim \pi_\theta}[\log \pi_\theta(a \mid s_t)]$ is the **entropy** of the policy at state s_t -- a measure of how "spread out" the action distribution is. The **temperature parameter** $\alpha > 0$ controls the tradeoff: higher α favors exploration (high entropy), lower α favors exploitation (high reward). Setting $\alpha = 0$ recovers the standard objective.

Grounding example. Consider a state where two actions have Q-values $Q(s, a_1) = -0.1$ and $Q(s, a_2) = -0.3$. Under the standard objective, the policy assigns all probability to a_1. Under maximum entropy with $\alpha =$

0.2, the policy assigns probabilities proportional to $\exp(Q/\alpha)$: $\pi(a_1) \propto \exp(-0.1/0.2) = \exp(-0.5) \approx 0.61$ and $\pi(a_2) \propto \exp(-0.3/0.2) = \exp(-1.5) \approx 0.22$. After normalizing: $\pi(a_1) \approx 0.73$ and $\pi(a_2) \approx 0.27$. The better action is more likely, but the worse action is not eliminated -- the policy retains the ability to discover it was wrong about a_1.

Non-example. High entropy does NOT mean random. A policy with $\alpha > 0$ still strongly prefers high-Q actions -- it just does not go all-in on the single best one. As training proceeds and Q-estimates become accurate, α decreases toward zero and the policy becomes nearly deterministic. The entropy bonus is a training aid, not a permanent handicap.

The Boltzmann policy

The maximum entropy objective has a clean closed-form solution: the optimal policy assigns action probabilities proportional to exponentiated Q-values:

$$\pi^*(a \mid s) \propto \exp\left(Q^*(s,a)/\alpha\right)$$

This is a **Boltzmann distribution** (sometimes called a "softmax" over continuous actions). The temperature α controls the sharpness of the distribution:

- $\alpha \to 0$: the distribution collapses to a point mass on the best action (deterministic, like standard RL)
- $\alpha \to \infty$: the distribution approaches uniform (every action equally likely)

The Boltzmann form is what makes SAC elegant. Rather than adding exploration noise externally (like epsilon-greedy or Ornstein-Uhlenbeck noise in DDPG), the exploration behavior emerges naturally from the objective itself. The policy explores because exploring is rewarded.

Off-policy learning: reusing experience

Before we get to SAC's specific components, we need to formalize how it differs from PPO at a structural level.

> **Definition (Off-policy learning).** An algorithm is **off-policy** if it can learn from data collected by any policy -- including old versions of the current policy, a random policy, or a different agent entirely. Transitions are stored in a **replay buffer** and reused across many gradient updates. The key property is that the data distribution and the policy being optimized are decoupled.

Compare this to PPO's on-policy constraint (defined in Chapter 3): PPO can only use data from the current policy. Every time the parameters change, old data becomes invalid for computing unbiased gradients, so it must be discarded.

Here is what this means concretely. PPO collects 8,192 transitions, uses them for 10 gradient epochs, then throws them all away. SAC stores every transition in a replay buffer (capacity 1,000,000) and samples from it repeatedly. A transition collected at

step 10,000 might be sampled again at step 500,000 -- the same data contributes to learning hundreds of times.

This difference drives the entire chapter. Off-policy learning is more sample-efficient, since every simulation step contributes to many updates rather than just one. It is also more complex, because learning from stale data introduces challenges (overestimation bias, moving targets) that require new machinery (twin critics, target networks). And it is required for HER -- Chapter 5's goal relabeling modifies stored transitions after the fact, which is only possible if transitions are stored and reusable.

Automatic temperature tuning

Choosing α manually requires care. Too low and the policy stops exploring early, settling on a suboptimal solution. Too high and the policy spends time exploring randomly instead of exploiting what it has learned. The right value depends on the task, the training stage, and the action dimensionality.

SAC can learn α automatically by targeting a desired entropy level. The idea: define a **target entropy** $\bar{\mathcal{H}}$ (how stochastic you want the policy to be), and adjust α to keep the policy's actual entropy near that target.

The target entropy is typically set to $-\dim(\mathcal{A})$, the negative of the action dimensionality. For Fetch tasks with 4D actions, $\bar{\mathcal{H}} = -4$. This is a heuristic from Haarnoja et al. (2018b) -- roughly, one nat of entropy per action dimension is enough to maintain useful exploration without being overly random.

The temperature loss function is:

$$L(\alpha) = \mathbb{E}_{a \sim \pi}\left[-\alpha\left(\log \pi(a \mid s) + \bar{\mathcal{H}}\right)\right]$$

The gradient pushes α in the right direction: if the policy's entropy is below the target ($\log \pi$ is large and negative, but not negative enough), α increases to encourage more exploration, whereas if entropy is above the target, α decreases to allow more exploitation.

In practice, SAC learns $\log \alpha$ rather than α directly, ensuring α stays positive. We initialize $\log \alpha = 0$ (so $\alpha = 1.0$) and let the optimizer adjust it.

Why this matters for robotics

In robotics, robustness matters as much as performance. A policy needs to handle real sensor noise, not just clean simulation. The maximum entropy objective helps here in several reinforcing ways. First, the policy explores many actions during training, which means the critic sees more of the state-action space and develops more reliable value estimates. This broader coverage then supports more robust behaviors, since a policy that does not commit fully to a single action develops softer trajectories that tolerate perturbations -- in our experiments, the entropy bonus at training time translates to less jerky motion even after α has decreased near zero. Finally, the Boltzmann-style

policy changes gradually as Q-values change, which smooths the training process and avoids the oscillations that plague deterministic policy optimization.

4.2 HOW: The SAC algorithm

The components

SAC maintains five networks:

Network	Purpose	Updates
Actor π_θ	Maps states to action distributions (squashed Gaussian)	Policy gradient
Critic 1 Q_{ϕ_1}	Estimates Q-values	Bellman backup
Critic 2 Q_{ϕ_2}	Second Q estimate (reduces overestimation)	Bellman backup
Target Critic 1 $Q_{\bar{\phi}_1}$	Stable target for critic updates	Polyak averaging
Target Critic 2 $Q_{\bar{\phi}_2}$	Stable target for critic updates	Polyak averaging

Why two critics? Q-learning tends to overestimate values because we take a maximum over noisy estimates: $\mathbb{E}[\max(Q_1, Q_2)] \geq \max(\mathbb{E}[Q_1], \mathbb{E}[Q_2])$. Using two critics and taking the minimum for the target counteracts this -- it is called **clipped double Q-learning** (Fujimoto et al., 2018).

Why target networks? If we update the critic using its own predictions as targets, we get a moving-target problem -- the thing we are trying to match keeps changing. **Target networks** are slow-moving copies that provide stable optimization targets. They are updated via **Polyak averaging** (also called soft update):

$$\bar{\phi} \leftarrow \tau\phi + (1-\tau)\bar{\phi}$$

with $\tau = 0.005$ (the target network inherits 0.5% of the main network's weights per update). This keeps targets stable while slowly tracking the improving critic.

The training loop

```
repeat:
    1. Collect transition using current policy, store in replay buffer
    2. Sample minibatch from replay buffer
    3. Update critics: minimize Bellman error (target uses min of two Q-
targets)
    4. Update actor: maximize Q-value + entropy
    5. Update temperature alpha (if auto-tuning)
    6. Soft-update target networks (Polyak averaging)
```

Step 1: Collect data. Unlike PPO, which collects full trajectories before updating, SAC collects one transition at a time and updates after each step. The transition $(s, a, r, s', \text{done})$ goes into the replay buffer.

Step 2: Sample minibatch. Uniformly sample `batch_size` transitions from the buffer. This is where data reuse happens -- the same transition might be sampled many times across training.

Step 3: Update critics. For each critic, minimize the squared Bellman error against a soft target (detailed in Section 4.6).

Step 4: Update actor. Maximize expected Q-value plus entropy bonus (detailed in Section 4.7).

Steps 5-6: Temperature and target updates. Adjust α to maintain target entropy, then blend the main critic weights into the targets.

PPO versus SAC

Both algorithms solved FetchReachDense-v4 in Chapter 3 and this chapter respectively. Here is how they compare:

Aspect	PPO (Chapter 3)	SAC (this chapter)
Data reuse	None (on-policy: use once, discard)	Extensive (replay buffer, reuse many ti
Exploration	Optional entropy bonus (we used 0.0)	Core: entropy is in the objective
Sample efficiency	Low (discards data)	High (reuses data)
Stability	High (clipped updates bound change)	Medium (moving targets, overestimatio
Complexity	Lower (1 actor + 1 critic)	Higher (1 actor + 2 critics + 2 targets +
Wall-clock speed	Faster per step (~1,300 fps)	Slower per step (~594 fps)

The throughput difference comes from network count: SAC updates five networks per step versus PPO's one shared network. But SAC compensates by extracting more learning from each transition. The tradeoff is clear: PPO is simpler and faster per step; SAC is more complex but more efficient per sample. For the tasks ahead -- especially sparse rewards with HER in Chapter 5 -- sample efficiency wins.

Key hyperparameters

Parameter	Default	What it controls
`buffer_size`	1,000,000	Replay buffer capacity (how far back we remember)
`batch_size`	256	Minibatch size for each gradient update
`learning_starts`	10,000	Steps of random data collection before training begins
`tau`	0.005	Target network update rate (Polyak averaging)
`ent_coef`	”auto”	Entropy temperature (auto-tuned by default)
`learning_rate`	3e-4	Gradient step size for all optimizers
`gamma`	0.99	Discount factor

For FetchReachDense-v4, SB3 defaults work well. We recommend training with these values first and only tuning if something goes wrong (see What Can Go Wrong later in this chapter).

Figure 4.1 shows the full SAC architecture -- how data flows from the environment through the replay buffer and into the three update steps.

SAC architecture diagram showing the data flow: environment transitions flow into a circular replay buffer, minibatches are sampled to update the twin Q-networks (critic loss), the actor (policy loss), and the temperature alpha, with target networks updated via Polyak averaging

Figure 4.1: The SAC architecture. Transitions flow from the environment into a circular replay buffer. At each update step, a minibatch is sampled and used to update three components: the twin Q-networks (minimizing Bellman error against slow-moving target networks), the actor (maximizing Q-value plus entropy), and the temperature α (maintaining target entropy). Target networks track the main critics via Polyak averaging ($\tau = 0.005$). (Illustrative diagram.)

4.3 Build It: Replay buffer

Off-policy learning requires storing transitions for reuse. Unlike PPO, which discards data after each update, SAC stores every transition in a circular buffer and samples from it repeatedly.

A **replay buffer** is a fixed-size array in which new transitions overwrite the oldest when the buffer is full:

```
[t_0, t_1, t_2, ..., t_{n-1}, t_n, t_{n+1}, ...]
 ^-- oldest                          ^-- newest
```

With a capacity of 1,000,000 and 1M training steps, each transition is stored once but sampled many times. The buffer size controls the "memory horizon" -- how far back the agent remembers. Too small and you lose useful old experience; too large and you dilute recent (better) experience with outdated data.

Each entry stores a five-tuple: $(s, a, r, s', \text{done})$ -- the state, action, reward, next state, and termination flag. Sampling is uniform random: every transition in the buffer has an equal chance of being drawn into a minibatch.

```python
# Replay buffer for off-policy learning
# (from scripts/labs/sac_from_scratch.py:replay_buffer)

class ReplayBuffer:
    def __init__(self, obs_dim, act_dim, capacity=100000):
        self.capacity, self.ptr, self.size = capacity, 0, 0
        self.obs = np.zeros((capacity, obs_dim), dtype=np.float32)
        self.actions = np.zeros((capacity, act_dim), dtype=np.float32)
        self.rewards = np.zeros(capacity, dtype=np.float32)
        self.next_obs = np.zeros((capacity, obs_dim), dtype=np.float32)
        self.dones = np.zeros(capacity, dtype=np.float32)

    def add(self, obs, action, reward, next_obs, done):
        self.obs[self.ptr] = obs
        self.actions[self.ptr] = action
```

```
        self.rewards[self.ptr] = reward
        self.next_obs[self.ptr] = next_obs
        self.dones[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)

    def sample(self, batch_size, device):
        idx = np.random.randint(0, self.size, size=batch_size)
        to_t = lambda x: torch.from_numpy(x).to(device)
        return {"obs": to_t(self.obs[idx]),
                "actions": to_t(self.actions[idx]),
                "rewards": to_t(self.rewards[idx]),
                "next_obs": to_t(self.next_obs[idx]),
                "dones": to_t(self.dones[idx])}
```

The buffer pre-allocates numpy arrays at initialization to avoid the overhead of growing a Python list. The `ptr` (pointer) tracks where the next transition goes, wrapping around via modular arithmetic so that old data is silently overwritten. When sampling, the buffer converts numpy arrays to PyTorch tensors on the target device.

> **Checkpoint.** Add 100 transitions with `obs_dim=10, act_dim=4` and sample a batch of 32. You should see: `obs` shape `(32, 10)`, `actions` shape `(32, 4)`, `rewards` shape `(32,)`, `buf.size == 100`, `buf.ptr == 100`. If shapes are wrong, check that the numpy arrays were initialized with the correct dimensions.
>
> **Note on the MLP helper.** The `TwinQNetwork` and `GaussianPolicy` classes use an `MLP` helper -- a standard feedforward network with ReLU activations. It lives in the lab file (`scripts/labs/sac_from_scratch.py`) alongside the snippet regions and takes three arguments: `input_dim`, `output_dim`, and `hidden_dims` (defaulting to `[256, 256]`). If you are copying code from the chapter, you will need this class. Run `--verify` to confirm everything assembles correctly.

4.4 Build It: Twin Q-network

SAC uses two Q-networks to reduce overestimation bias (as discussed in Section 4.2). Each Q-network takes a state-action pair and outputs a scalar Q-value. Having two independent networks and taking their minimum for the target counteracts the systematic upward bias of Q-learning:

$$Q_{\phi_1}(s, a), \quad Q_{\phi_2}(s, a)$$

The architecture is straightforward -- each Q-network is an MLP that concatenates state and action as input:

```
# Twin Q-networks for reducing overestimation bias
# (from scripts/labs/sac_from_scratch.py:twin_q_network)
```

```python
class TwinQNetwork(nn.Module):
    """Twin Q-networks for reducing overestimation."""

    def __init__(self, obs_dim: int, act_dim: int,
                 hidden_dims: list[int] = [256, 256]):
        super().__init__()
        self.q1 = MLP(obs_dim + act_dim, 1, hidden_dims)
        self.q2 = MLP(obs_dim + act_dim, 1, hidden_dims)

    def forward(self, obs, action):
        x = torch.cat([obs, action], dim=-1)
        return (self.q1(x).squeeze(-1),
                self.q2(x).squeeze(-1))
```

The two networks share no parameters -- they are fully independent MLPs with the same architecture (two hidden layers of 256 units, ReLU activations). The input dimension is `obs_dim + act_dim` because we concatenate state and action. The output is squeezed from `(batch, 1)` to `(batch,)` for convenient loss computation.

Why 256 hidden units? This is the SAC default from Haarnoja et al. (2018a), and it works well for low-dimensional state spaces like Fetch. PPO in Chapter 3 used 64 -- SAC's critics need more capacity because they approximate a function of both state AND action, which has a higher-dimensional input.

> **Checkpoint.** Create a `TwinQNetwork(obs_dim=10, act_dim=4)` and run a forward pass with batch size 32. You should see: Q1 shape `(32,)`, Q2 shape `(32,)`, both means near 0 at initialization (random weights produce centered outputs), and all values finite. If you get a shape error, check that `obs_dim + act_dim` matches the MLP input dimension.

4.5 Build It: Squashed Gaussian policy

SAC's policy outputs a continuous action distribution. The challenge: Fetch actions must be bounded in $[-1, 1]$, but a Gaussian distribution has unbounded support. The solution is a **squashed Gaussian** -- sample from a Gaussian, then apply tanh to bound the result:

$$z \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)^2), \qquad a = \tanh(z)$$

where μ_θ and σ_θ are neural network outputs (state-dependent mean and standard deviation).

The complication is in the log-probability. Because tanh is a nonlinear transformation, we need a Jacobian correction to get the correct density. The change-of-variables formula gives:

$$\log \pi(a \mid s) = \log p(z \mid s) - \sum_{i=1}^{d} \log(1 - \tanh^2(z_i))$$

where $p(z \mid s)$ is the Gaussian density and d is the action dimension. The correction term accounts for how tanh compresses the probability density near the boundaries ± 1. In code, we use a numerically stable formulation:

```
# Squashed Gaussian policy
# (from scripts/labs/sac_from_scratch.py:gaussian_policy)

class GaussianPolicy(nn.Module):
    LOG_STD_MIN, LOG_STD_MAX = -20, 2

    def __init__(self, obs_dim, act_dim,
                 hidden_dims=[256, 256]):
        super().__init__()
        self.backbone = MLP(obs_dim, hidden_dims[-1],
                            hidden_dims[:-1])
        self.mean_head = nn.Linear(hidden_dims[-1], act_dim)
        self.log_std_head = nn.Linear(hidden_dims[-1], act_dim)

    def forward(self, obs):
        features = F.relu(self.backbone.net[:-1](obs))
        features = self.backbone.net[-1](features)
        mean = self.mean_head(features)
        log_std = self.log_std_head(features)
        log_std = torch.clamp(log_std,
                              self.LOG_STD_MIN, self.LOG_STD_MAX)
        std = log_std.exp()
        # Reparameterization trick: sample = mean + std * noise
        dist = Normal(mean, std)
        x_t = dist.rsample()                # gradient flows through
        action = torch.tanh(x_t)            # squash to [-1, 1]
        # Log prob with numerically stable squashing correction
        log_prob = dist.log_prob(x_t).sum(dim=-1)
        log_prob -= (2 * (np.log(2) - x_t
                     - F.softplus(-2 * x_t))).sum(dim=-1)
        return action, log_prob
```

A few design choices in this code deserve attention. Unlike PPO in Chapter 3 (which used a state-independent `log_std` parameter), SAC learns a **state-dependent standard deviation**, so the policy can be more exploratory in unfamiliar states and more precise in well-understood ones. The `LOG_STD_MIN = -20` and `LOG_STD_MAX = 2` **clamps** prevent numerical instability, since without them `log_std` can go to $-\infty$ (zero variance, division by zero) or $+\infty$ (infinite variance, useless actions). We use `rsample()` instead of `sample()` -- the **reparameterization trick** -- so that gradients flow through the sampling operation, which is essential because without it we cannot

backpropagate through the actor loss. Finally, the **numerically stable log-prob** correction `2 * (log(2) - x_t - softplus(-2 * x_t))` is mathematically equivalent to `log(1 - tanh^2(x_t))` but avoids the issue where `tanh^2(x_t)` rounds to exactly 1.0 for large `x_t`, which would produce `log(0) = -inf`.

Math	Code	Meaning
$z \sim \mathcal{N}(\mu, \sigma^2)$	`x_t = dist.rsample()`	Sample from
$a = \tanh(z)$	`action = torch.tanh(x_t)`	Squash to $[-$
$\log p(z)$	`dist.log_prob(x_t).sum(-1)`	Gaussian log
$-\sum \log(1 - \tanh^2(z_i))$	`-(2 * (log(2) - x_t - softplus(-2*x_t))).sum(-1)`	Jacobian cor

> **Checkpoint.** Create a `GaussianPolicy(obs_dim=10, act_dim=4)` and run a forward pass with batch size 32. You should see: actions bounded in $[-1, 1]$, log-probs finite and negative (probabilities are less than 1, so log-probs are negative), shapes `(32, 4)` for actions and `(32,)` for log-probs. If actions exceed $[-1, 1]$, check that `torch.tanh` is applied. If log-probs contain NaN or $-\infty$, check the `LOG_STD_MIN` clamp.

4.6 Build It: Twin Q-network loss -- the soft Bellman backup

The critic update minimizes the squared difference between each Q-network's prediction and a target that incorporates the entropy bonus. This is the **soft Bellman backup**:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{B}} \left[\left(Q_{\phi_i}(s, a) - y \right)^2 \right]$$

where $\mathcal{B}$ is the replay buffer and the **Bellman target** y is:

$$y = r + \gamma(1 - d) \left[\min_{j=1,2} Q_{\bar{\phi}_j}(s', a') - \alpha \log \pi(a' \mid s') \right]$$

with $a' \sim \pi(\cdot \mid s')$ sampled from the current policy. The target has three parts: the immediate reward r, the discounted future value (estimated by the minimum of two target Q-networks), and the entropy bonus $-\alpha \log \pi$. The $(1 - d)$ term zeroes the bootstrap when the episode terminates.

```python
# Twin Q-network loss with soft Bellman backup
# (from scripts/labs/sac_from_scratch.py:twin_q_loss)

def compute_q_loss(q_network, target_q_network, policy,
                   batch, gamma=0.99, alpha=0.2):
    obs, actions = batch["obs"], batch["actions"]
    rewards, next_obs = batch["rewards"], batch["next_obs"]
    dones = batch["dones"]
```

```
    q1, q2 = q_network(obs, actions)  # current Q-values

    with torch.no_grad():  # no gradient through targets
        next_actions, next_log_probs = policy(next_obs)
        target_q1, target_q2 = target_q_network(next_obs, next_actions)
        target_q = torch.min(target_q1, target_q2)  # pessimistic
        # Soft Bellman backup
        target = rewards + gamma * (1.0 - dones) * (
            target_q - alpha * next_log_probs)

    q1_loss = F.mse_loss(q1, target)
    q2_loss = F.mse_loss(q2, target)

    info = {"q1_loss": q1_loss.item(),
            "q2_loss": q2_loss.item(),
            "q1_mean": q1.mean().item(),
            "q2_mean": q2.mean().item(),
            "target_q_mean": target.mean().item()}
    return q1_loss + q2_loss, info
```

The `torch.no_grad()` block is critical -- we do not backpropagate through the target computation. The target networks and the policy provide fixed targets for this update step. If gradients flowed through the target, the optimization would chase a moving target, defeating the purpose of having separate target networks.

Math	Code	Meaning
$Q_{\phi_i}(s, a)$	`q1, q2`	Current Q-value predictions
$\min_j Q_{\bar{\phi}_j}(s', a')$	`torch.min(target_q1, target_q2)`	Pessimistic target (reduces overestimati
$-\alpha \log \pi(a' \mid s')$	`-alpha * next_log_probs`	Entropy bonus in target
y	`target`	The Bellman target
$(1 - d)$	`(1.0 - dones)`	Zero bootstrap at episode end

> **Checkpoint.** Create a `TwinQNetwork`, copy it to make a `target_q_network`, create a `GaussianPolicy`, and compute the loss on a random batch. You should see: `q1_loss` finite and typically below 1.0 at initialization (random networks with zero-centered targets produce small errors), `q1_mean` near 0, `target_q_mean` near 0. If the loss is very large (above 100), check that rewards are bounded and that the done mask is applied correctly.

4.7 Build It: Actor loss with entropy

The policy update maximizes Q-values while maintaining entropy. The actor loss is:

$$L(\theta) = \mathbb{E}_{s \sim \mathcal{B},\, a \sim \pi_\theta} \left[\alpha \log \pi_\theta(a \mid s) - \min_{i=1,2} Q_{\phi_i}(s, a) \right]$$

Read this as: "find actions that have high Q-values ($-Q$ is minimized) while keeping the policy spread out ($\alpha \log \pi$ penalizes low entropy)." The balance between these two objectives is controlled by α.

```python
# Actor loss with entropy regularization
# (from scripts/labs/sac_from_scratch.py:actor_loss)

def compute_actor_loss(
    policy, q_network, obs, alpha=0.2,
):
    # Sample actions from current policy
    actions, log_probs = policy(obs)

    # Evaluate with Q-networks (use min for pessimism)
    q1, q2 = q_network(obs, actions)
    q_value = torch.min(q1, q2)

    # Loss = E[alpha * log_pi - Q]  (minimize)
    actor_loss = (alpha * log_probs - q_value).mean()

    info = {
        "actor_loss": actor_loss.item(),
        "entropy": -log_probs.mean().item(),  # H = -E[log pi]
        "log_prob_mean": log_probs.mean().item(),
    }
    return actor_loss, info
```

Notice that actions are sampled fresh from the current policy (not taken from the replay buffer). The Q-networks evaluate these new actions using their current weights. This is different from the critic update, which evaluated the actions that were actually taken when the transition was collected. The actor asks: "given this state, what action would I take NOW, and how good would it be?"

The entropy diagnostic `H = -E[log pi]` tracks how stochastic the policy is. At initialization, entropy is high (the policy has not learned any preferences). As training proceeds and α decreases, entropy drops and the policy becomes more deterministic. If entropy drops to near zero too early, that is worth investigating -- see What Can Go Wrong for diagnostics.

> **Checkpoint.** Compute the actor loss on random observations with `alpha=0.2`. You should see: `actor_loss` finite and positive at initialization (the policy has not learned to select high-Q actions yet), `entropy` positive (the random-initialized policy is stochastic), and `log_prob_mean` negative (probabilities are less than 1). If `actor_loss` is NaN, check that the squashed Gaussian log-prob computation in Section 4.5 is numerically stable.

4.8 Build It: Automatic temperature tuning

The actor loss in Section 4.7 uses a fixed $\alpha = 0.2$. That works for a single verification step, but in real training the right value is hard to choose ahead of time. Too low and exploration dies early. Too high and the policy wastes time on random actions. The right α depends on the task, the training stage, and the action dimensionality -- it changes as training progresses.

SAC solves this by learning α alongside the policy and critics. The idea (introduced in Section 4.1): define a target entropy $\bar{\mathcal{H}} = -\dim(\mathcal{A})$ -- for Fetch's 4D action space, $\bar{\mathcal{H}} = -4$ -- and adjust α to keep the policy's actual entropy near that target.

The temperature loss function is:

$$L(\alpha) = \mathbb{E}_{a \sim \pi}\left[-\alpha\left(\log \pi(a \mid s) + \bar{\mathcal{H}}\right)\right]$$

where $\log \pi(a \mid s)$ comes from the current policy's output (the same log-probabilities we computed in Section 4.5). The gradient pushes α in the right direction: if entropy is below the target, α increases to encourage more exploration; if entropy is above the target, α decreases to allow more exploitation.

In practice, we optimize $\log \alpha$ rather than α directly. Since $\exp(\cdot)$ is always positive, this ensures $\alpha > 0$ without needing a constrained optimizer.

```
# Automatic temperature tuning
# (from scripts/labs/sac_from_scratch.py:temperature_loss)

def compute_temperature_loss(
    log_alpha,          # learnable log(alpha)
    log_probs,          # log pi(a|s) from policy
    target_entropy,     # target entropy (-dim(A))
):
    alpha = log_alpha.exp()

    # If log_pi > -target (entropy too low): loss positive, alpha increases
    # If log_pi < -target (entropy too high): loss negative, alpha decreases
    alpha_loss = -(alpha * (log_probs.detach()
                            + target_entropy)).mean()

    info = {"alpha_loss": alpha_loss.item(),
            "alpha": alpha.item()}
    return alpha_loss, info
```

Notice the `.detach()` on `log_probs` -- we do not want the temperature gradient to flow back through the policy network. The temperature update adjusts only α, taking the current policy's entropy as a given. The policy has its own optimizer (Section 4.7) that handles its parameters.

Math	Code	Meaning
α	`log_alpha.exp()`	Temperature (always positive)
$\log \pi(a \mid s)$	`log_probs`	Policy log-probabilities (detached)
$\bar{\mathcal{H}} = -\dim(\mathcal{A})$	`target_entropy`	Target entropy (-4 for Fetch)

Checkpoint. Initialize `log_alpha = torch.tensor(0.0, requires_grad=True)` so that α starts at 1.0. After a few optimizer steps on random log-probs, α should have changed from 1.0 -- the direction depends on whether the random policy's entropy is above or below the target. The key check is that α remains positive (it always will, since we optimize $\log \alpha$). If `alpha_loss` is NaN, check that `log_probs` contains finite values.

Figure 4.2 shows what automatic temperature tuning looks like over a full training run. Early in training, when the policy is uncertain and entropy is naturally high, α is relatively large (around 0.47 at 30k steps). As the policy learns and becomes more deterministic, α drops -- reaching 0.0004 by 1M steps. The policy transitions from exploratory to exploitative without any manual schedule.

Entropy coefficient alpha over training steps, starting around 0.47 and decreasing to near 0.0004 over 1M steps, with annotations showing exploration phase (high alpha) and exploitation phase (low alpha)

Figure 4.2: Entropy coefficient α over training on FetchReachDense-v4. The automatic temperature tuning starts α around 0.47 (exploration phase) and drives it to 0.0004 (exploitation phase) as the policy converges. This happens without any manual schedule -- the dual gradient descent adjusts α to maintain the target entropy of $\bar{\mathcal{H}} = -4$. (Generated from TensorBoard logs in `runs/sac/FetchReachDense-v4/seed0/`, metric `replay/ent_coef`.)

4.9 Build It: SAC update -- wiring it together

We now have six components: replay buffer, twin Q-networks, squashed Gaussian policy, critic loss, actor loss, and temperature loss. The SAC update function wires them into a single gradient step. The sequence matters:

1. **Update Q-networks:** minimize Bellman error (uses current policy for next-action sampling)
2. **Update policy:** maximize Q-value plus entropy (uses updated Q-networks)
3. **Update temperature:** maintain target entropy (uses updated policy's log-probs)
4. **Soft-update target networks:** Polyak averaging $\bar{\phi} \leftarrow \tau\phi + (1-\tau)\bar{\phi}$

The first listing shows the three gradient updates -- Q-networks, policy, and temperature -- each using the loss functions we built in Sections 4.6-4.8:

```
# SAC update: gradient steps
# (from scripts/labs/sac_from_scratch.py:sac_update)

def sac_update(
```

```
    policy, q_network, target_q_network, log_alpha,
    policy_optimizer, q_optimizer, alpha_optimizer,
    batch, target_entropy, gamma=0.99, tau=0.005,
):
    alpha = log_alpha.exp().item()

    # 1. Q-network update
    q_loss, q_info = compute_q_loss(
        q_network, target_q_network, policy,
        batch, gamma, alpha)
    q_optimizer.zero_grad()
    q_loss.backward()
    q_optimizer.step()

    # 2. Policy update
    actor_loss, actor_info = compute_actor_loss(
        policy, q_network, batch["obs"], alpha)
    policy_optimizer.zero_grad()
    actor_loss.backward()
    policy_optimizer.step()

    # 3. Temperature update
    with torch.no_grad():
        _, log_probs = policy(batch["obs"])
    alpha_loss, alpha_info = compute_temperature_loss(
        log_alpha, log_probs, target_entropy)
    alpha_optimizer.zero_grad()
    alpha_loss.backward()
    alpha_optimizer.step()
```

Ordering matters. The Q-networks update first because the actor uses Q-values to evaluate its actions. The temperature updates last because it needs the updated policy's log-probs. Each component has its own Adam optimizer with learning rate 3e-4 -- coupling them into one optimizer would mix gradients that serve different objectives. We re-run the policy under `torch.no_grad()` to get fresh log-probs for the temperature update, because the policy parameters changed in step 2.

After the three gradient steps, the target networks blend in the new critic weights via Polyak averaging:

```
    # 4. Target network update (Polyak averaging)
    with torch.no_grad():
        for p, tp in zip(q_network.parameters(),
                         target_q_network.parameters()):
            tp.data.copy_(tau * p.data + (1 - tau) * tp.data)

    return {**q_info, **actor_info, **alpha_info}
```

This is a direct weighted blend of parameters, not a gradient step. It runs in `torch.no_grad()` and takes negligible time. The target network inherits 0.5% of the main network's weights per update ($\tau = 0.005$), providing the stable optimization targets that the critic loss (Section 4.6) depends on.

> **Checkpoint.** Initialize all networks, create three Adam optimizers, and run 20 updates on a random batch. You should see: `alpha` has changed from 1.0 (it may go up or down depending on the random policy's entropy relative to the target), `q1_loss` is finite, `actor_loss` is finite, and all values in the info dictionary are finite. If any value is NaN after 20 updates, check that each optimizer is attached to the correct parameters and that no gradients flow where they should not (target computation, temperature log-probs).

Quick verification: all seven components at once

Before running the full demo, verify that all seven components assemble and produce sane values:

```
bash docker/dev.sh python scripts/labs/sac_from_scratch.py --verify
```

Expected output:

```
============================================================
SAC From Scratch -- Verification
============================================================
Q-network:  Q1 mean ~0.04, Q2 mean ~0.06 (near zero at init)
Policy:     actions in [-0.99, 0.97], log_prob mean -2.63
SAC update: Q1 loss 2.35, actor loss -4.19, alpha 1.00 -> 0.99
            All values finite after 20 updates.
[PASS] All checks passed.
============================================================
```

This runs in under 30 seconds on CPU. If any value is NaN or if shapes are wrong, go back to the component that failed -- the error messages point to the specific check.

The demo: SAC solves Pendulum from scratch

The wired-up implementation can solve a real continuous control task. Run (~5 minutes on CPU, ~2 minutes on GPU):

```
bash docker/dev.sh python scripts/labs/sac_from_scratch.py --demo --steps 50000
```

Here are the results we got (your numbers may vary slightly with different seeds):

Step	Avg Return	Alpha	What's happening
5,000	-843	0.415	Learning started, still mostly random
10,000	-190	0.197	Solved (threshold: -200)
25,000	-141	0.048	Refining swing-up behavior
50,000	-134	0.017	Stable, nearly deterministic policy

The policy solves Pendulum by step 10,000 -- well within 50k. The temperature drops from 0.415 to 0.017, showing the automatic tuning at work: early exploration gives way to late exploitation without any manual schedule. Figure 4.3 shows the learning curve.

This is the same algorithm SB3 runs. The difference is pedagogical clarity, not algorithmic. Every gradient step you see in SB3's TensorBoard traces back to the functions you just built.

Learning curve from the SAC from-scratch demo on Pendulum-v1: average return on the y-axis improves from around -1200 to above -200 over 50k training steps on the x-axis, with a dashed line at -200 marking the solved threshold

Figure 4.3: Learning curve from the from-scratch SAC implementation on Pendulum-v1. The average return improves from random (~-1200) to solved (above -200) within 10k steps, then refines further to ~-134 by 50k steps. The dashed line marks the -200 solved threshold. This validates that all seven components -- replay buffer, twin Q-networks, squashed Gaussian policy, critic loss, actor loss, temperature tuning, and the wiring -- are correct. (Generated by `python scripts/labs/sac_from_scratch.py --demo --steps 50000`.)

4.10 Bridge: From-scratch to SB3

We have built SAC from scratch and verified each component. SB3 implements the same math in a production library. Before we use SB3 for the real training run, let's confirm that the two implementations agree on the core computation.

The key quantity to compare is the squashed Gaussian log-probability -- the mathematical object that appears in the actor loss (Section 4.7), the critic target (Section 4.6), and the temperature loss (Section 4.8). If the log-probs match, the losses match, and the gradients point in the same direction.

The bridging proof feeds the same random observations and pre-squash samples through our `GaussianPolicy.forward()` and through SB3's `SquashedDiagGaussianDistribution`. Both apply tanh squashing and compute log-probabilities, but with slightly different numerical formulas:

- **Our implementation:** `log_prob = dist.log_prob(x_t) - 2*(log(2) - x_t - softplus(-2*x_t))`
- **SB3's implementation:** `log_prob = dist.log_prob(x_t) - log(1 - tanh(x_t)^2 + 1e-6)`

The formulas are mathematically equivalent for the squashing correction. The ~0.02 nat typical difference comes from SB3's `1e-6` epsilon safety term, which prevents `log(0)` near the boundaries but introduces a small bias.

Run the comparison:

```
bash docker/dev.sh python scripts/labs/sac_from_scratch.py --compare-sb3
```

Expected output:

```
============================================================
```

```
SAC From Scratch -- SB3 Comparison
============================================================
Max abs log_prob diff: 2.055e-02
Tolerance (atol):      5.0e-02

[PASS] Our squashed Gaussian log_prob matches SB3
```

The match is within tolerance (0.02 < 0.05). For non-saturated actions -- the vast majority during training -- both formulas agree much more closely. The epsilon matters only when $\left|\tanh(u)\right| \to 1$, which happens at the action boundaries.

What SB3 adds beyond our from-scratch code

Our implementation handles one environment and stores transitions in a simple numpy buffer. SB3 adds engineering features that matter for real training:

- **Vectorized environments.** `n_envs=1` (SAC default in SB3) with `SubprocVecEnv` or `DummyVecEnv`. Unlike PPO which benefits heavily from parallel environments, SAC's off-policy nature means a single environment can fill the replay buffer adequately.
- **Efficient replay buffer.** SB3's replay buffer handles dictionary observations natively (the `observation`, `achieved_goal`, and `desired_goal` keys from Fetch), stores them efficiently in numpy arrays, and supports batched sampling.
- **`MultiInputPolicy`.** SB3 builds separate encoders for each key in the dictionary observation and concatenates them. Our from-scratch code assumed a flat observation vector.
- **Gradient clipping.** SB3 clips gradient norms by default, preventing a single bad batch from causing an explosively large update.
- **Automatic entropy coefficient** with constrained optimization -- the same math as our `compute_temperature_loss`, but with additional numerical safeguards.

Mapping SB3 TensorBoard metrics to our code

When you train with SB3 and open TensorBoard, the logged metrics correspond directly to the functions you built:

SB3 TensorBoard key	Our function	What it measures
`train/actor_loss`	`compute_actor_loss`	Policy loss (entropy-regularized)
`train/critic_loss`	`compute_q_loss`	Twin Q-network Bellman error
`replay/ent_coef`	`log_alpha.exp()`	Temperature α (auto-tuned)
`replay/q1_mean`	`TwinQNetwork` Q1 output	Average Q1 value in sampled batch
`replay/q2_mean`	`TwinQNetwork` Q2 output	Average Q2 value in sampled batch
`replay/q_min_mean`	`torch.min(q1, q2)`	Pessimistic Q-value estimate
`rollout/ep_rew_mean`	(environment)	Mean episode return
`rollout/success_rate`	(environment)	Fraction of episodes where goal is reached

We find this mapping useful for debugging. When you see `replay/ent_coef = 0.08` in TensorBoard, that is $\alpha = 0.08$ -- the output of the same `log_alpha.exp()` from

Section 4.8. When you see `train/critic_loss = 0.34`, that is the MSE Bellman error from `compute_q_loss` in Section 4.6. Having built each component yourself, you can trace any TensorBoard metric back to the exact computation that produced it.

4.11 Run It: Training SAC on FetchReachDense-v4

> **A note on script naming.** The production script is called `ch03_sac_dense_reach.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, SAC on dense Reach is chapter 3; in this book, it is chapter 4. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch03`. This is intentional -- renaming would break the tutorial infrastructure. When you see `ch03` in a command or file path, you are running the right script for this chapter.

```
------------------------------------------------------------
EXPERIMENT CARD: SAC on FetchReachDense-v4
------------------------------------------------------------
Algorithm:    SAC (soft actor-critic, off-policy, auto-tuned entropy)
Environment:  FetchReachDense-v4
Fast path:    500,000 steps, seed 0
Time:         ~14 min (GPU) / ~60 min (CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
    --seed 0 --total-steps 500000

Checkpoint track (skip training):
  checkpoints/sac_FetchReachDense-v4_seed0.zip

Expected artifacts:
  checkpoints/sac_FetchReachDense-v4_seed0.zip
  checkpoints/sac_FetchReachDense-v4_seed0.meta.json
  results/ch03_sac_fetchreachdense-v4_seed0_eval.json
  runs/sac/FetchReachDense-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.95
  mean_return > -5.0
  final_distance_mean < 0.03

Full multi-seed results: see REPRODUCE IT at end of chapter.
------------------------------------------------------------
```

Running the experiment

The one-command version:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all --seed 0
```

This runs training, evaluation, and comparison to PPO. It takes about 14 minutes on a GPU. For a quick sanity check that finishes in about 2 minutes:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py train --total-steps 1000
```

If you are using the checkpoint track (no training), the pretrained checkpoint is at `checkpoints/sac_FetchReachDense-v4_seed0.zip`. You can evaluate it directly:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py eval \
  --ckpt checkpoints/sac_FetchReachDense-v4_seed0.zip
```

Training milestones

Watch for these milestones during training. Note that `learning_starts=10000` means no training happens during the first 10k steps -- the agent collects random data to seed the replay buffer:

Timesteps	Success Rate	What's happening
0-10k	0%	Collecting random data (no training yet)
10k-50k	0-5%	Training begins, Q-values adjusting
50k-150k	20-60%	Rapid improvement, entropy decreasing
150k-400k	80-99%	Policy converging
400k-1M	100%	Fine-tuning, entropy near zero

Our test run achieved 100% success rate after approximately 300k steps, with an average goal distance of 18.6mm (the success threshold is 50mm) and throughput of approximately 594 steps per second on an NVIDIA GB10.

Reading TensorBoard

Launch TensorBoard to watch training in real time:

```
bash docker/dev.sh tensorboard --logdir runs --bind_all
```

Then open http://localhost:6006 in your browser. Here is what healthy training looks like:

Metric	Expected behavior	What to watch for
`rollout/ep_rew_mean`	Steadily increasing (less negative)	Should move from around -20
`rollout/success_rate`	0 -> 1 over training	The primary success metric
`replay/ent_coef`	Starts high (~0.47), decreases gradually	Should NOT drop to near-zero
`replay/q_min_mean`	Stabilizes in a reasonable range	Should NOT grow unbounded
`train/actor_loss`	Fluctuates, generally decreases	NaN means numerical instabil
`train/critic_loss`	Starts high, decreases and stabilizes	Tracks learning progress

These are the same quantities from the Build It sections: `replay/ent_coef` maps to `log_alpha.exp()`, `train/critic_loss` maps to `compute_q_loss`, and `replay/q_min_mean` maps to `torch.min(q1, q2)`. The TensorBoard metric mapping table in Section 4.10 has the full correspondence.

> **Tip.** Low GPU utilization (~5-10%) is expected. The bottleneck is CPU-bound MuJoCo simulation, not neural network operations. With small networks (two 256-unit hidden layers) and batch sizes (256), GPU operations complete in microseconds while the CPU runs physics. This is normal for state-based RL -- pixel-based policies in later chapters will use the GPU much more heavily.

SAC versus PPO: results comparison

Both algorithms solve FetchReachDense-v4, but with different tradeoffs. Figure 4.4 compares their learning curves.

Metric	PPO (Chapter 3)	SAC (this chapter)
Success Rate	100%	100%
Mean Return	-0.40	-1.06
Final Distance	4.6mm	18.6mm
Action Smoothness	1.40	1.68
Training Time	~6 min	~28 min
Throughput	~1,300 fps	~594 fps

Both achieve 100% success -- the off-policy stack is validated. SAC has a higher final distance (18.6mm vs 4.6mm), though still well within the 50mm threshold, and it is slower in wall-clock time because it updates more networks per step (actor + two critics + two targets versus PPO's single shared network). The slightly rougher actions (smoothness 1.68 vs 1.40) reflect the entropy bonus encouraging a spread of actions during training.

Both algorithms solve this task completely, so the interesting comparison is structural: PPO used each transition once and discarded it, while SAC stored every transition and reused it across many updates. On FetchReachDense, where signal is plentiful, this difference is a mild efficiency trade. On sparse-reward tasks (Chapter 5), the replay buffer becomes the key enabler for learning.

Learning curve comparison: PPO and SAC success rate over timesteps on FetchReachDense-v4, both reaching 100% but PPO reaching it somewhat earlier in wall-clock time while SAC has a different convergence profile

Figure 4.4: Learning curve comparison -- PPO (Chapter 3) versus SAC (this chapter) on FetchReachDense-v4. Both reach 100% success rate. PPO converges somewhat earlier measured by timesteps because its per-step updates are lighter. SAC converges to a stable solution with the replay buffer machinery that Chapter 5 requires. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/ppo/FetchReachDense-v4/seed0/` and `runs/sac/FetchReachDense-v4/seed0/`.)

Verifying results

After training completes, check the evaluation JSON:

```
cat results/ch03_sac_fetchreachdense-v4_seed0_eval.json | python -m json.tool | hea
```

Key fields to verify:

```
{
  "aggregate": {
    "success_rate": 1.0,
    "return_mean": -1.06,
    "final_distance_mean": 0.019
  }
}
```

The passing criteria are: success rate above 95%, mean return above -5.0, and final distance below 0.03 meters. Our runs consistently exceed these thresholds. If yours do not, see What Can Go Wrong below.

4.12 What can go wrong

Here are the failure modes we have encountered, organized by symptom. For each, we give the likely cause and a specific diagnostic.

`replay/q_min_mean` grows unbounded (above 100 and still rising)

Likely cause. Overestimation feedback loop: Q-targets use overestimated Q-values, which train the critic to overestimate further, which produces even higher targets. The twin Q-network minimum (Section 4.4) is supposed to prevent this, but it can still happen if target networks update too fast or rewards are misconfigured.

Diagnostic. Check three things: (1) rewards are bounded -- FetchReachDense should produce rewards in $[-1, 0]$; (2) target networks update with small τ (0.005, not 0.5 or 1.0) -- verify the Polyak averaging is actually running; (3) try reducing the learning rate from 3e-4 to 1e-4 to slow down the overestimation spiral.

`replay/ent_coef` drops to less than 0.01 within the first 10k steps

Likely cause. The policy collapsed to near-deterministic before learning anything useful. This can happen if the target entropy is too low or if the initial policy variance is too small.

Diagnostic. Check the target entropy setting -- it should be $-\dim(\mathcal{A}) = -4$ for Fetch. Try temporarily using a fixed entropy coefficient (`ent_coef=0.2` in SB3) to isolate whether the issue is auto-tuning or something else. If the fixed coefficient works, the auto-tuning's initial learning rate may be too aggressive -- try reducing the alpha learning rate to 1e-4.

Success rate stalls below 50% for more than 200k steps

Likely cause. Insufficient exploration or replay buffer sampling issues.

Diagnostic. Check the entropy coefficient -- if it is very low (below 0.01) early in training, the policy may have stopped exploring. Verify that `learning_starts` is not set too high (default 10,000 is appropriate). Check that `batch_size` is reasonable (256) -- too small means high-variance updates, too large means slow updates.

Training much slower than expected (below 200 fps on GPU)

Likely cause. GPU not in use, or excessive gradient steps per environment step.

Diagnostic. Check `nvidia-smi` inside the container -- a python process should be using GPU memory. Verify `gradient_steps=1` (the default, meaning one gradient step per environment step). Note that approximately 594 fps is typical for SAC on FetchReach with a GPU -- the bottleneck is CPU-bound MuJoCo simulation, not the neural networks.

Q1 and Q2 diverge significantly (difference greater than 10x)

Likely cause. One Q-network is stuck or has a different effective learning rate.

Diagnostic. Verify that both Q-networks share the same optimizer -- our `compute_q_loss` returns `q1_loss + q2_loss` and both should receive gradients through a single `q_optimizer.step()`. If you accidentally created separate optimizers for Q1 and Q2, check that both are being stepped.

`ep_rew_mean` flatlines near -20 for the entire run

Likely cause. Wrong environment or the policy is not receiving goal information.

Diagnostic. Check the env_id is `FetchReachDense-v4` (not the sparse variant `FetchReach-v4`, which produces rewards of exactly 0 or -1). Verify that SB3 is using `MultiInputPolicy` -- this handles the dictionary observation structure. Print `obs.keys()` to confirm you see `observation`, `achieved_goal`, and `desired_goal`.

`--compare-sb3` shows log-prob difference above 0.05

Likely cause. The squashing correction formula differs in a way that matters.

Diagnostic. Verify that your implementation uses the numerically stable formula: `2*(log(2) - x_t - softplus(-2*x_t))`. This is mathematically equivalent to `log(1 - tanh^2(x_t))` but avoids `log(0)` at the boundaries. If the difference is between 0.02 and 0.05, it is likely harmless -- the discrepancy comes from SB3's epsilon term and only affects near-saturated actions.

Build It `--verify` reports NaN in Q-loss or actor loss

Likely cause. Numerical instability in the squashed Gaussian log-probability computation.

Diagnostic. Check the `LOG_STD_MIN` and `LOG_STD_MAX` bounds in `GaussianPolicy`. Without them, the log standard deviation can go to $-\infty$ (producing zero variance and division by zero) or $+\infty$ (producing infinite variance and meaningless actions). The safe range is $[-20, 2]$. Also verify that the softplus-based squashing correction is used instead of the naive `log(1 - tanh^2(x))`, which produces $-\infty$ when `tanh(x)` rounds to exactly ± 1.

4.13 Summary

This chapter derived SAC from the maximum entropy objective and built it from the ground up. Here is what you accomplished:

- **The maximum entropy objective.** Instead of maximizing reward alone, SAC maximizes reward plus an entropy bonus: $J_{\text{MaxEnt}} = \mathbb{E}[\sum_t \gamma^t (r_t + \alpha \mathcal{H}(\pi))]$. This keeps the policy exploratory during training and lets it become deterministic as it converges. The optimal policy assigns probabilities proportional to exponentiated Q-values -- a Boltzmann distribution.
- **Off-policy learning.** SAC stores every transition in a replay buffer and reuses it across many updates, unlike PPO which discards data after each update. This makes SAC more sample-efficient and -- crucially -- enables the goal relabeling that Chapter 5 requires.
- **From-scratch implementation.** You built seven components: the replay buffer, twin Q-networks, squashed Gaussian policy, twin Q-network loss (soft Bellman backup), actor loss with entropy, automatic temperature tuning, and the full update step. Each was verified with concrete checks before wiring them together.
- **Bridge to SB3.** The bridging proof confirmed that our squashed Gaussian log-probability matches SB3's distribution implementation within 0.02 nats. SB3 adds engineering features (efficient replay storage, dictionary observation handling, gradient clipping) but computes the same underlying math.
- **Pipeline validation.** SAC achieved 100% success rate on FetchReachDense-v4, matching PPO's performance. The entire off-policy stack -- replay buffer, twin critics, automatic temperature tuning, Polyak averaging -- is validated and working.

That last point is the foundation for what comes next. Chapter 5 introduces Hindsight Experience Replay (HER), which tackles sparse rewards -- first on Reach (where the improvement is marginal) and then on Push (where it is transformative, lifting success from 5% to 99%). With sparse rewards, the agent receives a signal of either 0 (success) or -1 (failure) -- no distance information, no gradient toward the goal. Almost every episode is a failure, and failures carry no information about how to improve.

HER solves this by asking a simple question after each failed episode: ”I did not reach the desired goal, but what goal DID I reach?” It then relabels the transitions in the replay buffer with the achieved goal, turning a failure into a (synthetic) success. This

goal relabeling requires two things that you now have: (1) an off-policy algorithm that can learn from modified transitions, and (2) a replay buffer where those transitions are stored and accessible for relabeling. The off-policy machinery you built in this chapter is the foundation HER needs.

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
The results and pretrained checkpoints in this chapter
come from these runs:

  for seed in 0 1 2; do
    bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
      --seed $seed --total-steps 1000000
  done

Hardware:    Any machine with Docker (GPU optional; tested on NVIDIA GB10)
Time:         ~28 min per seed (Linux GPU), ~120 min per seed (Mac/CPU)
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/sac_FetchReachDense-v4_seed{0,1,2}.zip
  checkpoints/sac_FetchReachDense-v4_seed{0,1,2}.meta.json
  results/ch03_sac_fetchreachdense-v4_seed{0,1,2}_eval.json
  runs/sac/FetchReachDense-v4/seed{0,1,2}/

Results summary (what we got):
  success_rate: 1.00 +/- 0.00  (3 seeds x 100 episodes)
  return_mean:  -0.93 +/- 0.13
  final_distance_mean: 0.016 +/- 0.003

If your numbers differ by more than ~10%, check the
"What Can Go Wrong" section above.

The pretrained checkpoints are available in the book's
companion repository for readers using the checkpoint track.
------------------------------------------------------------
```

Exercises

1. (Verify) Reproduce the single-seed baseline.

Run the fast path command and verify your results match:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
  --seed 0 --total-steps 500000
```

Check the eval JSON: `success_rate` should be >= 0.95, `mean_return` > -5.0. Record your training time and steps per second for comparison with Chapter 3 (PPO). Expected: approximately 594 fps (versus PPO's approximately 1,300 fps), due to more network updates per step.

2. (Tweak) Twin Q-network ablation.

In the lab's `compute_q_loss()`, the target uses `torch.min(target_q1, target_q2)`. Change it to use only one Q-network: `target_q = target_q1`. Run `--verify` and observe: does `q1_mean` grow larger without the min clipping? Run `--demo --steps 50000` and compare final returns. Expected: without the min trick, Q-values may overestimate, leading to slightly worse or unstable training.

3. (Tweak) Fixed versus auto-tuned temperature.

In the lab's `sac_update()`, replace `alpha = log_alpha.exp().item()` with a fixed value: `alpha = 0.2`. Run `--demo --steps 50000` with fixed versus auto-tuned. Compare: (a) final return, (b) convergence speed, (c) policy entropy at the end. Expected: auto-tuned reaches better final performance because it reduces α as the policy improves; fixed $\alpha = 0.2$ may over-explore late in training.

4. (Extend) Add Q-value divergence monitoring.

In the verification code, add a check that tracks `|q1_mean - q2_mean|` over updates. Plot this divergence over the 20 verification steps. Expected: the two Q-networks should stay close (within approximately 0.5 of each other) because they see the same targets. Large divergence would indicate a problem -- for instance, if only one Q-network receives gradient updates.

5. (Challenge) SAC on Pendulum with different target entropies.

Modify the demo to try target entropy values of -0.5, -1.0 (default for 1D), and -2.0. For each, train for 50k steps and record: (a) final average return, (b) final alpha value, (c) steps to reach -200 return. Expected: lower target entropy leads to earlier exploitation (faster convergence but potentially less robust). Higher target entropy leads to more exploration (slower convergence but the policy may discover better strategies). This trade-off is the heart of the maximum entropy objective.

\newpage

Part 3 -- Sparse Goals, Real Progress

\newpage

5 HER on Sparse Reach and Push: Learning from Failure

This chapter covers:

- Why sparse rewards create a needle-in-a-haystack problem -- and why SAC alone achieves only 5% success on FetchPush-v4 when rewards are binary
- The HER insight: relabeling failed trajectories with goals that were actually achieved, turning every failure into a learning opportunity
- Implementing HER from scratch: goal sampling strategies (future, final, episode), transition relabeling with reward recomputation, and the full episode processing pipeline that amplifies data from 50 transitions to ~210
- Bridging from-scratch code to SB3's HerReplayBuffer: verifying the compute_reward invariant holds across both implementations
- Training SAC+HER on two sparse-reward tasks: FetchReach-v4 (marginal improvement, 96% -> 100%) and FetchPush-v4 (transformative improvement, 5% -> 99% success rate across 3 seeds)

In Chapter 4, you derived SAC from the maximum entropy objective, implemented it from scratch (replay buffer, twin Q-networks, squashed Gaussian policy, automatic temperature tuning), bridged to SB3, and achieved 100% success on FetchReachDense-v4. The entire off-policy stack is validated -- every transition goes into a replay buffer and gets reused across many gradient updates.

But dense rewards required us to hand-design a distance-based signal. The robot received continuous feedback proportional to how far it was from the goal: ”you're getting warmer” at every timestep. What if we only know whether the robot succeeded or failed? Sparse rewards -- $r = 0$ for success, $r = -1$ for failure -- are more natural because they do not require knowing the right distance metric, but they create a needle-in-a-haystack exploration problem. On FetchPush-v4 with sparse rewards, SAC alone achieves only 5% success: the puck almost never lands on the goal by chance, so the agent has no signal to learn from.

This chapter introduces Hindsight Experience Replay (HER), which transforms failures into learning signal by asking: ”what goal would this trajectory have achieved?” You will implement goal sampling, transition relabeling, and reward recomputation from scratch, verify each component, bridge to SB3's HerReplayBuffer, and watch success on FetchPush-v4 jump from 5% to 99%. HER uses the off-policy replay buffer from Chapter 4 -- relabeled transitions were not generated by the current policy, so only off-policy algorithms can use them.

One note on where this leads: HER solves the exploration problem for goal-conditioned tasks with known goal spaces. Chapter 6 applies the full SAC+HER stack to the hardest Fetch task -- PickAndPlace -- where the robot must lift an object off the table, requiring curriculum strategies and stress-testing to achieve reliability.

5.1 WHY: The sparse reward problem

Failure first: SAC without HER on FetchPush-v4

Before we introduce any new ideas, let's look at what happens when you train the SAC algorithm from Chapter 4 on a sparse-reward pushing task -- no HER, no tricks, just the off-policy stack you already built.

FetchPush-v4 is a step up from FetchReach. Instead of moving the gripper to a target, the robot must push a puck across a table to a goal position. The observation is 25-dimensional (gripper position, velocity, puck position, puck velocity, relative positions), the action is the same 4D Cartesian delta as before, and the reward is sparse: $r = 0$ if the puck is within 5cm of the goal, $r = -1$ otherwise.

Here are the results from training SAC without HER on FetchPush-v4 for 2 million steps (3 seeds, 100 evaluation episodes each):

Metric	SAC (no HER)
Success Rate	5.0% +/- 0.0%
Mean Return	-47.50 +/- 0.00
Final Distance	184.5mm +/- 0mm

A 5% success rate means the puck lands near the goal roughly once every 20 episodes -- essentially by accident. The mean return of -47.50 out of a maximum of 0 tells you the agent is failing at nearly every timestep of every episode. And the final distance of 184.5mm (the puck ends up about 18cm from the goal on average) shows the agent has not learned to push at all.

The three seeds agree almost perfectly, which tells us this is a structural limitation of SAC on sparse rewards, not a matter of unlucky initialization. Figure 5.1 makes this visible: a flat line at 5% for the entire training run.

SAC without HER on FetchPush-v4: success rate stuck at approximately 5 percent over 2M training steps, with three seeds overlaid showing consistent failure. A dashed line at 99 percent shows the HER target for contrast.

Figure 5.1: SAC without HER on FetchPush-v4. The success rate stays at approximately 5% across 2M training steps, consistent across 3 seeds. The dashed line at 99% shows what SAC+HER achieves. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/sac/FetchPush-v4/seed{0,1,2}/`.)

Why does standard RL fail here?

The problem is not that SAC is a bad algorithm. SAC achieved 100% success on FetchReachDense in Chapter 4. The problem is the reward structure.

With sparse rewards, the agent receives $r = -1$ for almost every transition, and the consequences cascade through the entire learning pipeline. Initial exploration is random -- the gripper moves chaotically while the puck sits on the table -- so most episodes fail completely, since the gripper rarely contacts the puck and pushing it to

exactly the right spot by chance is extremely unlikely. Because nearly every transition in the replay buffer carries reward -1, the critic learns that everything is equally bad, which means the Q-value function becomes flat with no state-action pair looking better than any other. Without reward variation, the policy has no direction to improve: the actor loss from Chapter 4 (Section 4.7) relies on Q-value differences to select actions, and when all Q-values are the same, the gradient points nowhere useful.

This is the needle-in-a-haystack problem. The sparse reward contains information -- it tells you whether you succeeded -- but it provides that information so rarely that the learning algorithm has nothing to work with.

Sparse rewards are the natural formulation

Despite this failure, sparse rewards are more honest than dense rewards. In Chapter 4, we used a dense reward proportional to distance:

$$R_{\text{dense}}(s, g) = -\|g_{\text{achieved}} - g_{\text{desired}}\|$$

This works, but it encodes a strong assumption: that the straight-line distance to the goal is the right measure of progress. For reaching tasks that assumption holds, and for pushing it partially holds (the puck should get closer to the goal), but for more complex tasks -- stacking, assembly, tool use -- designing the right distance metric is itself a hard problem, so you end up engineering the reward rather than solving the task.

Sparse rewards avoid this:

$$R_{\text{sparse}}(s, g) = \begin{cases} 0 & \text{if } \|g_{\text{achieved}} - g_{\text{desired}}\| < \epsilon \\ -1 & \text{otherwise} \end{cases}$$

where $\epsilon = 0.05$ meters (5cm) is the success threshold. The sparse reward asks a binary question -- ”did you succeed?” -- without assuming anything about how to measure progress, which means that if we can learn from sparse rewards, we have a more general solution.

The challenge is making that learning possible.

Why Push is harder than Reach

You might wonder: does SAC without HER also fail on sparse Reach? Not quite. Here are the Reach results:

Metric	SAC (no HER) on FetchReach-v4
Success Rate	96.0% +/- 7.0%

Reach achieves 96% without HER because random exploration occasionally succeeds. The gripper workspace is roughly 15cm across and the success threshold is 5cm, so

random flailing in a 15cm cube has a reasonable chance of landing within 5cm of a random target -- roughly every few episodes, the gripper stumbles into the goal by accident. Those accidental successes provide enough reward signal for the critic to learn: ”being near the goal is good.”

Push is different. Success requires a coordinated sequence: approach the puck, make contact, push it in the right direction, and stop it at the goal. Even a generous estimate of the probability of achieving this by random exploration is vanishingly small. We call this the **effective horizon** -- the number of coordinated steps needed for success. For Reach, the effective horizon is roughly 2-5 steps (just move toward the target). For Push, it is 20-50 steps (approach + contact + push + stop). Random exploration at each step has perhaps a 10-20% chance of being correct, so the probability of 20 consecutive correct steps is $(0.15)^{20} \approx 10^{-16}$.

In 2 million training steps with 8 parallel environments, the agent experiences roughly 40,000 episodes. Even at 40,000 episodes, the chance of a single random success on Push is negligible. The agent needs a way to extract learning signal from all these failed episodes -- and that is exactly what HER provides.

5.2 HOW: Hindsight Experience Replay

The insight

Here is the idea that makes HER work, stated plainly:

> A trajectory that fails to reach goal g is a successful demonstration of how to reach wherever it ended up.

If the robot tried to push the puck to position $(0.3, 0.2)$ but the puck ended at $(0.5, 0.4)$, we have evidence that the executed actions push the puck to $(0.5, 0.4)$. We can **relabel** the trajectory: pretend the goal was $(0.5, 0.4)$ all along, and now we have a successful episode with reward $r = 0$.

This is Hindsight Experience Replay (Andrychowicz et al., 2017). The name captures the idea: in hindsight, we reinterpret what the agent was trying to do. We are not changing the physics or the actions -- we are changing the question. Instead of ”did you reach the intended goal?”, we ask ”what goal did you reach?”

The insight is simple, but it changes everything about how the agent learns. Every failed episode becomes a source of learning signal. The agent does not need to succeed by chance -- it learns from its failures by reinterpreting them as successes for different goals.

Formal definition

> **Definition (Hindsight Experience Replay).**
>
> *Motivating problem.* With sparse rewards, the replay buffer contains almost exclusively failed transitions ($r = -1$). The critic cannot distinguish promising actions from useless ones, because all actions look equally bad.

Intuitive description. After each episode, we add the original transitions to the replay buffer (with the real goal), and also add relabeled copies where the goal is replaced with a goal that was actually achieved during the episode. The reward is recomputed for the new goal. Since the achieved goal is exactly where the agent ended up, many of these relabeled transitions are successes ($r = 0$).

Formal definition. Given an episode of transitions $\{(s_t, a_t, r_t, s_{t+1}, g_d)\}_{t=0}^{T-1}$ with desired goal g_d and achieved goals $\{g_a^t\}_{t=0}^{T-1}$, HER adds to the replay buffer:

1. The original transitions (with goal g_d and reward r_t)
2. For each transition t, k relabeled copies where g_d is replaced by $g' \sim S(t)$ (a goal sampled according to a strategy S), and the reward is recomputed: $r'_t = R(g_a^t, g')$

The reward recomputation is critical. We do not assume the relabeled transition is a success -- we **recompute** the reward using the environment's `compute_reward` function. If $\|g_a^t - g'\| < \epsilon$, then $r'_t = 0$ (success); otherwise $r'_t = -1$ (failure).

Grounding example. A 50-step failed episode where the puck ends up 18cm from the goal. Every original transition has $r = -1$. With $k = 4$ relabeled copies per transition and a relabeling ratio of 0.8, we generate $50 + 50 \times 0.8 \times 4 = 210$ total transitions. Because the relabeled goals come from the trajectory itself (where the puck actually went), many relabeled transitions satisfy $\|g_a - g'\| < 0.05$, producing $r' = 0$. The success fraction in the augmented data jumps from ~0% to ~4-80%, depending on how closely the trajectory's achieved goals cluster -- random trajectories produce ~4-8%, while purposeful trajectories from a partially trained policy produce 60-80%.

Non-example. HER does not invent data. The observations, actions, and achieved goals are real -- they came from actual environment interaction. Only the desired goal and the reward are modified. HER also does not guarantee that relabeled transitions are successes: if the achieved goal at timestep t is far from the sampled goal g', the relabeled reward is still -1.

Goal sampling strategies

The sampling strategy $S(t)$ determines which achieved goals are used for relabeling. Andrychowicz et al. (2017, Section 3.3) proposed three strategies:

Strategy	Description	Formal
FUTURE	Sample from achieved goals at timesteps after t	$g' \sim \text{Uniform}(\{g_a^{t'} : t' > t\})$
FINAL	Always use the last achieved goal	$g' = g_a^{T-1}$
EPISODE	Sample from any achieved goal in the episode	$g' \sim \text{Uniform}(\{g_a^{t'} : t' \in [0, T-1]\})$

FUTURE is the most common choice and works best in practice, because goals from

future timesteps are ”reachable” from the current state -- the agent demonstrated that it could get there from here. Goals from earlier timesteps are less useful since they represent states the agent has already passed through.

FINAL is simpler but less diverse, since every relabeled transition uses the same goal. The agent therefore only learns about reaching the endpoint of each trajectory, not the intermediate states along the way.

EPISODE is the most diverse but includes goals from before the current timestep. These backward goals are reachable in principle (the agent was there earlier), yet the actions taken to reach them point in the reverse direction, which can confuse the critic.

The parameter k: how many goals to sample

For each original transition, HER creates k relabeled copies. The default from Andrychowicz et al. (2017) is $k = 4$, which means each transition appears in the buffer once with the real goal and up to four times with relabeled goals.

Why $k = 4$ and not $k = 1$ or $k = 100$? With $k = 1$, you double the data but the success fraction stays relatively low. With $k = 100$, you generate 100x more data, but it is all from the same underlying trajectory -- the marginal value of each additional relabeled copy decreases. $k = 4$ provides a good balance between data amplification and diversity. Our 120-run hyperparameter sweep (detailed in the Reproduce It section) confirmed that $k = 4$ performs within noise of $k = 8$, consistent with the original paper's recommendation.

Data amplification: the quantitative effect

The arithmetic of HER reveals why it is so powerful. Consider a 50-step episode with $k = 4$ and `her_ratio = 0.8` (the probability that a given transition receives relabeled copies):

- **Original transitions:** $T = 50$
- **Relabeled transitions:** $T \times$ her_ratio $\times\, k = 50 \times 0.8 \times 4 = 160$
- **Total:** $50 + 160 = 210$

The **data amplification ratio** is $210/50 = 4.2$x. From a single episode, we extract over four times as many training transitions.

More importantly, the **success fraction** changes dramatically. In the original 50 transitions, all have $r = -1$ (the episode failed). In the 160 relabeled transitions, a significant fraction have $r = 0$ because the relabeled goal was achieved. The exact fraction depends on how closely the trajectory's achieved goals cluster -- with random trajectories it might be ~4-8%, with a partially trained policy it can reach 60-80%.

This is the core quantitative insight of HER: it manufactures dense reward signal from sparse feedback, not by changing the reward function, but by reinterpreting the data.

Why HER requires off-policy learning

HER only works with off-policy algorithms like SAC and TD3. The reason is structural, and understanding it helps clarify what HER actually does to the data. Here are the three conditions:

Definition (Off-policy requirement for HER).

HER requires off-policy learning because:

1. **Relabeled transitions were not generated by the current policy.** The policy that collected the data was trying to reach goal g_d, not goal g'. The actions were chosen based on observations conditioned on g_d. Using these transitions to learn about g' is inherently off-policy.
2. **The reward is retroactively changed.** The agent experienced $r = -1$, but HER stores $r' = 0$. An on-policy method like PPO computes policy gradients using the actual rewards the agent received. Substituting a different reward invalidates the gradient estimate.
3. **The goal is modified after collection.** On-policy methods require that the data distribution matches the current policy. Changing the goal changes the effective data distribution -- the transition now represents behavior under a different goal-conditioned policy than the one being optimized.

This is why the curriculum builds SAC (Chapter 4) before HER (this chapter). SAC's replay buffer stores transitions and reuses them regardless of which policy generated them. HER exploits this by modifying what the stored transitions mean. PPO cannot do this -- it must use each transition exactly as collected, then discard it. Figure 5.2 illustrates how relabeling transforms a single failed trajectory.

HER relabeling diagram showing a 5-step trajectory as a horizontal sequence of states. The original desired goal (red X) is far away. Achieved goals (blue dots) are along the path. Arrows show how HER substitutes the desired goal with an achieved goal from a future timestep, turning a failure into a success.

Figure 5.2: HER relabeling in action. A 5-step trajectory (blue dots showing achieved goals at each timestep) fails to reach the original desired goal (red X). HER relabels the transition at timestep 2 with the achieved goal at timestep 4. Because the puck was at that position at timestep 4, the relabeled transition becomes a success ($r = 0$). The three strategies (future, final, episode) differ in which timesteps are eligible for goal sampling. (Illustrative diagram.)

5.3 Build It: Data structures

Before we implement relabeling, we need to define what a goal-conditioned transition looks like. Recall from Chapter 2 that Fetch environments produce dictionary observations with three keys: `observation` (robot state), `achieved_goal` (where the agent is or what it accomplished), and `desired_goal` (the target). HER operates on both goal fields -- it reads `achieved_goal` to know what happened and modifies `desired_goal` to change the question.

Each transition carries seven fields. This is the standard five-tuple from Chapter 4's replay buffer -- $(s, a, r, s', \text{done})$ -- extended with the two goal arrays:

```
# (from scripts/labs/her_relabeler.py:data_structures)

class Transition(NamedTuple):
    """A single environment transition in a goal-conditioned setting."""
    obs: np.ndarray           # observation (e.g., robot state)
    action: np.ndarray        # action taken
    reward: float             # reward received
    next_obs: np.ndarray      # next observation
    done: bool                # episode terminated?
    achieved_goal: np.ndarray # goal actually achieved at next_obs
    desired_goal: np.ndarray  # goal the agent was trying to reach


class Episode(NamedTuple):
    """A complete episode of transitions."""
    transitions: list[Transition]

    def __len__(self) -> int:
        return len(self.transitions)


class GoalStrategy(Enum):
    """HER goal sampling strategies (Andrychowicz et al., 2017)."""
    FINAL = "final"      # Use final achieved goal
    FUTURE = "future"    # Sample from future timesteps
    EPISODE = "episode"  # Sample from anywhere in episode
```

The `Transition` extends Chapter 4's five-tuple with `achieved_goal` and `desired_goal`. An `Episode` is a list of transitions from a single environment rollout. The `GoalStrategy` enum encodes the three sampling strategies we discussed in Section 5.2. (The source file also includes a `RANDOM` strategy that samples from the full replay buffer -- we omit it here because it requires buffer access and is rarely used in practice.)

Checkpoint. Construct a transition manually and verify all seven fields are accessible:

```
import numpy as np
t = Transition(
    obs=np.zeros(10), action=np.zeros(4), reward=-1.0,
    next_obs=np.zeros(10), done=False,
    achieved_goal=np.array([0.5, 0.4, 0.15]),
    desired_goal=np.array([0.3, 0.2, 0.10]),
)
assert t.reward == -1.0
assert t.achieved_goal.shape == (3,)
```

```
assert GoalStrategy.FUTURE.value == "future"
```

All seven fields should be present. `GoalStrategy.FUTURE.value` should return `"future"`.

5.4 Build It: Goal sampling

HER needs to choose which achieved goals to use as relabeled targets. The three strategies from Section 5.2 are formalized in `sample_her_goals`:

- **FUTURE:** Sample g' from $\{g_a^{t'} : t' > t\}$ -- achieved goals from future timesteps
- **FINAL:** Always use $g' = g_a^{T-1}$ -- the episode's last achieved goal
- **EPISODE:** Sample g' from $\{g_a^{t'} : t' \in [0, T-1]\}$ -- any achieved goal

```
# (from scripts/labs/her_relabeler.py:goal_sampling)

def sample_her_goals(
    episode: Episode,
    transition_idx: int,
    strategy: GoalStrategy = GoalStrategy.FUTURE,
    k: int = 4,
) -> list[np.ndarray]:
    """Sample alternative goals for HER relabeling."""
    n = len(episode)
    goals = []
    if strategy == GoalStrategy.FINAL:
        final_goal = episode.transitions[-1].achieved_goal
        goals = [final_goal.copy() for _ in range(k)]

    elif strategy == GoalStrategy.FUTURE:
        future_indices = list(range(transition_idx + 1, n))
        if len(future_indices) == 0:
            goals = [episode.transitions[-1].achieved_goal.copy()
                     for _ in range(k)]
        else:
            sampled_indices = np.random.choice(
                future_indices,
                size=min(k, len(future_indices)),
                replace=False,
            ).tolist()
            while len(sampled_indices) < k:
                sampled_indices.append(n - 1)
            goals = [episode.transitions[i].achieved_goal.copy()
                     for i in sampled_indices]
```

The EPISODE strategy follows the same pattern but samples from the full episode instead of only future timesteps:

```
    # (continued from sample_her_goals)

    elif strategy == GoalStrategy.EPISODE:
        all_indices = list(range(n))
        sampled_indices = np.random.choice(
            all_indices, size=k, replace=True
        ).tolist()
        goals = [episode.transitions[i].achieved_goal.copy()
                 for i in sampled_indices]

    return goals
```

The FUTURE strategy is the most nuanced. For transition at index t, it samples from indices $[t+1, T-1]$ without replacement (up to k samples). If fewer than k future indices exist (near the end of the episode), it pads with the final achieved goal. This ensures we always return exactly k goals.

FINAL is the simplest -- it repeats the same goal k times. All relabeled transitions from this strategy share the same target, which means less diversity but a strong signal about the trajectory's endpoint.

EPISODE samples with replacement from the full episode, providing the most diversity but including backward goals that may be less useful.

Checkpoint. Create a 20-step synthetic episode and verify each strategy:

```
episode = create_synthetic_episode(n_steps=20)

# FUTURE at idx=5: goals from indices [6, 19]
goals = sample_her_goals(episode, transition_idx=5,
                         strategy=GoalStrategy.FUTURE, k=4)
assert len(goals) == 4
assert all(g.shape == (3,) for g in goals)

# FINAL: all goals identical (last achieved_goal)
goals = sample_her_goals(episode, transition_idx=5,
                         strategy=GoalStrategy.FINAL, k=4)
assert all(np.array_equal(g, goals[0]) for g in goals)
```

FUTURE should return 4 goals each with shape `(3,)`. FINAL should return 4 identical goals.

5.5 Build It: Transition relabeling and reward recomputation

The core HER operation substitutes a new goal and recomputes the reward. Given a transition $(s_t, a_t, r_t, s_{t+1}, g_d)$ and a new goal g', the relabeled reward is:

$$r'_t = R(g^t_a, g') = \begin{cases} 0 & \text{if } \|g^t_a - g'\| < \epsilon \\ -1 & \text{otherwise} \end{cases}$$

where g_a^t is the achieved goal (unchanged) and $\epsilon = 0.05$ meters. The observation and action stay the same -- only the goal and reward change.

The **critical invariant** from Chapter 2 applies here: the reward must be recomputed, not assumed. We do not set $r' = 0$ because "the goal was achieved" -- we call the environment's reward function and let it compute the answer. This matters because the achieved goal might be close to but not within the threshold of the new goal, in which case $r' = -1$ even after relabeling.

```
# (from scripts/labs/her_relabeler.py:relabel_transition)

def relabel_transition(
    transition: Transition,
    new_goal: np.ndarray,
    compute_reward_fn: callable,
    threshold: float = 0.05,
) -> Transition:
    """Relabel a transition with a new goal and recompute reward."""
    new_reward = compute_reward_fn(
        transition.achieved_goal,
        new_goal,
        {"distance_threshold": threshold},
    )
    return Transition(
        obs=transition.obs,
        action=transition.action,
        reward=new_reward,
        next_obs=transition.next_obs,
        done=transition.done,
        achieved_goal=transition.achieved_goal,  # unchanged
        desired_goal=new_goal,  # replaced
    )
```

The `sparse_reward` function mirrors the Gymnasium-Robotics `compute_reward` API -- it takes achieved and desired goals plus an info dict, and returns the binary reward:

```
# (from scripts/labs/her_relabeler.py:relabel_transition)

def sparse_reward(
    achieved_goal: np.ndarray,
    desired_goal: np.ndarray,
    info: dict,
) -> float:
    """Sparse reward: 0 if within threshold, -1 otherwise."""
    threshold = info.get("distance_threshold", 0.05)
    distance = np.linalg.norm(achieved_goal - desired_goal)
    return 0.0 if distance < threshold else -1.0
```

Notice that `achieved_goal` stays the same in the relabeled transition -- only `desired_goal` changes. HER makes the mostly-negative sparse reward learnable by

turning failures into a mix of successes and failures through relabeling.

Checkpoint. Relabeling with the transition's own achieved goal should always produce a success (reward = 0), because the distance is zero:

```
episode = create_synthetic_episode(n_steps=10)
transition = episode.transitions[5]

# Relabel with own achieved goal -> guaranteed success
relabeled = relabel_transition(
    transition, transition.achieved_goal, sparse_reward
)
assert transition.reward == -1.0   # original: failure
assert relabeled.reward == 0.0     # relabeled: success
assert np.array_equal(relabeled.achieved_goal,
                      transition.achieved_goal)  # unchanged

# Relabel with a distant goal -> still failure
far_goal = transition.achieved_goal + np.array([1.0, 1.0, 1.0])
relabeled_far = relabel_transition(
    transition, far_goal, sparse_reward
)
assert relabeled_far.reward == -1.0  # distance >> 0.05m
```

If `achieved_goal = [0.50, 0.40, 0.15]` and we relabel with `new_goal = [0.50, 0.40, 0.15]`, distance = 0.000m < 0.05m, so reward = 0 (success). If we relabel with `new_goal = [1.50, 1.40, 1.15]`, distance = 1.73m > 0.05m, so reward = -1 (failure).

5.6 Build It: Episode processing and wiring

The three components -- data structures, goal sampling, and transition relabeling -- wire together into a single function that processes an entire episode. For each transition, `process_episode_with_her` adds the original, then (with probability `her_ratio`) samples k alternative goals and creates relabeled copies:

```
# (from scripts/labs/her_relabeler.py:her_buffer_insert)

def process_episode_with_her(
    episode: Episode,
    compute_reward_fn: callable,
    strategy: GoalStrategy = GoalStrategy.FUTURE,
    k: int = 4,
    her_ratio: float = 0.8,
) -> list[Transition]:
    """Process an episode: HER-augmented transitions."""
    all_transitions = []

    for idx, transition in enumerate(episode.transitions):
```

```
        all_transitions.append(transition)  # always keep original

        if np.random.random() < her_ratio:  # probabilistic relabeling
            her_goals = sample_her_goals(episode, idx, strategy, k)
            for new_goal in her_goals:
                relabeled = relabel_transition(
                    transition, new_goal, compute_reward_fn
                )
                all_transitions.append(relabeled)

    return all_transitions
```

A small helper tells us how many transitions in the augmented batch are successes:

```
# (from scripts/labs/her_relabeler.py:her_buffer_insert)

def compute_success_fraction(
    transitions: list[Transition],
) -> float:
    """Fraction of transitions with non-negative reward."""
    if not transitions:
        return 0.0
    successes = sum(1 for t in transitions if t.reward >= 0)
    return successes / len(transitions)
```

The processing function iterates through each transition, always keeping the original, and probabilistically adding k relabeled copies. The success fraction is the metric that should jump from ~0% to a meaningful fraction after HER processing.

Checkpoint. Process a 50-step synthetic episode and verify the data amplification arithmetic:

```
episode = create_synthetic_episode(n_steps=50)

# Original: ~0% success (random trajectory, sparse rewards)
original_success = compute_success_fraction(episode.transitions)

# With HER: ~210 transitions, success fraction > 0%
her_transitions = process_episode_with_her(
    episode, sparse_reward,
    strategy=GoalStrategy.FUTURE, k=4, her_ratio=0.8
)
her_success = compute_success_fraction(her_transitions)

n_original = len(episode)           # 50
n_total = len(her_transitions)      # ~210
amplification = n_total / n_original  # ~4.2x

assert n_total > n_original
```

```
assert her_success > original_success
```

You should see approximately 210 total transitions (50 original + ~160 relabeled) with a success fraction meaningfully above 0%. The exact success fraction depends on the random trajectory -- with random actions, expect ~4-8%. With a partially trained policy that moves more purposefully, this would be much higher (60-80%). HER amplifies competence; it does not create it from nothing.

Verification: the full lab

Run the from-scratch implementation's sanity checks end-to-end:

```
bash docker/dev.sh python scripts/labs/her_relabeler.py --verify
```

Expected output:

```
============================================================
HER Relabeler -- Verification
============================================================
Verifying goal sampling...
  final: sampled 4 goals
  future: sampled 4 goals
  episode: sampled 4 goals
  [PASS] Goal sampling OK

Verifying relabeling...
  Original reward: -1.0
  Relabeled reward: 0.0
  [PASS] Relabeling OK

Verifying HER processing...
  Original success rate: 0.0%
  HER success rate: ~4%
  Transitions: 50 -> ~218
  [PASS] HER processing OK

============================================================
[ALL PASS] HER implementation verified
============================================================
```

The demo: seeing amplification in action

Run the demo to see how different strategies compare:

```
bash docker/dev.sh python scripts/labs/her_relabeler.py --demo
```

This processes a 50-step synthetic episode with each of the three strategies and reports the transition count and success fraction for each. FUTURE typically produces the

highest success fraction because it provides diverse goals that are naturally close to the achieved states along the trajectory. Figure 5.3 visualizes the amplification effect.

Data amplification visualization: bar chart showing original episode with 50 transitions and approximately 0 percent success compared to HER-processed episode with approximately 210 transitions and a meaningful fraction of successes, annotated with k=4 and her_ratio=0.8.

Figure 5.3: Data amplification effect of HER. A 50-step episode with all failures (left) is processed with $k = 4$ and `her_ratio = 0.8`, producing approximately 210 transitions with a significant fraction of successes (right). The exact success fraction depends on trajectory coherence -- random trajectories produce ~4-8%, while purposeful trajectories from a partially trained policy produce 60-80%. (Generated by `python scripts/labs/her_relabeler.py --demo`.)

This lab is not how we train policies -- SB3's HER wrapper handles that in the Run It section. The lab shows *what* relabeling does to your data, with every operation visible and verifiable. The goal sampling strategies from Section 5.4, the reward recomputation from Section 5.5, and the episode processing from this section are the same math that SB3's `HerReplayBuffer` computes internally. In the next span, we verify that claim directly by comparing our from-scratch output against SB3.

5.7 Bridge: From-scratch to SB3

We claimed that the goal sampling, transition relabeling, and reward recomputation you built in Sections 5.3-5.6 are the same operations that SB3's `HerReplayBuffer` performs internally. Let's verify that directly.

The bridging proof feeds the same synthetic episode through our `process_episode_with_her()` and through SB3's `HerReplayBuffer`, then checks the key invariant: **sampled rewards must equal `compute_reward(achieved_goal, desired_goal)`**. Because sparse rewards are binary (0 or -1), this comparison requires exact equality -- no floating-point tolerance needed.

Run the comparison:

```
bash docker/dev.sh python scripts/labs/her_relabeler.py --compare-sb3
```

Expected output:

```
HER Relabeler -- SB3 Comparison
Max abs reward diff: 0.000e+00
Success fraction:    0.799
n_sampled_goal:      4

[PASS] SB3 HER relabeling is reward-consistent (compute_reward invariant)
```

The reward difference is exactly zero -- every relabeled reward in SB3's buffer matches what `compute_reward(achieved_goal, desired_goal)` returns. The success fraction of 0.799 confirms that HER is creating successes from failures, and `n_sampled_goal=4` confirms both implementations use $k = 4$.

Mapping SB3 parameters to our concepts

SB3's `HerReplayBuffer` constructor takes the same conceptual parameters we built, under slightly different names:

SB3 parameter	Our concept	Section
`n_sampled_goal=4`	$k = 4$ goals per transition	5.2
`goal_selection_strategy="future"`	`GoalStrategy.FUTURE`	5.4
`online_sampling=True`	Relabel at sample time	5.6

The last parameter -- `online_sampling` -- reveals an implementation difference worth understanding. Our from-scratch code relabels at **insertion time**: when an episode ends, `process_episode_with_her()` generates all relabeled copies and stores them immediately. SB3 relabels at **sample time**: it stores original episodes as-is, then performs relabeling lazily when sampling a batch for training. The math is identical; the timing differs.

Online sampling is more memory-efficient -- SB3 stores each episode once and generates relabeled variants on the fly, avoiding the 4.2x storage overhead. For teaching, insertion-time relabeling makes the data amplification visible and verifiable. For production training, SB3's approach scales better.

What SB3 adds beyond our from-scratch code

Our lab processes one episode at a time with explicit Python loops. SB3 adds several engineering features that matter for real training. It provides vectorized episode storage across 8 parallel environments with automatic episode boundary detection, so that it knows where each episode starts and ends within the buffer. This storage integrates directly with SAC's training loop -- no manual episode collection is needed, since the replay buffer, goal relabeling, and gradient updates run as a unified pipeline. SB3 also stores `observation`, `achieved_goal`, and `desired_goal` arrays separately rather than bundling them into named tuples, which is more memory-efficient. And as noted above, online relabeling at sample time avoids the 4.2x storage overhead of materializing all relabeled copies up front.

The from-scratch code exists so you understand *what* relabeling does. SB3 handles the *how fast* and *at what scale*. They compute the same function; the bridging proof confirms it.

5.8 Run It: SAC+HER on sparse Reach and Push

> **A note on script naming.** The production script is called `ch04_her_sparse_reach_push.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, HER on sparse Reach/Push is chapter 4; in this book, it is chapter 5. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch04`. When you see `ch04` in a command or file path, you are running the right script for this chapter.

```
------------------------------------------------------------
EXPERIMENT CARD: SAC + HER on FetchPush-v4 (sparse)
------------------------------------------------------------
Algorithm:   SAC + HER (future strategy, n_sampled_goal=4, ent_coef=0.05, gamma=0.95)
Environments: FetchReach-v4 (sparse, validation), FetchPush-v4 (sparse, primary)
Fast path:    FetchPush-v4, 500,000 steps, seed 0
Time:         ~14 min (GPU) / ~60 min (CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
    --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05

Checkpoint track (skip training):
  checkpoints/sac_her_FetchPush-v4_seed0.zip

Expected artifacts:
  checkpoints/sac_her_FetchPush-v4_seed0.zip
  checkpoints/sac_her_FetchPush-v4_seed0.meta.json
  results/ch04_sac_her_fetchpush-v4_seed0_eval.json
  runs/sac_her/FetchPush-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.90
  mean_return > -20.0
  final_distance_mean < 0.05

Full multi-seed results: see REPRODUCE IT at end of chapter.
------------------------------------------------------------
```

The Push environment

FetchPush-v4 is a step up from the reaching tasks you have seen so far. Instead of moving the gripper to a target position, the robot must push a puck across a table to a goal location. Figure 5.4 shows the environment layout.

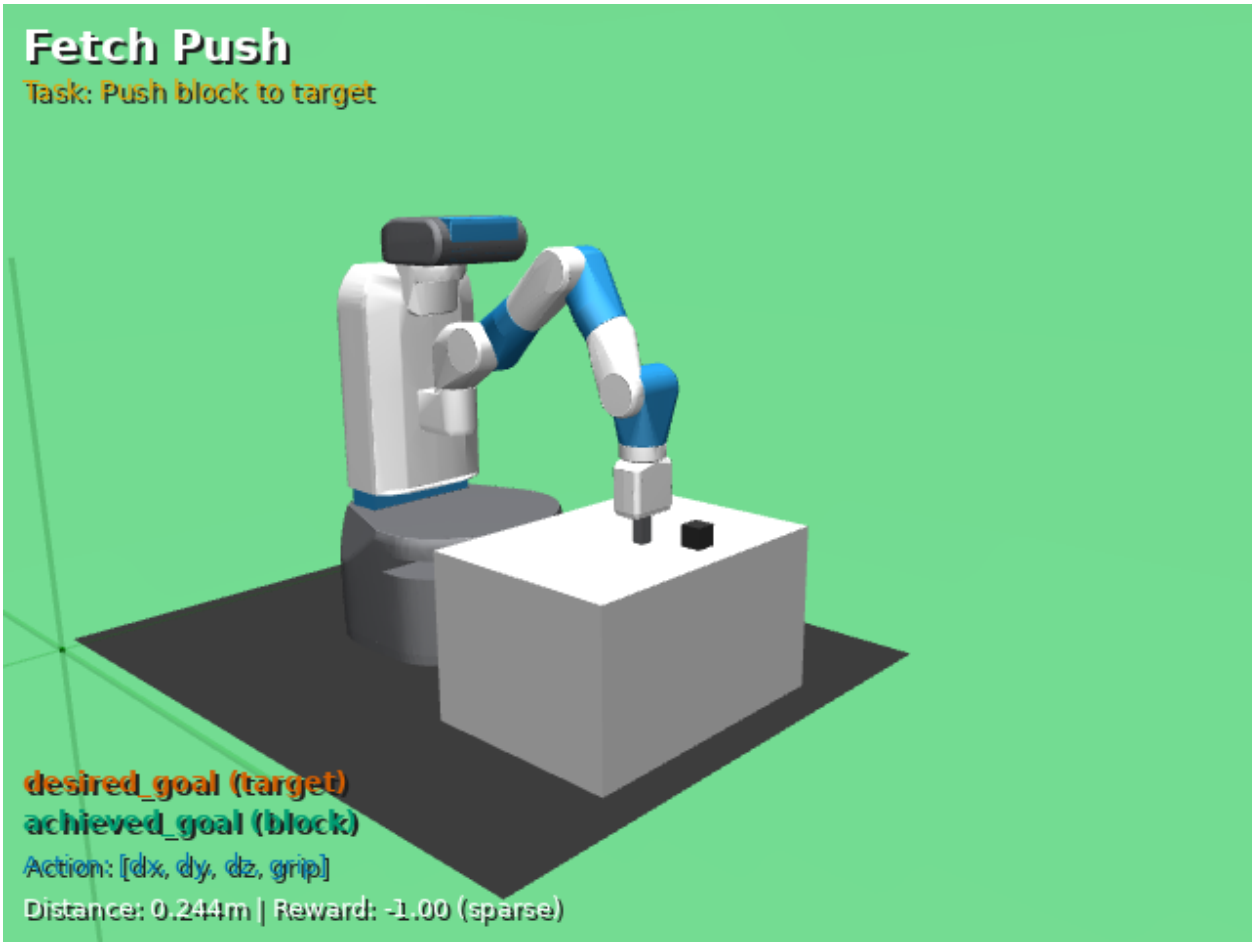

Figure 5.4: FetchPush-v4 environment. A 7-DoF robotic arm must push a puck (cylindrical object on the table) to a target position (red marker). The observation is 25-dimensional (gripper state + puck state + relative positions), the action is 4D (Cartesian deltas + gripper), and the reward is sparse: 0 if the puck is within 5cm of the goal, -1 otherwise. (Generated by `python scripts/capture_proposal_figures.py env-setup`.)

The observation is 25-dimensional -- gripper position, gripper velocity, puck position, puck velocity, and relative positions between gripper, puck, and goal. The action is the same 4D Cartesian delta from Chapter 2 (dx, dy, dz, gripper). The complexity comes from indirect control: the gripper must contact the puck and push it in the right direction, requiring a coordinated sequence of approach, contact, push, and stop.

Two key hyperparameters for sparse Push

Before running the experiment, we need to explain two hyperparameter choices that differ from Chapter 4's defaults:

`ent_coef=0.05` (fixed, not auto-tuned). SAC's automatic entropy tuning (Chapter 4, Section 4.8) works well on dense rewards but fails on sparse tasks. The reason is that with sparse rewards, almost every transition has $r = -1$ early in training, so the critic sees no reward variation and Q-values remain nearly flat. The policy then becomes

arbitrarily "confident" (low entropy), and the auto-tuner interprets this as "the policy knows what it is doing," reducing α to near zero. Once that happens, exploration stops and the agent never discovers push behavior.

Fixed `ent_coef=0.05` bypasses this failure mode. It maintains a small but steady entropy bonus throughout training, ensuring the agent keeps trying diverse push strategies until it discovers effective behavior.

`gamma=0.95` (not 0.99). The discount factor controls how far into the future the agent "looks," with an effective horizon of $T_{\text{eff}} = 1/(1-\gamma)$: at $\gamma = 0.99$, that is 100 steps, while at $\gamma = 0.95$ it is 20 steps. Since a successful push takes roughly 15-25 coordinated steps, $\gamma = 0.95$ matches the task timescale, whereas $\gamma = 0.99$ forces the agent to optimize over steps where it is just waiting after the push is complete -- adding noise without useful signal.

There is also an interaction between these two parameters, because SAC's entropy bonus accumulates over the effective horizon: roughly $\alpha \times T_{\text{eff}} \times \bar{\mathcal{H}}$ total entropy contribution. At $\alpha = 0.05$ and $\gamma = 0.95$ ($T_{\text{eff}} = 20$), this contribution is moderate, but at $\alpha = 0.2$ and $\gamma = 0.98$ ($T_{\text{eff}} = 50$) it becomes 10x larger. In our sweep, we found that high entropy plus long horizon can overwhelm the sparse reward signal entirely.

These values come from a 120-run hyperparameter sweep (24 configurations, 5 seeds each) on FetchPush-v4. The sweep confirmed that γ is the dominant factor (+11 percentage points marginal effect) and that `ent_coef=0.05` outperforms higher values. The full sweep analysis is available in the tutorial repository; for the book, we present the winning configuration directly.

Running the experiments

The one-command version trains both SAC (no HER) and SAC+HER, evaluates both, and generates a comparison report:

```
# FetchReach-v4: validation task (~60 min for all 6 runs)
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-all \
  --env FetchReach-v4 --seeds 0,1,2 --total-steps 1000000

# FetchPush-v4: primary task (~3 hours for all 6 runs)
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-all \
  --env FetchPush-v4 --seeds 0,1,2 --total-steps 2000000 --ent-coef 0.05
```

For a quick validation on a single seed (about 14 minutes on GPU):

```
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05
```

If you are using the checkpoint track, evaluate the pretrained model directly:

```
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py eval \
  --ckpt checkpoints/sac_her_FetchPush-v4_seed0.zip
```

Training milestones for Push

Watch for these milestones during SAC+HER training on FetchPush-v4. The learning curve shows a characteristic pattern -- slow initial progress as HER builds up successful transitions in the buffer, then rapid improvement once the critic has enough signal:

Timesteps	Success Rate	What's happening
0-100k	0-5%	Collecting data, HER creating relabeled successes in buffer
100k-500k	5-40%	Critic learning from relabeled successes, policy improving
500k-1M	40-80%	Rapid improvement -- positive feedback loop: better policy -> better tr
1M-2M	80-99%	Convergence, policy refining push accuracy

The no-HER baseline stays at approximately 5% throughout. This flat line is exactly the failure we saw in Section 5.1.

Results: FetchReach-v4 (sparse)

Reach is the validation task. We run it to confirm that HER does not hurt on an easy task. The results (3 seeds, 100 evaluation episodes each, 1M training steps):

Metric	HER	No-HER	Delta
Success Rate	100% +/- 0%	96% +/- 7%	+4%
Mean Return	-1.68 +/- 0.02	-2.92 +/- 2.04	+1.24
Final Distance	17.0mm +/- 6mm	19.5mm +/- 8mm	-2.5mm

The separation is marginal -- both methods succeed. HER achieves a slightly higher success rate (100% vs 96%) and lower variance. This is expected: FetchReach has a short effective horizon (2-5 steps), so random exploration occasionally succeeds even without relabeling. The 96% no-HER result means roughly 4 out of 100 evaluation episodes fail -- usually when the goal spawns at the edge of the workspace.

Reach confirms that HER does not hurt on an easy task. To see where HER becomes essential, we turn to Push.

Results: FetchPush-v4 (sparse)

Push is where HER's effect becomes clear. The results (3 seeds, 100 evaluation episodes each, 2M training steps):

Metric	HER	No-HER	Delta
Success Rate	99% +/- 1%	5% +/- 0%	**+94%**
Mean Return	-13.20 +/- 1.48	-47.50 +/- 0.00	+34.30
Final Distance	25.7mm +/- 1mm	184.5mm +/- 0mm	-158.8mm

The improvement is transformative. Without HER, the puck ends up 184.5mm from the goal on average -- the agent has not learned to push at all. With HER, the puck ends up 25.7mm from the goal -- well within the 50mm success threshold. The success rate jumps from 5% to 99%, a 94 percentage point improvement that is consistent across all 3 seeds.

Figure 5.5 shows the learning curves for both environments, making the contrast visible.

HER vs no-HER learning curves on FetchReach-v4 and FetchPush-v4: left panel shows both methods reaching near-100 percent success on Reach, right panel shows no-HER stuck at 5 percent while HER climbs to 99 percent on Push.

Figure 5.5: HER vs no-HER learning curves. Left: FetchReach-v4 -- both methods succeed, with HER providing a marginal improvement (100% vs 96%). Right: FetchPush-v4 -- without HER, success stays flat at approximately 5%; with HER, it climbs to 99%. The shaded regions show +/- one standard deviation across 3 seeds. The contrast on Push makes HER's value concrete: a task that SAC alone cannot solve becomes a 99% success rate with goal relabeling. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/sac{,_her}/Fetch{Reach,Push}-v4/seed{0,1,2}/`.)

What the mean return tells you

The mean return of -13.20 for Push with HER deserves a brief interpretation. Each episode is 50 steps, and the reward is -1 for failure, 0 for success, so a return of -13.20 means the agent fails for roughly the first 13 steps (approaching and aligning the push) and then succeeds for the remaining 37 steps. This maps directly to the task structure: the puck needs to be contacted and pushed before the agent can "succeed" at each timestep. A perfect policy that pushes immediately would have a return near -5 to -10 (a few steps to approach), whereas a return of -47.50 means failure at almost every step -- the no-HER agent never contacts the puck meaningfully.

Reading TensorBoard during training

Launch TensorBoard to watch the experiments:

```
bash docker/dev.sh tensorboard --logdir runs --bind_all
```

The key metrics map directly to the concepts you built in the Build It sections:

SB3 TensorBoard key	What it measures	HER expected behavior
`rollout/success_rate`	Fraction of episodes reaching the goal	0% -> 99% over 2M steps
`rollout/ep_rew_mean`	Mean episode return	-50 -> -13 (closer to 0 is better)
`train/ent_coef`	Entropy coefficient α	Fixed at 0.05 (recommended for
`train/critic_loss`	Twin Q-network Bellman error	Should decrease and stabilize
`train/actor_loss`	Policy loss (entropy-regularized)	Should decrease as policy impro

The `rollout/success_rate` metric is what HER directly improves. Without HER, the replay buffer contains almost exclusively $r = -1$ transitions, so the critic cannot

distinguish good from bad actions. With HER, the relabeled successes from Section 5.6 -- the same `process_episode_with_her` math that SB3's `HerReplayBuffer` computes -- provide the gradient signal the critic needs.

5.9 What can go wrong

Here are the failure modes we have encountered when training SAC+HER on sparse rewards. For each, we give the symptom, the likely cause, and a specific diagnostic.

Push success_rate stalls at approximately 5% with HER enabled

Likely cause. The entropy coefficient is too high. With `ent_coef=auto` or a value above 0.1, SAC over-explores and never exploits the successful relabeled transitions. The entropy bonus overwhelms the sparse reward signal.

Diagnostic. Check three things: (1) verify `ent_coef=0.05` (fixed, not auto-tuned); (2) verify the `--her` flag is actually passed -- check the checkpoint's `.meta.json` to confirm `"her": true`; (3) look at TensorBoard's `train/ent_coef` -- if it shows a value much higher than 0.05, the fixed setting did not take effect.

Push success_rate stalls at approximately 5% even with correct entropy

Likely cause. Wrong discount factor. With `gamma=0.99`, the effective horizon is 100 steps -- far beyond the 15-25 steps a push takes. The critic tries to estimate value over steps where the agent is idle, adding noise.

Diagnostic. Verify `gamma=0.95` (not 0.99). Also check that `total_steps >= 2000000` -- Push needs substantially more training than Reach. Finally, verify `n_envs=8` -- fewer environments mean slower data collection.

Reach success_rate is 0% for the first 100k steps then jumps to 100%

Likely cause. This is normal behavior, not a bug. SAC+HER on sparse Reach takes longer to start learning than dense Reach because the initial replay buffer contains only $r = -1$ transitions. Once enough relabeled successes accumulate, the critic rapidly learns the value function. This is the ”hockey-stick” learning curve pattern.

Diagnostic. No fix needed. Let it train. The jump is expected once the buffer has enough relabeled successes for the critic to distinguish promising states from hopeless ones.

`ep_rew_mean` stays at -50 throughout training

Likely cause. HER is not enabled, or the environment is wrong. A mean return of -50 on a 50-step episode means every single timestep is a failure ($r = -1$), which indicates the agent has not learned any push behavior at all.

Diagnostic. Verify the `--her` flag is set. Check that the environment is `FetchPush-v4` (not `FetchPushDense-v4`, which uses a different reward scale). Print the SB3 model to confirm `HerReplayBuffer` is being used as the replay buffer class.

`--compare-sb3` fails with reward mismatch

Likely cause. SB3 version incompatibility. The `HerReplayBuffer` API changed between SB3 versions, and older versions may handle batched `compute_reward` calls differently.

Diagnostic. Check that `stable-baselines3 >= 2.0` is installed. Verify that the `GoalEnvStub` used in the comparison handles batched inputs correctly -- `compute_reward(achieved_goal, desired_goal, info)` must work when `achieved_goal` and `desired_goal` are 2D arrays, not just 1D.

`--compare-sb3` shows success_fraction equal to 0.0

Likely cause. The synthetic episode's goals are not far enough apart. If the desired goal happens to be near the achieved goals, all original transitions are already successes, and HER relabeling has no failures to turn into successes.

Diagnostic. Verify that the stub environment places the desired goal at least 10 units from the achieved goals (well beyond the 0.05m success threshold). Check that `n_sampled_goal=4` -- a value of 0 would mean no relabeling occurs.

Training much slower than expected (below 300 fps on GPU)

Likely cause. HER adds overhead to replay buffer sampling. Each minibatch requires goal relabeling and reward recomputation, which adds approximately 20% overhead compared to plain SAC.

Diagnostic. Check that `gradient_steps=1` (the default). Approximately 500 fps is typical for SAC+HER on Fetch tasks with GPU. If throughput is below 200 fps, run `nvidia-smi` inside the container to verify the GPU is in use.

NaN in Q-values during Push training

Likely cause. Sparse reward combined with high gamma creates large Bellman targets. The entropy bonus accumulates over a long effective horizon, and the resulting target values can exceed the range of float32 precision.

Diagnostic. Reduce `gamma` from 0.99 to 0.95. Verify that `ent_coef` is not 0.0 (some exploration is needed to populate the buffer with diverse transitions). Check that rewards are the expected binary values (0 or -1), not something unexpected from a misconfigured environment.

Reach HER barely outperforms no-HER baseline

Likely cause. This is expected, not a failure. FetchReach-v4 has a short effective horizon (2-5 steps), so random exploration occasionally reaches the goal by chance. Without HER, those accidental successes are sufficient for learning. HER's benefit is marginal on tasks where random success is common.

Diagnostic. Run the Push experiment to see the dramatic improvement. HER's value scales with the difficulty of the exploration problem -- the harder the task, the larger the gap between HER and no-HER.

5.10 Summary

This chapter introduced Hindsight Experience Replay and demonstrated its transformative effect on sparse-reward goal-conditioned tasks. Here is what you accomplished:

- **The sparse reward problem.** You trained SAC without HER on FetchPush-v4 and observed a 5% success rate across 3 seeds -- consistent across seeds, pointing to a structural limitation rather than unlucky initialization. Sparse rewards provide almost no gradient signal because the agent rarely succeeds by chance. The effective horizon for Push (20-50 coordinated steps) makes random success vanishingly improbable.
- **The HER insight.** A trajectory that fails to reach its intended goal is a successful demonstration of reaching wherever it ended up. By relabeling the desired goal with an achieved goal and recomputing the reward, HER turns failures into learning signal.
- **From-scratch implementation.** You built four components: data structures (the 7-tuple `Transition`, `Episode`, `GoalStrategy` enum), goal sampling (FUTURE, FINAL, EPISODE strategies), transition relabeling with reward recomputation (the critical invariant -- never assume the reward, always recompute it), and the full episode processing pipeline. A 50-step episode with $k = 4$ and `her_ratio=0.8` produces approximately 210 transitions with a success fraction that jumps from 0% to 4-80% depending on trajectory coherence.
- **Bridge to SB3.** The bridging proof confirmed that our relabeled rewards match SB3's `HerReplayBuffer` output exactly (max absolute reward difference = 0.0). SB3 adds vectorized episode storage, automatic boundary detection, and online relabeling at sample time -- but computes the same relabeling math.
- **Production results.** SAC+HER on FetchPush-v4 achieved 99% +/- 1% success rate across 3 seeds, a 94 percentage point improvement over the no-HER baseline. On FetchReach-v4, HER provided a marginal improvement (100% vs 96%), confirming it does not hurt on easy tasks but showing that Push is where HER's value becomes visible.

Two hyperparameter choices matter for sparse Push: fixed `ent_coef=0.05` (bypassing auto-tuning that collapses on sparse rewards) and `gamma=0.95` (matching the 15-25 step task timescale). These came from a 120-run hyperparameter sweep that identified gamma as the dominant factor.

What comes next

HER solves the exploration problem for goal-conditioned tasks where the goal space is known and the agent can learn from achieved goals. But Push is a table-level task -- the puck stays on the surface. Chapter 6 applies the full SAC+HER stack to FetchPickAndPlace-v4, the hardest Fetch task, where the robot must lift an object off the table and place it at a 3D goal that can be above the surface. PickAndPlace introduces new challenges: the gripper must close on the object (a discrete-like decision within continuous actions), grasp stability matters (the object can slip), and the goal space is three-dimensional rather than two-dimensional. We will explore curriculum strategies that introduce goals at increasing difficulty and stress-test the trained policy to build confidence before deployment.

The off-policy machinery from Chapter 4 (SAC's replay buffer and value estimation) and the goal relabeling from this chapter (HER's data amplification) form the complete algorithmic stack. Chapter 6 is about applying that stack to a harder task and developing the engineering practices -- curriculum, evaluation, robustness testing -- that bridge the gap between ”it works on the benchmark” and ”it works reliably.”

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
The results and pretrained checkpoints in this chapter
come from these runs:

  # Sparse Reach: HER vs no-HER (3 seeds, 1M steps each)
   bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-
all \
    --env FetchReach-v4 --seeds 0,1,2 --total-steps 1000000

  # Sparse Push: HER vs no-HER (3 seeds, 2M steps each)
   bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-
all \
   --env FetchPush-v4 --seeds 0,1,2 --total-steps 2000000 --ent-coef 0.05

Hardware:    Any machine with Docker (GPU optional; tested on NVIDIA GB10)
Time:      ~28 min per seed for Reach (Linux GPU), ~56 min per seed for Push (Linux GPU
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/sac_her_FetchReach-v4_seed{0,1,2}.zip
  checkpoints/sac_her_FetchReach-v4_seed{0,1,2}.meta.json
  checkpoints/sac_FetchReach-v4_seed{0,1,2}.zip       (no-HER baselines)
  checkpoints/sac_her_FetchPush-v4_seed{0,1,2}.zip
  checkpoints/sac_her_FetchPush-v4_seed{0,1,2}.meta.json
```

```
  checkpoints/sac_FetchPush-v4_seed{0,1,2}.zip        (no-HER baselines)
   results/ch04_fetchreach-v4_comparison.json
   results/ch04_fetchpush-v4_comparison.json

Results summary (what we got):

  FetchReach-v4 (sparse):
                    HER          no-HER
    success_rate:  1.00 +/- 0.00   0.96 +/- 0.07   (3 seeds x 100 episodes)
     return_mean:  -1.68 +/- 0.02  -2.92 +/- 2.04
     final_dist:    17.0mm +/- 6mm  19.5mm +/- 8mm

  FetchPush-v4 (sparse):
                    HER          no-HER
    success_rate:  0.99 +/- 0.01   0.05 +/- 0.00   (3 seeds x 100 episodes)
     return_mean: -13.20 +/- 1.48  -47.50 +/- 0.00
     final_dist:    25.7mm +/- 1mm  184.5mm +/- 0mm

If your numbers differ by more than ~10%, check the
"What Can Go Wrong" section above.

The pretrained checkpoints are available in the book's
companion repository for readers using the checkpoint track.
------------------------------------------------------------
```

Exercises

1. (Verify) Reproduce the single-seed Push baseline.

Run the fast path command and verify your results match:

```
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py eval \
  --ckpt checkpoints/sac_her_FetchPush-v4_seed0.zip
```

Check the eval JSON: `success_rate` should be >= 0.90, `mean_return` > -20.0. Compare training time to Chapter 4's SAC on FetchReachDense (expect approximately 20% overhead from HER relabeling).

2. (Tweak) Change n_sampled_goal.

The default is $k = 4$. Try $k = 2$ and $k = 8$:

```
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 2000000 --ent-coef 0.05 \
  --n-sampled-goal 2
```

Compare `success_rate` at convergence. Expected: $k = 2$ may be slightly slower to converge; $k = 8$ may be slightly faster but with diminishing returns. The original paper

(Andrychowicz et al., 2017) found $k = 4$ to be a good balance, and our 120-run sweep confirmed that $k = 8$ provides only +1.2 percentage points -- within seed variance.

3. (Tweak) Replace FUTURE strategy with FINAL.

In the lab's `process_episode_with_her()`, change the default strategy from `GoalStrategy.FUTURE` to `GoalStrategy.FINAL`. Run `--demo` and compare the success fraction. Expected: FINAL produces slightly lower success fraction because all relabeled goals are the same (the episode's last achieved goal), providing less diversity. FUTURE provides more diverse relabeled goals from different timesteps, which helps the critic learn a richer value function.

4. (Extend) Visualize the data amplification effect.

Write a short script that processes 10 synthetic episodes with `process_episode_with_her()` and plots three things using matplotlib: (a) number of original vs relabeled transitions per episode, (b) success fraction before and after HER for each episode, (c) histogram of distances between achieved goals and relabeled desired goals. Use the lab's `create_synthetic_episode()` function. Expected: you should see approximately 4.2x amplification per episode, with success fractions varying based on how clustered each trajectory's achieved goals are.

5. (Challenge) SAC without HER on sparse Reach vs Push.

Run the no-HER baseline on both environments:

```
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchReach-v4 --seed 0 --total-steps 1000000
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --seed 0 --total-steps 2000000 --ent-coef 0.05
```

Compare the two: Reach achieves approximately 96% without HER, Push achieves approximately 5%. Explain the difference in terms of the effective horizon T_{eff}: for Reach, random exploration reaches the goal within 2-5 steps (the gripper workspace is about 15cm and the success threshold is 5cm). For Push, the puck must be contacted AND pushed to the target, requiring 20-50 coordinated steps. The probability of a random success drops exponentially with the effective horizon. HER's benefit scales directly with this difficulty -- the harder the exploration problem, the more HER matters.

\newpage

Part 4 -- Engineering-Grade Robotics RL

\newpage

9 Pixels, No Cheating: From State Vectors to Camera Images

This chapter covers:

- Building a complete visual observation pipeline from scratch -- pixel wrappers, CNN encoders, spatial feature extraction, and memory-efficient replay buffers -- that turns 84x84 camera images into features a policy can act on
- Understanding why NatureCNN fails on manipulation tasks (stride-4 destroys 5-pixel objects) and implementing ManipulationCNN + SpatialSoftmax as the fix
- Discovering why architecture alone is not enough: SB3's default gradient routing starves the encoder of learning signal, and the DrQ-v2 pattern (encoder in the critic's optimizer) is the 15 lines that make the difference between 5% and 95%
- Reading the three-phase loss signature in pixel RL -- declining losses with flat success means the critic is memorizing failure, not learning structure; rising losses during the hockey-stick means real value learning has begun
- Running a 5-step progressive investigation that arrives at a pixel Push agent achieving 95%+ success, with each failure teaching a transferable debugging principle

Through Chapters 3-8, you built SAC from scratch, added HER for sparse rewards, solved Push at 89% from state vectors, tested robustness under noise, and learned to tune hyperparameters systematically. You have a working, validated, robust manipulation pipeline -- but it assumes the agent directly observes state vectors: joint positions, object coordinates, goal positions.

Real robots do not observe state vectors. They see pixels from cameras. Can the pipeline you built survive the switch from 25 numbers to 84x84 images? The answer is: not without significant changes to the encoder, the gradient routing, and the training budget. This chapter adds a visual observation pipeline (pixel wrappers, CNN encoders, spatial feature extraction), the gradient routing that makes the encoder actually learn (the critic-encoder pattern from DrQ-v2), and the diagnostic skills to read pixel RL training curves. You will discover these through a 5-step investigation: try the obvious thing, watch it fail, diagnose why, fix one component at a time, and arrive at a working solution. This chapter achieves 95% Push from pixels, matching state-based performance at the cost of 2-4x more training steps. Chapter 10 will explore how pre-trained encoders and world models can reduce this overhead.

9.1 WHY: The pixel penalty

What changes

State-based Push uses a 25-dimensional vector: 3D gripper position, 3D object position, 3D relative position, 2D gripper state, 3D object rotation, and various velocities. Every number is precise to millimeter accuracy, calibrated, and cheap to process -- a two-layer MLP (256 x 256 parameters) handles it in microseconds, and the GPU sits idle at 5-10% utilization because the bottleneck is CPU-side MuJoCo simulation.

Pixel Push replaces most of that with a camera image: 84 x 84 x 3 = 21,168 values per frame, stacked 4 deep for temporal information, producing 84 x 84 x 12 = 84,672 val-

ues per observation. These values are raw pixel intensities, uncalibrated, and expensive to process through a convolutional neural network. GPU utilization rises to 40-60% because the CNN processes batches of pixel observations for both `obs` and `next_obs` on every training step. Training throughput drops from roughly 600 fps (state-based) to 30-50 fps (pixel-based). Figure 9.1 shows what the policy actually sees.

An 84x84 pixel observation from FetchPush-v4 showing the gripper (approximately 4 pixels wide), the puck (approximately 5 pixels wide), and the goal marker on the table surface

Figure 9.1: FetchPush-v4 at 84x84 resolution -- what the pixel agent sees. The puck is roughly 5 pixels wide and the gripper roughly 4 pixels wide. The spatial relationship between them -- the one thing the policy needs to act on -- occupies a tiny fraction of the image. (Generated by `bash docker/dev.sh python scripts/ch09_pixel_push.py render-frame --seed 0`.)

Four compounding challenges

Every pixel RL challenge from earlier chapters compounds when we add object interaction:

1. Tiny objects. The puck in FetchPush is roughly 5 pixels wide at 84x84 resolution, and the gripper is roughly 4 pixels, so the spatial relationship between them -- the signal the policy needs to act on -- occupies a tiny fraction of the image.

2. Sparse rewards plus pixels. FetchReachDense gave continuous distance feedback, meaning every arm movement changed the reward. FetchPush with sparse rewards ($R = 0$ on success, $R = -1$ otherwise) means the CNN must learn useful spatial features with almost no reward signal. HER helps by relabeling goals, but the CNN still needs to extract spatial coordinates from pixels before HER's relabeled rewards become useful.

3. Two learning problems at once. The agent must simultaneously learn visual representations (CNN: pixels to spatial features) and a control policy (actor-critic: spatial features to push actions). In state-based Push, the first problem does not exist because the observation IS the spatial features. Adding pixels means the agent must solve representation learning AND policy learning from scratch, using the same sparse reward signal for both.

4. Contact dynamics from images. Pushing requires understanding what happens after contact -- does the puck move in the right direction, and how far? This temporal reasoning must be inferred from sequences of pixel observations. Frame stacking (4 frames, concatenated along the channel dimension) provides a weak velocity signal, since pixel differences between consecutive frames imply motion direction and speed. But the motion signal is subtle at 84x84: a puck moving 1 cm per timestep shifts by roughly 1 pixel. Without frame stacking, the environment becomes a POMDP (partially observable MDP), because a single static image cannot distinguish ”puck moving left” from ”puck moving right.”

The observation design space

Our `PixelObservationWrapper` with `goal_mode="both"` and proprioception produces:

```
{
  "pixels":        (12, 84, 84)  uint8   # 4-frame stack x 3 RGB channels
  "proprioception": (10,)      float64 # grip_pos, gripper_state, velocities
  "achieved_goal":  (3,)          float64  # object position (for HER)
  "desired_goal":   (3,)          float64  # target position (for HER)
}
```

The CNN processes only the `pixels` key. Proprioception, achieved_goal, and desired_goal are concatenated as flat vectors alongside the CNN features:

```
pixels (12, 84, 84)     -> CNN -> spatial features (64D)
proprioception (10D)     -> passthrough
achieved_goal (3D)       -> passthrough
desired_goal (3D)        -> passthrough
                            Total: 80D feature vector
```

Why proprioception? The CNN should only learn about the WORLD -- where the puck is, where obstacles are -- because the robot's own state (joint positions, velocities, gripper width) comes from direct sensors with millimeter precision at microsecond latency. Forcing the CNN to also learn ”where is my arm?” wastes capacity on a problem that cheaper sensors already solve. This mirrors how real robotic systems operate (joint encoders plus cameras, never cameras alone), and we call it the **sensor separation principle**.

Visual HER: two kinds

”Visual HER” can mean two very different architectures:

Approach	Policy sees	HER relabels	Com
Our approach (`goal_mode="both"`)	Pixels + goal vectors	3D goal vectors (swap)	Mod
Full visual HER (Nair et al., 2018, RIG)	Pixels + goal *images*	Goal images (requires VAE)	High

We use the first approach: the policy sees pixel observations, but HER operates on 3D goal vectors (`achieved_goal`, `desired_goal`). Relabeling is a swap of two 3D vectors. No image-space goal representation is needed.

This creates an intentional **information asymmetry**: the policy must learn to control from pixels, but it does not need to learn to specify goals from pixels. We are testing whether a CNN can learn spatial features good enough for manipulation, not whether it can learn a visual goal representation. Full visual HER (Nair et al., 2018) tackles the harder problem of specifying goals as images, requiring a VAE to map images to a latent goal space where relabeling is meaningful. That is a worthwhile problem, but it adds an entire representation learning subsystem on top of what is already a challenging pixel RL task. We scope this chapter to the first approach: prove the CNN can learn to control, using vector goals as scaffolding.

Hadamard check

Before investing 40+ hours of GPU time per seed (at 30-50 fps, 5M steps takes 28-46 hours), we check our three practical questions. **Can this be solved?** Yes -- we have the 89% state-based result as existence proof, and the information is visually present in the image (you can see the puck and gripper), so the question reduces to whether a CNN can extract it. **Is the solution reliable?** We will need 3 seeds to find out, since one seed at 95% could be lucky. **Is the solution stable?** Prior experience with pixel RL says no -- small changes in hyperparameters, encoder architecture, or gradient routing can mean the difference between 95% and 5%. Henderson et al. (2018) showed that even state-based RL exhibits high variance across seeds and implementations, and adding a CNN encoder multiplies the sensitivity surface. This chapter maps that sensitivity by systematically varying components and observing their impact.

9.2 Build It: The visual observation pipeline

We now build the components that convert raw camera frames into training data. The visual pipeline has three parts: a rendering function that captures camera images, a wrapper that integrates them into Gymnasium's observation structure with frame stacking and proprioception, and a replay buffer that stores pixel transitions without exhausting system memory. Each component is individually testable, so we verify shapes and types before connecting them.

9.2.1 Rendering and resizing

The rendering function captures a MuJoCo camera frame and converts it to a CHW uint8 tensor that PyTorch and SB3 expect. MuJoCo's `render()` returns an HWC array (height x width x channels) -- the image format used by NumPy and PIL -- but PyTorch and SB3 expect CHW format (channels first), so we transpose. This transpose is a zero-cost memory reinterpretation, not a data copy.

MuJoCo renders at its default resolution (480x480 for Fetch), and we resize to 84x84 via PIL bilinear interpolation. When you create the environment with `gym.make("FetchPush-v4", render_mode="rgb_array", width=84, height=84)`, MuJoCo renders directly at 84x84 and the resize step is skipped entirely -- a worthwhile optimization since rendering happens every step.

Listing 9.1: render_and_resize -- camera capture to CHW tensor

```
def render_and_resize(
    env: gym.Env,
    image_size: tuple[int, int] = (84, 84),
) -> np.ndarray:
    """Render MuJoCo scene -> CHW uint8 array (3, H, W)."""
    frame = env.render()                # HWC uint8 (480x480x3)
    assert frame is not None, "render_mode must be 'rgb_array'"

    # Fast path: already at target size (native rendering)
    if frame.shape[:2] == image_size:
```

```
        return frame.transpose(2, 0, 1).copy()

    # Resize via PIL bilinear interpolation
    from PIL import Image
    pil_img = Image.fromarray(frame)
    pil_img = pil_img.resize(
        (image_size[1], image_size[0]), Image.Resampling.BILINEAR
    )
    return np.array(pil_img, dtype=np.uint8).transpose(2, 0, 1)
```

The `.copy()` on the fast path matters -- MuJoCo may reuse the internal render buffer on the next call, so we need the array to own its data.

> **Checkpoint:** The output should be `(3, 84, 84)` with dtype `uint8` and values in `[0, 255]`. If the image is all black, check that `render_mode="rgb_array"` was passed to `gym.make()`.

9.2.2 PixelObservationWrapper

The wrapper replaces Gymnasium's flat `"observation"` key with a `"pixels"` key, optionally stacks frames for temporal information, and passes through proprioception and goal vectors.

Listing 9.2: PixelObservationWrapper -- core observation method

```
class PixelObservationWrapper(gym.ObservationWrapper):
    def observation(self, observation):
        self.last_raw_obs = observation
        pixels = render_and_resize(self.env, self._image_size)

        if self._frame_stack > 1:
            pixels = self._get_stacked_pixels(pixels)

        out = {"pixels": pixels}
        if self._proprio_indices is not None:
            out["proprioception"] = (
                observation["observation"][self._proprio_indices]
            )
        if self._goal_mode in {"desired", "both"}:
            out["desired_goal"] = observation["desired_goal"]
        if self._goal_mode == "both":
            out["achieved_goal"] = observation["achieved_goal"]
        return out
```

Frame stacking concatenates the last N frames along the channel dimension: 4 frames of RGB produce `(12, 84, 84)`. On reset, the deque is filled with copies of the first frame so the stack is always complete -- this avoids a cold-start artifact where the first few observations would otherwise contain stale frames from a previous episode.

The `last_raw_obs` attribute stores the unwrapped Gymnasium observation for debugging. When you need to check the ground-truth object position or goal distance, `wrapper.last_raw_obs["achieved_goal"]` gives you the 3D coordinates without parsing the pixel observation.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/pixel_wrapper.py --verify`. Expected: `[ALL PASS] Pixel wrapper verified`. Verify the output observation dict has keys `pixels` (12, 84, 84), `proprioception` (10,), `achieved_goal` (3,), `desired_goal` (3,).

9.2.3 Pixel replay buffer with uint8 storage

Pixel observations are expensive to store. Each frame is `12 x 84 x 84 = 84,672` bytes as uint8. Each transition stores both `obs` and `next_obs`, so:

$$\text{bytes per transition} = 84{,}672 \times 2 = 169{,}344 \approx 165 \text{ KB}$$

> **Sidebar: The memory wall.** A 500,000-transition pixel buffer costs $500{,}000 \times 169{,}344 \approx 80$ GB for pixel arrays alone. Adding proprioception, goals, actions, and rewards brings the total to roughly 85 GB. On our DGX with 119 GB RAM, this leaves about 34 GB for the OS, CUDA, and the model -- tight but feasible. A 1M buffer would need 170 GB and is physically impossible on this machine. For comparison, a 500K state-based buffer uses about 250 MB. Pixels cost 340x more per transition.

What if you don't have 120 GB? Smaller buffers work, at the cost of delayed or weaker convergence. The buffer retains early diverse exploration that HER relabels into learning signal; when old episodes are overwritten before their value propagates through the Bellman equation, the hockey-stick ignites later or stalls at a lower plateau.

Buffer size	Buffer RAM	Total needed	Expected effect
500K	~80 GB	~85-90 GB	Full performance; hockey-stick at ~2.2M steps
300K	~48 GB	~53-58 GB	Hockey-stick may shift to ~3M; final success 90%+
200K	~32 GB	~37-42 GB	Slower convergence; may need 5M+ steps for 90%
100K	~16 GB	~21-24 GB	May plateau at 70-85%; consider `--full-state` first

Readers with 64 GB should use `--buffer-size 300000`. Readers with 32 GB should use `--buffer-size 100000`. Below 32 GB, pixel Push training is experimental -- verify the pipeline with `--full-state` first, then try pixels with `--buffer-size 50000`.

The key insight is to store pixels as uint8 (1 byte per value) and convert to float32 only at sample time. The conversion cost is negligible for a 256-sample batch (256 x 84,672 x 4 bytes = 87 MB for obs + next_obs) but would quadruple memory if applied to the full buffer. SB3's `DictReplayBuffer` already stores pixels as uint8 when the observation space dtype is `np.uint8`, so our wrapper defines the pixel space with `dtype=np.uint8` to ensure this behavior. One caveat: SB3's `DictReplayBuffer` does

NOT support `optimize_memory_usage=True` (which would avoid storing `next_obs` separately for a 2x savings), raising `ValueError` if you try, so we live with the full obs + next_obs cost.

Listing 9.3: PixelReplayBuffer -- uint8 storage, float32 sampling

```
class PixelReplayBuffer:
    def __init__(self, img_shape, goal_dim, act_dim, capacity):
        # uint8 storage: 1 byte/pixel (not 4 bytes for float32)
        self.pixels = np.zeros((capacity, *img_shape), dtype=np.uint8)
        self.next_pixels = np.zeros((capacity, *img_shape), dtype=np.uint8)
        # Goals, actions, rewards are small -- float32 is fine
        self.goals = np.zeros((capacity, goal_dim), dtype=np.float32)
        self.achieved = np.zeros((capacity, goal_dim), dtype=np.float32)
        self.actions = np.zeros((capacity, act_dim), dtype=np.float32)
        self.rewards = np.zeros((capacity, 1), dtype=np.float32)
        self.dones = np.zeros((capacity, 1), dtype=np.float32)
        self.capacity, self.size, self.pos = capacity, 0, 0

    def add(self, obs_pixels, next_pixels, achieved, goal, action, reward, done):
        self.pixels[self.pos] = obs_pixels          # store uint8 directly
        self.next_pixels[self.pos] = next_pixels
        self.goals[self.pos] = goal
        self.achieved[self.pos] = achieved
        self.actions[self.pos] = action
        self.rewards[self.pos] = reward
        self.dones[self.pos] = done
        self.pos = (self.pos + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)

    def sample(self, batch_size):
        idx = np.random.randint(0, self.size, size=batch_size)
        return {
            # Convert to float32 [0, 1] at sample time only
            "pixels": self.pixels[idx].astype(np.float32) / 255.0,
            "next_pixels": self.next_pixels[idx].astype(np.float32) / 255.0,
            "achieved_goal": self.achieved[idx],
            "desired_goal": self.goals[idx],
            "actions": self.actions[idx],
            "rewards": self.rewards[idx],
            "dones": self.dones[idx],
        }
```

This is the complete from-scratch implementation. The core insight is in the dtype asymmetry: `__init__` allocates `uint8` for pixels (1 byte each), while `sample` converts to `float32` (4 bytes each). The conversion cost is negligible for a 256-sample batch but would quadruple memory if applied to the full 500K-transition buffer. SB3's `DictReplayBuffer` implements this same pattern automatically when the observation space dtype is `np.uint8`. In Run It, we use SB3's production buffer for its integration with

vectorized environments and HER -- but the storage principle is identical to what you see here.

> **Checkpoint:** Create a buffer with `capacity=100`, add 50 transitions, sample a batch of 16. The internal `.pixels` array should have dtype `uint8`; the sampled batch should have dtype `float32` with values in `[0, 1]`.

9.3 Build It: Encoder architecture

The encoder is where pixel observations become features a policy can act on. We build two encoders -- NatureCNN (the "wrong" one, to understand why it fails) and ManipulationCNN (the "right" one) -- then add SpatialSoftmax to extract spatial coordinates and ManipulationExtractor to wire everything together for SB3.

9.3.1 NatureCNN -- the "wrong" encoder

NatureCNN (Mnih et al., 2015) is the default visual encoder in SB3 and was the workhorse of the Atari RL era. Its architecture uses large kernels with aggressive downsampling:

```
NatureCNN spatial progression (84x84 input):
  Layer 1: Conv2d(C, 32, 8x8, stride=4)   84 -> 20   (4x reduction!)
  Layer 2: Conv2d(32, 64, 4x4, stride=2)   20 ->  9
  Layer 3: Conv2d(64, 64, 3x3, stride=1)    9 ->  7
  Flatten -> Linear(3136, 512)
```

Listing 9.4: NatureCNN -- Atari-era encoder (from `scripts/labs/visual_encoder.py:nature`

```python
class NatureCNN(nn.Module):
    def __init__(self, in_channels=3, features_dim=512):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
        )
        with torch.no_grad():
            n_flat = self.conv(torch.zeros(1, in_channels, 84, 84)).shape[1]
        self.fc = nn.Sequential(
            nn.Linear(n_flat, features_dim), nn.ReLU()
        )
```

The problem is in the first layer. The puck is roughly 5 pixels wide, so after stride-4 it becomes roughly 1 pixel, which means the spatial relationship between gripper and puck -- the signal the policy needs -- is destroyed in the FIRST layer.

NatureCNN was designed for Atari, where game sprites are 10-30 pixels wide and decisions are coarse ("go left" vs "go right"). Manipulation requires millimeter-precision spatial reasoning about objects that are 3-5 pixels wide, making the architecture fundamentally mismatched.

In our experiments, NatureCNN achieves 5-8% success on FetchPush across 2M+ training steps -- indistinguishable from the random-policy baseline. Since the algorithm (SAC + HER) is proven to work from state at 89%, the encoder is the bottleneck.

9.3.2 ManipulationCNN -- the "right" encoder

The fix is gentle downsampling: 3x3 kernels throughout, stride 2 only in layers 1 and 4, padding to preserve resolution where possible. This follows the DrQ-v2 encoder design (Yarats et al., 2021).

```
ManipulationCNN spatial progression (84x84 input):
  Layer 1: Conv2d(C, 32, 3x3, stride=2, pad=1)   84 -> 42   (2x, not 4x)
  Layer 2: Conv2d(32, 32, 3x3, stride=1, pad=1)   42 -> 42
  Layer 3: Conv2d(32, 32, 3x3, stride=1, pad=1)   42 -> 42
  Layer 4: Conv2d(32, 32, 3x3, stride=2, pad=1)   42 -> 21
  Output: (B, 32, 21, 21) feature map
```

A 5-pixel puck survives layer 1 as roughly 3 pixels -- still wide enough to carry spatial information. After all four layers, the puck occupies roughly 1-2 pixels on the 21x21 feature map. The spatial relationship between gripper and puck is preserved throughout the network, and a subsequent SpatialSoftmax layer (Section 9.3.3) can extract precise coordinates from these activations.

Listing 9.5: ManipulationCNN -- gentle downsampling for small objects

```python
class ManipulationCNN(nn.Module):
    def __init__(self, in_channels=12, num_filters=32):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, num_filters, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=2, padding=1),
            nn.ReLU(),
        )

    def forward(self, pixels):
        """(B, C, 84, 84) float32 [0,1] -> (B, 32, 21, 21) feature map."""
        return self.conv(pixels)
```

Figure 9.2 illustrates the difference side by side.

Side-by-side comparison of NatureCNN and ManipulationCNN spatial progression. NatureCNN: 84 to 20 to 9 to 7 pixels (puck goes from 5px to 1px). ManipulationCNN: 84 to 42 to 42 to 42 to 21 pixels (puck goes from 5px to 3px)

Figure 9.2: NatureCNN vs ManipulationCNN spatial progression. A 5-pixel puck becomes roughly 1 pixel after NatureCNN's stride-4 first layer (left), but survives as roughly 3 pixels through ManipulationCNN's stride-2 downsampling (right). The spatial relationship between gripper and puck -- the critical signal for Push -- is preserved in ManipulationCNN and destroyed in NatureCNN. (Generated by `bash docker/dev.sh python scripts/labs/manipulation_encoder.py --compare-nature`.)

> **Checkpoint:** Feed a `(4, 12, 84, 84)` tensor through ManipulationCNN. Output shape should be `(4, 32, 21, 21)`, all values finite.

9.3.3 SpatialSoftmax -- ”where” not ”what”

ManipulationCNN produces a `(B, 32, 21, 21)` feature map -- 32 channels, each 21x21 pixels. The standard approach would flatten this into a 14,112-dimensional vector, but for manipulation we do not need to know what the image looks like; we need to know WHERE things are.

SpatialSoftmax (Levine et al., 2016) extracts the expected (x, y) coordinate of each channel's activation peak. For C channels, the output is $2C$ values in $[-1, 1]$ -- spatial coordinates, not pixel values.

The operation per channel with height H and width W:

1. Softmax over all $H \times W$ spatial positions: $\alpha_{h,w} = \text{softmax}(f_{h,w}/\tau)$ where τ is a learnable temperature
2. Expected x-coordinate: $\bar{x} = \sum_{h,w} \alpha_{h,w} \cdot \text{pos}_x(w)$, where $\text{pos}_x \in [-1, 1]$
3. Expected y-coordinate: $\bar{y} = \sum_{h,w} \alpha_{h,w} \cdot \text{pos}_y(h)$

The temperature τ is learnable: a high temperature produces uniform attention early in training (when the network is unsure where to look), while a low temperature produces peaked attention once the policy converges (focusing on precise object locations).

Listing 9.6: SpatialSoftmax -- expected (x, y) coordinates per channel

```
class SpatialSoftmax(nn.Module):
    def __init__(self, height, width, num_channels, temperature=1.0):
        super().__init__()
        self.temperature = nn.Parameter(torch.ones(1) * temperature)
        pos_x = torch.linspace(-1.0, 1.0, width)
        pos_y = torch.linspace(-1.0, 1.0, height)
        self.register_buffer("pos_x", pos_x.reshape(1, 1, -1))
        self.register_buffer("pos_y", pos_y.reshape(1, 1, -1))

    def forward(self, features):
        B, C, H, W = features.shape
        attn = F.softmax(
            features.reshape(B, C, -1) / self.temperature, dim=-1
```

```
        ).reshape(B, C, H, W)
        exp_x = (attn.sum(dim=2) * self.pos_x).sum(dim=-1)
        exp_y = (attn.sum(dim=3) * self.pos_y).sum(dim=-1)
        return torch.cat([exp_x, exp_y], dim=-1)  # (B, 2C)
```

With 32 channels, the output is 64 values: 32 x-coordinates and 32 y-coordinates, each in $[-1, 1]$. This is a powerful inductive bias for manipulation -- the policy needs spatial coordinates (where is the puck? where is the gripper?), not a compressed image representation.

The LayerNorm and Tanh layers that follow SpatialSoftmax in the full pipeline (see Listing 9.7) normalize the coordinate values to a bounded range. This stabilizes gradients early in training when the SpatialSoftmax output might be noisy, and ensures the spatial features have comparable scale to the proprioception and goal vectors they are concatenated with.

> **Checkpoint:** Feed `(4, 32, 21, 21)` random features through SpatialSoftmax. Output shape should be `(4, 64)`. Values should be in `[-1, 1]`. Place a strong activation at position `(0, 0)` (top-left corner) and verify the output coordinates are near `(-1, -1)`.

9.3.4 ManipulationExtractor -- SB3-compatible wiring

SB3 needs a `BaseFeaturesExtractor` subclass that takes a dict observation space and produces a flat feature vector. ManipulationExtractor routes each key to the appropriate sub-encoder: image keys (detected by `is_image_space`) go through ManipulationCNN + SpatialSoftmax + LayerNorm + Tanh, while vector keys (proprioception, goals) pass through unchanged.

Listing 9.7: ManipulationExtractor -- routing dict observations

```
class ManipulationExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space, spatial_softmax=True,
                 num_filters=32, flat_features_dim=50):
        super().__init__(observation_space, features_dim=1)
        extractors, total = {}, 0
        for key, subspace in observation_space.spaces.items():
            if is_image_space(subspace):
                cnn = ManipulationCNN(subspace.shape[0], num_filters)
                with torch.no_grad():
                    feat = cnn(torch.zeros(1, *subspace.shape) / 255.0)
                    _, C, H, W = feat.shape
                if spatial_softmax:
                    extractors[key] = nn.Sequential(
                        cnn, SpatialSoftmax(H, W, C),
                        nn.LayerNorm(2 * C), nn.Tanh(),
                    )
                    total += 2 * C  # 64 for 32 filters
                else:
```

```
                    # Flatten + learned projection (DrQ-v2 trunk pattern)
                    flat_size = C * H * W
                    extractors[key] = nn.Sequential(
                        cnn, nn.Flatten(),
                        nn.Linear(flat_size, flat_features_dim),
                        nn.LayerNorm(flat_features_dim), nn.Tanh(),
                    )
                    total += flat_features_dim
            else:
                extractors[key] = nn.Flatten()
                total += gym.spaces.utils.flatdim(subspace)
        self.extractors = nn.ModuleDict(extractors)
        self._features_dim = total
```

For FetchPush with `spatial_softmax=True`, proprioception, and `goal_mode="both"`:

Component	Dimension
Pixels -> CNN -> SpatialSoftmax -> LN -> Tanh	64
Proprioception passthrough	10
achieved_goal passthrough	3
desired_goal passthrough	3
Total features_dim	**80**

9.3.5 Proprioception passthrough and sensor separation

The ManipulationExtractor does not force the CNN to learn about the robot's own body. Joint positions, velocities, and gripper width come from the `"proprioception"` key as a 10D vector that passes through unchanged, so the CNN only sees pixels and learns about the world (object positions, obstacles), not the self.

This is the sensor separation principle in practice: cameras observe the environment while joint encoders observe the robot. Mixing these signals in the CNN wastes network capacity on a problem that cheaper sensors already solve with perfect accuracy. In real robotic systems, proprioception comes from encoders sampling at kHz rates with sub-millimeter resolution -- no camera can compete with that for self-state measurement -- so the CNN should focus on what cameras are uniquely good at: perceiving the world beyond the robot's own body.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/manipulation_encoder.py --verify`. Expected: `[ALL PASS] Manipulation encoder verified`. Key checks: `features_dim = 80` (64 spatial + 10 proprio + 3 ag + 3 dg), SpatialSoftmax coordinates in $[-1, 1]$, ManipulationCNN output `(B, 32, 21, 21)`.

9.4 Build It: Data augmentation

DrQ (Kostrikov et al., 2020) regularizes the Q-function by augmenting pixel observations at sample time from the replay buffer. Each replay of a transition sees a different random crop, creating implicit regularization against Q-function overfitting to pixel-level details.

We build it here because it is a standard technique for pixel RL, used in DrQ, DrQ-v2, CURL, and many other visual RL methods. Whether it helps for manipulation Push with SpatialSoftmax is an empirical question we will answer in Section 9.7 -- the answer will surprise you. Building it now lets us run the controlled comparison later.

9.4.1 DrQ random shift

The augmentation is a pad-and-crop: pad the image by k pixels on all four sides using replicate padding (border pixels extended outward), then randomly crop back to the original size. With $k = 4$ on an 84x84 image, shifts of up to roughly 5% of the image size are possible.

Listing 9.8: RandomShiftAug -- pad-and-crop augmentation

```python
class RandomShiftAug(nn.Module):
    def __init__(self, pad: int = 4):
        super().__init__()
        self.pad = pad

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B, C, H, W = x.shape
        padded = F.pad(x, (self.pad,)*4, mode="replicate")
        crop_h = torch.randint(0, 2*self.pad + 1, (B,), device=x.device)
        crop_w = torch.randint(0, 2*self.pad + 1, (B,), device=x.device)
        # Index into padded tensor (each image gets independent shift)
        h_idx = torch.arange(H, device=x.device)[None] + crop_h[:, None]
        w_idx = torch.arange(W, device=x.device)[None] + crop_w[:, None]
        b_idx = torch.arange(B, device=x.device)[:, None, None, None]
        c_idx = torch.arange(C, device=x.device)[None, :, None, None]
        return padded[
            b_idx.expand(B,C,H,W), c_idx.expand(B,C,H,W),
            h_idx[:,None,:,None].expand(B,C,H,W),
            w_idx[:,None,None,:].expand(B,C,H,W),
        ]
```

Replicate padding is important because zero padding would create artificial dark borders that the CNN might learn to exploit as a position signal. Replicate padding instead extends the border pixels outward, maintaining the visual appearance of the scene edges, and the shift magnitudes are small enough (± 4 pixels on 84x84, roughly $\pm 5\%$) that the augmented images remain plausible views of the same scene.

> **Checkpoint:** Augment a batch of 8 images twice with the same input. The two outputs should differ (random shifts are independent). Augmenting a

constant-valued image should return the same constant (replicate padding of a constant is the same constant).

9.4.2 DrQ replay buffer

The augmentation happens at sample time, not store time. This means each replay of the same transition produces a different view -- the source of DrQ's regularization effect.

Listing 9.9: DrQDictReplayBuffer -- augment at sample time

```
class DrQDictReplayBuffer(DictReplayBuffer):
    def __init__(self, *args, aug_fn=None, image_key="pixels", **kwargs):
        super().__init__(*args, **kwargs)
        self.aug_fn = aug_fn
        self.image_key = image_key

    def _get_samples(self, batch_inds, env=None):
        samples = super()._get_samples(batch_inds, env)
        if self.aug_fn is not None:
            aug_obs = dict(samples.observations)
            aug_next = dict(samples.next_observations)
            aug_obs[self.image_key] = self.aug_fn(
                samples.observations[self.image_key])
            aug_next[self.image_key] = self.aug_fn(
                samples.next_observations[self.image_key])
            return DictReplayBufferSamples(
                observations=aug_obs, actions=samples.actions,
                next_observations=aug_next,
                dones=samples.dones, rewards=samples.rewards)
        return samples
```

Notice that obs and next_obs are augmented independently with different random shifts. This is by design in DrQ -- each sees a different view. We will revisit whether this is a good idea for SpatialSoftmax in Section 9.7.

9.4.3 HER + DrQ composed buffer

When using HER with pixel observations, we need goal relabeling AND augmentation. The HerDrQDictReplayBuffer composes both: HER first produces its merged batch (real + relabeled transitions with recomputed rewards), then we apply pixel augmentation to the result.

Listing 9.10: HerDrQDictReplayBuffer -- HER relabeling then DrQ augmentation

```
class HerDrQDictReplayBuffer(HerReplayBuffer):
    def __init__(self, *args, aug_fn=None, image_key="pixels", **kwargs):
        super().__init__(*args, **kwargs)
        self.aug_fn = aug_fn
```

```
        self.image_key = image_key

    def sample(self, batch_size, env=None):
        samples = super().sample(batch_size, env)  # HER relabeling
        if self.aug_fn and self.image_key in samples.observations:
            aug_obs = dict(samples.observations)
            aug_next = dict(samples.next_observations)
            aug_obs[self.image_key] = self.aug_fn(
                samples.observations[self.image_key])
            aug_next[self.image_key] = self.aug_fn(
                samples.next_observations[self.image_key])
            return DictReplayBufferSamples(
                observations=aug_obs, actions=samples.actions,
                next_observations=aug_next,
                dones=samples.dones, rewards=samples.rewards)
        return samples
```

We override `sample()` rather than `_get_samples()` because HER's internal flow calls `_get_real_samples()` + `_get_virtual_samples()` and merges them without going through `_get_samples()`. Goals are never augmented -- only pixel observations change -- which preserves the HER invariant: `compute_reward(achieved_goal, desired_goal)` must be consistent with the reward stored in the transition, and augmenting goal vectors would break this invariant.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/image_augmentation.py --verify`. Expected: `[ALL PASS] Image augmentation verified.` Key checks: augmented images have same shape as input, two augmentations of the same input differ, goals and rewards are unchanged by augmentation.

9.5 Build It: Gradient routing

This is the most subtle component in the pipeline -- and the one that makes the difference between 5% and 95% success. The code changes are small (about 15 lines of overrides), but the impact is decisive.

9.5.1 The problem: SB3's default puts encoder in the actor optimizer

When you create SB3's SAC with `share_features_extractor=True` (the default for shared encoders), SB3 puts the encoder parameters in the **actor's** optimizer. During critic training, SB3 wraps the encoder forward pass in `set_grad_enabled(False)` -- gradients do not flow through the encoder during TD loss computation.

This means the encoder learns ONLY from the actor's policy gradient. Early in training the policy is random -- it pushes in random directions and succeeds roughly 5% of the time by luck -- so the actor loss gradient is essentially noise: with a near-uniform random policy, the gradient signal says "all directions are equally bad," which gives the encoder nothing to learn from. The encoder therefore remains at its random initialization, the critic cannot distinguish states (since they all look the same through

a random encoder), and training never bootstraps. In our experiments, this default configuration produces 5-8% success, flat, for 2M+ steps -- regardless of whether you use NatureCNN or ManipulationCNN.

9.5.2 DrQ-v2 pattern: encoder in the critic optimizer

DrQ-v2 (Yarats et al., 2021) does the opposite: it places the encoder in the critic's optimizer, so the critic's TD loss directly asks "does this visual feature help predict future value?" -- a rich, stable learning signal. Meanwhile, the actor receives detached features, which prevents noisy policy gradients (especially at low success rates) from corrupting the encoder's representation.

The key insight is that the critic provides value-based supervision ("this state leads to high/low returns"), which is exactly what the encoder needs to learn useful spatial features. Even when success rate is near zero, the critic's TD error still provides signal: "this state where the gripper is near the puck has slightly higher Q than this state where the gripper is far away." That signal is weak, but it is directional -- it consistently pushes the encoder toward features that discriminate states by spatial proximity to goals. The actor's policy gradient, by contrast, is noisy and uninformative until the encoder already represents something useful -- a chicken-and-egg problem that the critic breaks.

The implementation requires one `forward()` override, one `.detach()` call, and an optimizer rewiring.

9.5.3 CriticEncoderCritic and CriticEncoderActor

Two small class overrides implement the DrQ-v2 pattern:

Listing 9.11: CriticEncoderCritic -- enable gradients through shared encoder

```python
class CriticEncoderCritic(ContinuousCritic):
    def forward(self, obs, actions):
        # CHANGED: always enable gradients through features_extractor
        # (SB3 default wraps this in set_grad_enabled(False))
        features = self.extract_features(obs, self.features_extractor)
        qvalue_input = th.cat([features, actions], dim=1)
        return tuple(q_net(qvalue_input) for q_net in self.q_networks)
```

Listing 9.12: CriticEncoderActor -- detach features before policy MLP

```python
class CriticEncoderActor(Actor):
    def get_action_dist_params(self, obs):
        features = self.extract_features(obs, self.features_extractor)
        features = features.detach()  # CHANGED: stop encoder gradient
        latent_pi = self.latent_pi(features)
        mean_actions = self.mu(latent_pi)
        log_std = self.log_std(latent_pi)
        log_std = th.clamp(log_std, LOG_STD_MIN, LOG_STD_MAX)
        return mean_actions, log_std, {}
```

The first override removes the gradient gate so that the TD loss updates the encoder, while the second adds `.detach()` so the policy loss does NOT update the encoder. Together, these two changes route all encoder learning through the critic.

9.5.4 DrQv2SACPolicy -- wiring the optimizers

The policy subclass ties everything together: one shared encoder, encoder parameters in the critic optimizer, encoder parameters excluded from the actor optimizer.

Listing 9.13: DrQv2SACPolicy -- encoder in critic optimizer

```
class DrQv2SACPolicy(SACPolicy):
    def _build(self, lr_schedule):
        self.actor = self.make_actor()       # CriticEncoderActor
        self.critic = self.make_critic(      # CriticEncoderCritic
            features_extractor=self.actor.features_extractor  # shared
        )
        # REVERSED from SB3 default:
        # encoder in CRITIC optimizer, excluded from actor optimizer
        encoder_ids = {id(p) for p in
                       self.actor.features_extractor.parameters()}
        actor_params = [p for p in self.actor.parameters()
                        if id(p) not in encoder_ids]
        self.actor.optimizer = self.optimizer_class(
            actor_params, lr=lr_schedule(1), **self.optimizer_kwargs)
        self.critic.optimizer = self.optimizer_class(
            list(self.critic.parameters()),  # includes shared encoder
            lr=lr_schedule(1), **self.optimizer_kwargs)
        # Target critic gets its own encoder copy (not shared)
        self.critic_target = self.make_critic(features_extractor=None)
        self.critic_target.load_state_dict(self.critic.state_dict())
```

The identity-based filtering (`id(p) not in encoder_ids`) avoids fragility if parameter naming conventions change across SB3 versions, since we match on Python object identity rather than parameter name strings. The target critic gets its own separate encoder that is updated via Polyak averaging ($\tau = 0.005$), as in standard SAC -- this is important because the target encoder must NOT be the same object as the online encoder, or Polyak averaging would be a no-op.

It is worth tracing the gradient flow carefully through this shared-encoder setup:

1. **Forward pass:** Computing the actor loss requires calling the critic's Q-networks: $L_{\text{actor}} = \alpha \log \pi(a|s) - Q(s, a)$. The critic uses the shared encoder to compute features for $Q(s, a)$.
2. **Backward pass:** When PyTorch runs `actor_loss.backward()`, it computes gradients for *every* parameter on the computational graph -- including the shared encoder, because the encoder sits between the pixels and the Q-value.
3. **Why no update happens:** PyTorch's `optimizer.step()` only updates parameters that are in `optimizer.param_groups`, and the actor optimizer

does not contain encoder parameters (we filtered them out). So even though `.backward()` writes gradients into the encoder's `.grad` tensors, `actor_optimizer.step()` ignores them entirely, and the encoder is only updated when `critic_optimizer.step()` runs.

4. **The `CriticEncoderActor` detach:** Our actor wrapper detaches features before the policy MLP as a safety measure, preventing encoder gradients from being *computed* during the actor loss backward pass. This is a belt-and-suspenders approach: the optimizer filtering (step 3) is sufficient on its own, but the detach avoids wasting compute on gradients that would be ignored anyway.

The net effect: the encoder learns from Bellman error (critic loss) only, not from the actor's policy gradient. This is the DrQ-v2 design -- the encoder should learn *state features* from TD error, not learn to *fool the critic* via the actor.

Checkpoint: Run `bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --verify`. Expected: `[ALL PASS] DrQ-v2 SAC policy verified`. Key checks:

- 0 encoder params in actor optimizer, all encoder params in critic optimizer
- After `critic.backward()`: encoder params have non-zero gradients
- After `actor.backward()`: encoder params have NO gradients (features detached)
- Actor and critic share the same encoder instance (`is` check passes)
- Target critic has its own encoder (separate instance)
- Save/load round-trip: predictions match within 1e-5 tolerance

9.6 Bridge: From scratch to SB3

You have now built 13 components across five lab files: a pixel wrapper, a replay buffer, two CNN encoders, SpatialSoftmax, an SB3-compatible feature extractor, DrQ augmentation, two DrQ replay buffers, and three gradient routing overrides. These are the same components that SB3 uses when you launch a pixel training run, so we can verify the full pipeline with three commands:

```
bash docker/dev.sh python scripts/labs/pixel_wrapper.py --verify
bash docker/dev.sh python scripts/labs/manipulation_encoder.py --verify
bash docker/dev.sh python scripts/labs/image_augmentation.py --verify
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --verify
```

All four should print `[ALL PASS]`. Then run the bridging proof to verify that our from-scratch components produce identical results to what SB3 uses internally:

```
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --bridge
```

This feeds the same pixel observation through our ManipulationExtractor and SB3's features_extractor, comparing output tensors. Expected: `max_diff < 1e-6` -- the implementations are numerically identical.

Next, run the gradient probe to see how SB3's default compares to our override:

```
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --probe
```

This prints the encoder parameter membership in the actor and critic optimizers for both SB3's default `SACPolicy` and our `DrQv2SACPolicy`, so you should see encoder parameters in the actor optimizer only (SB3 default) versus encoder parameters in the critic optimizer only (our override).

Here is how the Build It components map to what you see in TensorBoard during training:

TensorBoard metric	Build It component	What it measures
`train/critic_loss`	CriticEncoderCritic (Listing 9.11)	TD error through sha
`train/actor_loss`	CriticEncoderActor (Listing 9.12)	Policy loss on detach
`train/ent_coef`	SAC automatic tuning (Ch3)	Entropy temperature
`rollout/success_rate`	Full pipeline: wrapper + encoder + routing + HER	End-to-end task perf

Before moving to the full investigation, you can see the pipeline produce actual (short) learning curves on CPU:

```
bash docker/dev.sh python scripts/ch09_pixel_push.py demo --seed 0
```

This runs roughly 10 minutes on CPU with a small buffer and fewer steps. It will not reach the hockey-stick -- the demo is too short for that -- but you will see the critic loss decline during Phase 1, confirming that the pipeline is wired correctly and the encoder receives gradients.

What SB3 adds on top of our components: vectorized environment rollout (`SubprocVecEnv` for parallel pixel rendering), optimized replay sampling with proper episode boundary handling, TensorBoard logging, and checkpoint management. These are engineering concerns, not learning concerns -- the math is what you built.

9.7 Run It: The five-step investigation

This is where the components meet reality. We run five experiments, each adding one piece, and watch the success rate. The first two fail, the third succeeds (eventually), and the fourth shows that a standard technique (DrQ) actually hurts. Each failure teaches a transferable debugging principle.

Step 0: NatureCNN baseline (pixels are not drop-in)

Start with the obvious approach: take the working SAC + HER pipeline from Chapter 5 and replace the 25D state vector with 84x84 pixels. Use SB3's default NatureCNN encoder.

```
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --total-steps 2000000
```

Result: 5-8% success, flat, for 2M+ steps -- indistinguishable from a random policy.

The algorithm is proven to work at 89% from state vectors, and the only change is the observation modality, so something about the visual processing pipeline is fundamentally wrong.

Look at the spatial progression from Section 9.3.1: NatureCNN's stride-4 first layer crushes an 84x84 image down to 20x20. The puck, which starts at roughly 5 pixels wide, becomes roughly 1 pixel. The gripper-puck spatial relationship -- the entire signal the policy needs -- is destroyed in the first convolutional layer.

Principle: Pixels are not a drop-in replacement for state vectors. The encoder architecture must match the task's spatial requirements.

Step 1: Architecture fix (ManipCNN + SpatialSoftmax + proprioception)

Replace NatureCNN with the components from Sections 9.3.2-9.3.5: ManipulationCNN (gentle 3x3 stride-2 downsampling), SpatialSoftmax (extract "where" coordinates, not "what" features), and proprioception passthrough (10D robot state alongside pixel features). Same pipeline, better encoder -- the CNN can now represent precise spatial relationships between 5-pixel objects.

Result: Still 5-8% flat. Architecture is necessary but not sufficient.

This is the more subtle failure. The encoder CAN represent the right information -- ManipulationCNN preserves the puck at roughly 3 pixels through all four layers, and SpatialSoftmax extracts precise (x, y) coordinates from those activations -- but CAN represent and DOES represent are different things. The network has the capacity; the question is whether it is learning.

Checking where the encoder's gradients come from reveals the problem. In SB3's default configuration, the encoder sits in the actor optimizer, and the critic disables gradients through the encoder during TD updates, so the encoder learns only from the actor's policy gradient -- which is noisy and weak early in training because the policy is near-random. A random policy generates the gradient signal "all directions are equally bad," which gives the encoder nothing useful to learn from.

Principle: Architecture determines what a network CAN represent. Training determines what it DOES represent. We fixed the capacity; now we need to fix the learning signal.

Step 2: Gradient routing fix (critic-encoder) -- the breakthrough

Add the 15 lines from Section 9.5: `CriticEncoderCritic`, `CriticEncoderActor`, and `DrQv2SACPolicy`. Move the encoder into the critic optimizer. Detach features before the actor.

```
------------------------------------------------------------
EXPERIMENT CARD: SAC + HER + Pixel Pipeline on FetchPush-v4
------------------------------------------------------------
Algorithm:    SAC + HER (critic-encoder gradient routing,
              ManipulationCNN + SpatialSoftmax, no DrQ)
Environment:  FetchPush-v4 (pixel observations, sparse reward)
```

```
Fast path:    5,000,000 steps, seed 0
Time:         ~40 hours (GPU); not feasible on CPU for full run
              (Build It --verify: < 2 min CPU; --demo: ~10 min CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --no-drq --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 5000000 \
    --checkpoint-freq 500000

Checkpoint track (skip training):
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.zip

Expected artifacts:
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.zip
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.meta.json
  results/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0_eval.json
  runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed0/

Success criteria (fast path):
  success_rate >= 0.90 (at 5M steps)
  hockey-stick inflection visible in TensorBoard by ~2.5M steps
  critic_loss non-monotonic trajectory (decline -> rise -> decline)

Full multi-seed results: see REPRODUCE IT at end of chapter.
------------------------------------------------------------
```

Result: Flat at 6% for 2M steps -- you might think this failed too, but it did not. At 2.2M steps a slow upward trend appears; by 2.5M, success hits 25-34%; by 3.5M it crosses 70%; and by 4.4M it reaches 95%.

This is the **hockey-stick learning curve**: a long flat phase where the encoder is learning spatial structure from the critic's TD signal, followed by a rapid climb once the representation becomes good enough for the policy to exploit. The flat phase is not failure -- it is the representation learning overhead that pixel RL imposes. Section 9.8 explains the three mechanisms behind this curve.

The difference between Step 1 and Step 2 is 15 lines of gradient routing code -- the architecture is identical. The only change is WHERE the encoder's learning signal comes from: the critic's TD loss (rich, stable, directional even at low success rates) versus the actor's policy gradient (noisy, uninformative when the policy is near-random). This is the decisive choice of the chapter.

Principle: Where gradients flow matters as much as what architecture you use. The encoder needs value-based supervision from the critic, not noisy policy gradients from the actor.

Step 3: Reading the training curve (the patience tax)

Before adding more components, we pause to understand what just happened. Step 2's curve is not a smooth ascent -- it has three distinct phases, each with a characteristic loss signature. Learning to read this signature helps you decide when to be patient and when to intervene. Section 9.8 unpacks the three-phase loss signature in detail.

The headline: pixel RL needs 2-4x the training budget of state-based RL. State-based Push reached 89% at 2M steps, while pixel Push reached 95% at 4.4M steps -- a 2.2x overhead. If your stop rules are calibrated from state-based experience ("kill the run if no progress at 2M steps"), it helps to recalibrate for pixel RL, because otherwise runs get killed during Phase 1 before the hockey-stick has a chance to appear.

Principle: The representation learning phase is an unavoidable overhead. Rising losses during the hockey-stick are good news, not bad. Always read loss curves alongside success rate.

Step 4: DrQ ablation -- augmentation versus representation

You might wonder: we never added DrQ data augmentation, and DrQ is the standard technique for pixel RL (Kostrikov et al., 2020). Would adding it help?

Run Step 2's exact configuration with one change -- DrQ augmentation enabled (omit the `--no-drq` flag) -- while keeping all other hyperparameters (buffer size, HER strength, encoder, gradient routing) identical to Step 2:

```
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --critic-encoder --buffer-size 500000 \
  --her-n-sampled-goal 8 --total-steps 2000000
```

Result: 3% success, flat at 1.54M steps. DrQ makes things worse. (By comparison, Step 2 showed a clear upward trend by 1.4M steps with the same hyperparameters minus DrQ.)

Here is why. DrQ shifts images by +/-4 pixels via pad-and-crop, which after the CNN (84 -> 21 spatial resolution) becomes +/-1 pixel on the 21x21 feature map. SpatialSoftmax converts this to +/-0.10 in the [-1, 1] coordinate space, and -- critically -- DrQ augments `obs` and `next_obs` independently with different random shifts, doubling the noise in Bellman targets.

The gripper-puck distance signal in SpatialSoftmax coordinates is roughly 0.25-0.50 units, so with +/-0.10 noise on both `obs` and `next_obs` (independent), the noise-to-signal ratio reaches 40-80%. The TD target becomes unreliable, and the critic cannot learn the value structure it needs to train the encoder.

Principle: Data augmentation is not universally good. Choose your representation, then choose your augmentation. SpatialSoftmax extracts precise spatial coordinates; DrQ injects spatial noise. They are fundamentally incompatible.

Investigation summary

Figure 9.3 summarizes all five experimental steps.

Five-step investigation summary showing success rate for each configuration: Step 0 NatureCNN at 5%, Step 1 ManipCNN+SS at 5%, Step 2 plus critic-encoder at 95% after hockey-stick, Step 3 interprets the training curve, Step 4 plus DrQ at 3%

Figure 9.3: The five-step investigation. Steps 0 and 1 fail because the architecture is wrong (stride-4) and gradient routing is wrong (encoder in actor optimizer). Step 2 succeeds after the representation learning phase completes. Step 3 interprets the training curve. Step 4 shows that DrQ augmentation is harmful when combined with SpatialSoftmax. The breakthrough is gradient routing (Step 2), not architecture (Step 1) or augmentation (Step 4). (Generated by `bash docker/dev.sh python scripts/ch09_pixel_push.py plot-investigation`.)

Step	What changed	Success rate	Lesson
Full-state control	None (25D vectors)	89% at 2M	Pipeline validat
0: NatureCNN	Pixels with default CNN	5% flat	Stride-4 destro
1: ManipCNN + SS + proprio	Better encoder	5% flat	Architecture is
2: + critic-encoder	Gradient routing fix	95% at 4.4M	WHERE gradie
3: Training curve	Interpret Step 2's loss signature	(see Section 9.8)	Rising losses =
4: + DrQ	Data augmentation	3% flat	Augmentation

9.8 Reading the training curve

The Step 2 training curve is not a smooth ascent. It has three distinct phases, each with a characteristic loss signature. Learning to read this signature helps you tell the difference between a genuinely stuck run and one that just needs more time ("the critic loss is declining, the representation is warming up -- give it another 2M steps").

The three-phase loss signature

Figure 9.4 shows the three phases annotated on the actual training curve from Step 2.

Three-phase loss signature showing success rate, critic loss, and actor loss over training steps, with Phase 1 (flat success, declining critic loss), Phase 2 (rising success, rising losses), and Phase 3 (saturating success, declining losses) annotated

Figure 9.4: The three-phase loss signature from Step 2. Phase 1: the critic memorizes uniform failure. Phase 2: real value learning begins -- losses rise because the problem becomes harder for the critic. Phase 3: the value function converges to an accurate model. (Generated from TensorBoard logs at `runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed0/`.)

Phase	Steps	success_rate	critic_loss	actor_loss	What
1: Flat	0-2.2M	3-7%	Declining to 0.07	~0	Critic
2: Hockey-stick	2.2-3.5M	10-90%	Rising to 0.3+	Rising to 1.0+	Encod
3: Convergence	3.5M+	95%+	Declining to 0.15-0.35	Negative (-0.2 to -0.7)	Value

The counterintuitive lesson: **rising losses during Phase 2 are good news.** In supervised learning, rising loss means the model is getting worse. In sparse RL, rising critic loss means the critic has moved past the trivial solution (predict constant $Q = -18.5$ for all states) and is now trying to learn which states actually lead to success. That is a harder prediction problem, so the loss rises -- but it is a productive rise. The success rate climbing alongside the loss confirms this.

Phase 1's declining loss with flat success is the diagnostic red flag: it looks healthy on a loss plot, but the critic is not learning value structure -- it is memorizing the fact that all trajectories fail. If you only watch loss curves without checking success rate, Phase 1 looks indistinguishable from Phase 3, which is why we always read loss curves alongside success rate. The loss value alone is ambiguous.

The actor loss going negative in Phase 3 is a SAC-specific convergence signal. The actor loss is $L_{\text{actor}} = \alpha \log \pi(a|s) - Q(s,a)$. When Q is high (near 0, meaning "success is likely"), the Q-value term dominates the entropy penalty $\alpha \log \pi$, making the total loss negative. This means the policy is confidently selecting high-value actions -- a sign of convergence, not divergence.

Sidebar: Why the hockey-stick?

The flat phase followed by rapid improvement is not accidental. Three mechanisms interact to produce it.

1. Value propagation bottleneck. The critic learns Q-values through Bellman backups: $Q(s_t, a_t|g) \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a'|g)$. Each backup propagates value information one step outward from states where the agent has succeeded. HER seeds this process by relabeling nearby goals, but the "value wavefront" still grows one backup step at a time. Laidlaw et al. (2024) formalize this as the effective horizon k^* -- the minimum number of backup steps before greedy actions become near-optimal. Sample complexity is exponential in k^*, creating a sharp threshold between "not enough training" and "enough."

2. Geometric phase transition. Test goals are sampled uniformly over the table surface. The agent's "competence region" -- the set of goals it can push to -- starts small and grows as the value wavefront expands. In 2D, the overlap between a growing competence region and the fixed test goal distribution scales quadratically with the competence radius. When the radius is small, almost no test goals are reachable. Once it crosses a critical threshold, many become reachable at once. This creates the sharp inflection from 5% to 30%+.

3. Positive feedback loop. Once real test goals are reached, successful episodes provide higher-quality training signal than HER's relabeled goals (no distribution mismatch from goal substitution). Better signal drives the policy to explore further, reaching more goals, generating more signal. This converts the linear wavefront expansion into super-linear growth -- the steep part of the hockey-stick.

The full picture is assembled from three independent theoretical results (Laidlaw et al., 2024 on effective horizon; Wang & Isola, 2022 on quasimetric Q-

functions; Huang et al., 2025 on difficulty spectrum dynamics), not derived from a single theorem. No closed-form expression for the inflection point exists. The practical lesson: if your HER training curve is flat at 1M steps but critic loss is declining, the wavefront may not have reached the test goal distribution yet. Our Step 2 went from 6% at 2M to 95% at 4.4M.

The patience tax

Pixel RL needs 2-4x the training budget of state-based RL. State-based Push reached 89% at 2M steps, while pixel Push reached 95% at 4.4M steps -- a 2.2x overhead. This overhead is the representation learning phase: the encoder must learn useful spatial features before the policy can exploit them, and there is no shortcut.

We find it helpful to recalibrate stop rules for pixel RL. State-based intuitions ("no progress at 2M means it is broken") can lead to terminating runs during Phase 1, before the hockey-stick has a chance to emerge.

> **Tip:** For pixel RL with sparse rewards, set your initial training budget to at least 4x the state-based budget. Monitor critic loss: if it is declining during the flat phase, the representation is warming up. If critic loss is flat AND success is flat for 3M+ steps, something is likely wrong -- check the "What Can Go Wrong" table.

9.9 What Can Go Wrong

Pixel RL has more failure modes than state-based RL because the visual pipeline adds a large surface area for bugs. This table covers the failures we encountered and the ones readers are most likely to hit.

Symptom	Likely cause
Flat at 5% after 3M+ steps	Missing `--criti`
OOM during training	Pixel replay buff
Flat at 5% WITH `--critic-encoder` after 2M steps	Normal pre-hock
`critic_loss` declining monotonically for 3M+ steps, success still flat	Critic memorizin
DrQ + SpatialSoftmax stuck at 3%	Augmentation-r
Very slow FPS (< 15 fps)	Multiple Docker
TensorBoard shows mixed/confusing curves	Log contaminat
`--probe` shows encoder in actor optimizer	Using SB3's def
Hockey-stick at 2.2M but convergence stalls at 50-60%	Buffer too small
`RuntimeError: Unable to sample before end of first episode` on resume	`learning_star`

The first row is the most common failure mode. We watched it happen multiple times during development. Everything looks correct -- ManipulationCNN, SpatialSoftmax, proprioception, HER, large buffer -- but the encoder learns nothing because it is in the wrong optimizer. One flag fixes it.

9.10 Summary

This chapter added a visual observation pipeline to the SAC + HER stack from Chapters 4-5 and achieved 95%+ success on FetchPush from raw 84x84 pixels -- matching the state-based performance at the cost of 2.2x more training steps.

What you built (13 components across 5 lab files):

- `render_and_resize`: camera capture to CHW tensor
- `PixelObservationWrapper`: frame stacking, proprioception passthrough, goal modes
- `PixelReplayBuffer`: uint8 storage, float32 at sample time
- `NatureCNN` (the "wrong" encoder): stride-4 destroys 5-pixel objects
- `ManipulationCNN` (the "right" encoder): 3x3 stride-2 preserves spatial information
- `SpatialSoftmax`: extracts "where" coordinates, not "what" features
- `ManipulationExtractor`: routes dict observations for SB3 compatibility
- `RandomShiftAug`: DrQ pad-and-crop augmentation
- `DrQDictReplayBuffer` and `HerDrQDictReplayBuffer`: augment at sample time
- `CriticEncoderCritic` and `CriticEncoderActor`: gradient routing overrides
- `DrQv2SACPolicy`: encoder in critic optimizer, detached actor features

Three transferable principles:

1. **Architecture must match the task.** NatureCNN's stride-4 was designed for Atari sprites, not 5-pixel manipulation objects, so ManipulationCNN's stride-2 with SpatialSoftmax is needed to preserve spatial precision.
2. **Gradient routing is decisive.** The encoder needs the critic's value-based supervision, not the actor's noisy policy gradient -- 15 lines of code and the difference between 5% and 95%.
3. **Augmentation must match representation.** DrQ's random shift corrupts SpatialSoftmax's precise coordinates, so choosing the representation first and then selecting compatible augmentation is essential.

The compositional insight: This chapter did not invent a new algorithm; it composed SAC (Ch4) + HER (Ch5) + a visual pipeline (this chapter) and discovered that the critical missing piece was not a new loss function or a clever trick, but the routing of gradients through the encoder. The components from earlier chapters transferred directly -- the innovation was in wiring them correctly for pixel observations.

Looking ahead: We achieved 95% from pixels, but it took 4.4M steps and roughly 40 hours of GPU time per seed, because the representation learning phase -- the long flat period before the hockey-stick -- is an unavoidable overhead when learning visual features from scratch. Chapter 10 explores whether pre-trained visual encoders and world models can reduce this tax, and what happens when the gap between simulation and reality becomes the bottleneck.

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
```

```
The results and pretrained checkpoints in this chapter
come from these runs:

  for seed in 0 1 2; do
    bash docker/dev.sh python scripts/ch09_pixel_push.py train \
      --seed $seed --critic-encoder --no-drq --buffer-size 500000 \
      --her-n-sampled-goal 8 --total-steps 5000000 \
      --checkpoint-freq 500000
  done

Hardware:     NVIDIA GPU with >= 60 GB system RAM
              (tested on DGX; any modern GPU works, times will vary)
Time:         ~40 hours per seed at ~30 fps (GPU; not feasible on
              CPU -- use the checkpoint track instead)
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}.zip
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}.meta.json
  results/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}_eval.json
  runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed{0,1,2}/

Results summary (what we got -- seed 0; seeds 1-2 pending):
  success_rate: 0.95  (seed 0, 100 episodes)
  hockey_stick_onset: ~2.2M steps
  90%_success: ~3.5M steps
  training_time: ~40h per seed
  Multi-seed variance will be reported after seeds 1 and 2 complete.
  We expect similar results but cannot claim +/- bounds from one seed.

Comparison runs (failure baselines for the investigation):
  # Step 0: NatureCNN baseline
  bash docker/dev.sh python scripts/ch09_pixel_push.py train \
    --seed 0 --total-steps 2000000

  # Step 4: DrQ ablation (shows augmentation-representation conflict)
  bash docker/dev.sh python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 2000000

If your numbers differ by more than ~10% (hockey-stick not visible
by 3M, or final success below 80%), check the "What Can Go Wrong"
section above.

The pretrained checkpoints are available in the book's
companion repository for readers using the checkpoint track.
------------------------------------------------------------
```

Exercises

1. (Verify) Run the full-state control baseline.

Confirm that the Ch9 script wiring is correct independently of pixel processing:

```
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --full-state --total-steps 2000000
```

Expected: success_rate >= 0.85 at 2M steps, matching Ch5's state Push results. If this fails, the problem is in the script wiring, not the visual pipeline. Always validate the non-pixel path first.

2. (Tweak) HER relabeling strength.

Change `--her-n-sampled-goal` from 8 to 4 (the Ch5 default). Run for 3M steps with the winning config (`--critic-encoder --no-drq --buffer-size 500000`). Does the hockey-stick onset shift later? What about the final success rate?

Expected: later inflection (more steps needed to build the value wavefront with fewer relabeled transitions). Final success may be similar but convergence is slower. Quantify: at what step does success first exceed 20% with HER-4 vs HER-8?

3. (Explore) SpatialSoftmax ablation.

Run with `--no-spatial-softmax` (flatten + linear instead of SpatialSoftmax). Does the agent still learn? The flatten pathway produces 50D features instead of 64D spatial coordinates. What does this tell you about whether SpatialSoftmax is a necessary component or a helpful inductive bias?

Expected: likely still learns (ManipulationCNN + critic-encoder may be sufficient), but possibly slower or with lower final success. The result tells you whether "where not what" is critical or supplementary.

4. (Challenge) Test DrQ with flat features.

Run WITH DrQ but WITHOUT SpatialSoftmax (`--no-spatial-softmax`, without `--no-drq`). DrQ was designed for flat CNN features, not spatial coordinates. Does removing SpatialSoftmax make DrQ useful again?

Expected: if DrQ helps with flat features, the conclusion is "DrQ and SpatialSoftmax are separately good ideas but fundamentally incompatible." If DrQ still hurts, the conclusion is "DrQ's random shift is harmful for manipulation regardless of feature type."

5. (Challenge) Measure gradient magnitudes.

Read the `verify_gradient_flow()` function in `scripts/labs/drqv2_sac_policy.py`. Modify it to measure the actual gradient magnitude (L2 norm) flowing through the encoder during critic versus actor training. How much larger is the critic gradient? This quantifies the argument from Section 9.5 that the critic provides a richer learning signal than the actor.

Expected: critic gradient magnitude should be substantially larger than actor gradient (before the detach). Report the ratio as a concrete number -- this is the quantitative evidence for why critic-encoder routing matters.