Robotics Reinforcement Learning in Action

Build reproducible goal-conditioned manipulation agents

Vlad Prytula

2026-02-17

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 makes sure that does not happen to you silently. By the end, you will have a working environment that you *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 qualitatively different from traditional software bugs. 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.”

The practical consequence for you: unless you can *verify* that your environment, your rendering stack, your training loop, and your evaluation protocol all work correctly, you have no way to tell whether a training failure means ”the algorithm can't solve this” or ”something is broken in my setup.”

This chapter systematically eliminates the second possibility. When a training run fails after completing this chapter, you will know that the *infrastructure* is sound and the

problem lies in the algorithm, the hyperparameters, or the task itself -- which are 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 -- but you do not need the math background. 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.

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.

The Fetch environments form a natural difficulty ladder that we will climb throughout the book:

Environment	Task	Difficulty	Book chapters
`FetchReachDense-v4`	Move end-effector to target	Easiest	Ch1-4
`FetchReach-v4`	Same, but sparse reward	Moderate	Ch5
`FetchPush-v4`	Push block to target position	Hard	Ch5
`FetchPickAndPlace-v4`	Pick up block, place at target	Hardest	Ch6

All four environments share the same interface, the same observation structure, and the same action space. What changes is the task complexity: reaching requires only arm control, pushing adds object interaction, and pick-and-place adds the coordination of grasping and releasing. By keeping the interface constant and increasing the task difficulty, we isolate what matters: the algorithm's ability to learn, not the plumbing.

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). Each component is in [-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. 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 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 (the "no vibes" rule)

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_eval.json`
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." Checkpoints are files with known paths and machine-readable metadata. 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 without provenance is not a result -- it is an anecdote.

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 call this the "no vibes" rule: if you cannot point to a file that contains the number, the number does not exist.

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. This is not a convenience choice -- it is a reproducibility choice.

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 difference is expected. MuJoCo physics runs on CPU regardless of platform; on Linux, the GPU handles neural network forward and backward passes in microseconds, while on Mac both physics and neural networks compete for CPU time. This is fine for development, debugging, and running the early chapters. For serious training runs (Chapter 5 onward, where you may need 500,000 to 3,000,000 environment steps), use a GPU-equipped machine.

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 skills are diagnostic tools you will use every time 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
```

```python
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. Typical workspace bounds for the Fetch arm are 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.

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 is not just a convenience function -- it 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. The rewards match -- this is the *critical invariant*. This is the *critical invariant* for the entire book: `compute_reward(achieved_goal, desired_goal, info)` must equal the reward from `env.step()`. Every chapter that uses HER depends on this.

Why is this so important? 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 the relationship between the three things we have inspected:

1. **Observations** tell the policy where it is and where it should be
2. **Rewards** give the policy a training signal (dense: how close? sparse: success or failure?)
3. **Success** is the binary metric we ultimately care about

The reward drives learning; 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

These are not just warm-up exercises. They connect 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)

Success criteria:
  smoke_frame.png exists, non-empty, shows Fetch robot
  ppo_smoke.zip exists, loadable by PPO.load()
  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.

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 not long enough to learn a good policy -- it is long enough to verify that the entire training pipeline (environment interaction, gradient computation, parameter updates, checkpoint saving) works end-to-end.

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 two 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, optimizer state, and environment metadata. 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.

Do not evaluate this checkpoint 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. Learning starts in Chapter 3.

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 GPU availability (Test 1) -- without a GPU, training runs 10-20x slower.

When something fails, diagnose in order. If rendering fails, check Test 2 first -- maybe MuJoCo itself cannot initialize. If training is unexpectedly slow, check Test 1 -- maybe CUDA is not available. The `all` subcommand runs the tests sequentially and the output makes it clear which test failed.

What ”proof of life” means

After these four tests pass, you know:

- The container has access to the GPU (or is correctly falling back to CPU)
- MuJoCo and Gymnasium-Robotics are installed and functional
- Headless rendering produces valid images
- The full training loop (env -> policy -> training -> checkpoint) executes without error

Together with the Build It checks from section 1.5, you also know:

- Observations have the expected dictionary structure and shapes
- `compute_reward` matches `env.step()` (the critical invariant)
- The success signal correctly reflects goal distance

This is what ”alive” means: the environment can produce valid results. It does not yet mean the environment produces *good* results -- that is what the rest of the book is for.

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, not vibes in your head), and the willingness to inspect the environment directly rather than trusting that ”it probably works.”

You do *not* yet know whether training produces good policies. The PPO smoke test ran for only 50,000 steps -- enough to verify the loop, not enough to learn. You do not yet know 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?), or how to tell whether a flat reward curve means ”broken” or ”needs more time.”

That is what the rest of the book is for. 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

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())

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 both artifacts (`smoke_frame.png` and `ppo_smoke.zip`) 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”? Without answering these questions, you cannot debug training failures or 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. The observation shapes you document here determine the network architecture. 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?

Without understanding what the agent sees and what rewards mean, you cannot answer this question. 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. You trained for 10 hours, but you never checked whether the environment's outputs make sense for the algorithm you chose.

This is not hypothetical. In our experience, environment misunderstandings are the single most common source of wasted compute in robotics RL. Not bad hyperparameters, not wrong algorithms -- wrong assumptions about what the environment is actually doing.

Three questions you cannot answer without anatomy

Here are three questions that come up in every training failure. Try to answer them without inspecting 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. You might spend days trying to make HER work on an environment that does not support it, never realizing the problem is structural, not algorithmic.

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 succ

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

There is a positive way to frame this too. 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:

- **Checking action bounds** requires knowing the action space shape and range (section 2.4)
- **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 -- dense rewards are negative distances, sparse rewards are 0 or -1 (section 2.5)
- **Checking the `compute_reward` invariant** requires knowing the API signature and how it relates to `env.step()` (section 2.5)

These are not abstract skills. In Chapter 3, when PPO training stalls, the first thing you will do is check whether the reward is moving. In Chapter 5, when HER does not improve performance, the first thing you will do is check whether `compute_reward` handles relabeled goals correctly. The anatomy you learn 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 0.4 to 1.1
```

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. 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.

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. It sees 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, $\phi(s)$ = gripper position. For Push, $\phi(s)$ = 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:` x roughly [1.05, 1.55], y roughly [0.40, 1.10], z roughly [0.42, 0.60]

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:

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.

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 displacement 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

The sparse reward uses 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, -1 for failure. Not +1 for success. Not -1/+1. The reward is always non-positive. This 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. A random policy averages about -15 to -25 per episode. 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.

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)`.

This is not paranoid caution. Different versions of gymnasium-robotics have had bugs affecting reward computation, and API changes have altered the signature of `compute_reward`. Running these checks 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. 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). 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, step by step:

1. **Take a step** to get a non-trivial achieved goal (the gripper moved somewhere).
2. **Sample 10 random goals** from the goal space -- positions the environment never intended as targets.
3. **Call `compute_reward`** with the actual achieved goal and each random goal.
4. **Verify** 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. This is why relabeling is 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: when `achieved_goal` changes from tracking the gripper (Reach) to tracking the object (Push, PickAndPlace), the goal-achievement mapping ϕ changes. For Reach, $\phi(s)$ = gripper position. For Push, $\phi(s)$ = object position. The environment handles this transparently -- 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` performs the same checks -- the same three-way reward comparison, the same relabeling test, the same structure verification -- but in an automated pipeline that produces JSON artifacts.

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
```

100 steps, 5 relabeled goals per step = 500 total reward checks. 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: never trust a pipeline you have not verified by hand.

2.9 Run It: The inspection pipeline

Now we run the full production inspection. The script `scripts/ch01_env_anatomy.py` automates everything we have done by hand, producing machine-readable JSON artifacts.

```
------------------------------------------------------------
EXPERIMENT CARD: Fetch Environment Inspection
------------------------------------------------------------
Algorithm:    None (inspection and verification only)
Environment:  FetchReachDense-v4, FetchReach-v4, FetchPush-v4

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
------------------------------------------------------------
```

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. This confirms something we mentioned in section 2.2: episodes are truncated at 50 steps, never terminated early. 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
------------------------------------------------------------
This chapter does not train any policies. The verification
commands below confirm that your environment installation
is correct and that all inspection outputs match expected
values.

  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 produced:
  results/ch01_env_describe.json
  results/ch01_random_metrics.json

Expected inspection outputs:
  describe:
    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]
    distance_threshold == 0.05
    reward_type == "dense"

  reward-check:
    "OK: reward checks passed" (500 steps, 3 random goals
    per step, atol=1e-6, zero mismatches)
```

```
  random-episodes (FetchReachDense-v4, 10 episodes):
    success_rate: 0.0-0.1
    return_mean: approx -15 to -25
    ep_len_mean: 50
    final_distance_mean: approx 0.05-0.15

Lab verification:
  bash docker/dev.sh python scripts/labs/env_anatomy.py --verify
    All checks pass in < 1 min on CPU.

  bash docker/dev.sh python scripts/labs/env_anatomy.py --bridge
    Manual compute_reward matches env.step() reward on 100 steps.
    Dense reward matches -np.linalg.norm(ag - dg) within atol=1e-10.
    Sparse reward matches threshold formula within atol=1e-10.
    Relabeled goals: compute_reward produces correct reward for
    arbitrary goals (simulating HER relabeling).

If any check fails, see "What Can Go Wrong" in this chapter.
------------------------------------------------------------
```

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 (x roughly 1.05-1.55, y roughly 0.40-1.10, z roughly 0.42-0.60). 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 -15 to -25; 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 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. 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. This validates 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. That off-policy machinery is what Chapter 5 (HER) requires when we tackle sparse rewards.

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: how do you take a gradient of this? The expectation depends on θ in a complicated way -- θ determines the policy, which determines the actions, which determines the states visited, which 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). 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. Sometimes they are way off, and we make bad updates.

Big updates break everything. Suppose we estimate that some action is great ($A \gg 0$) and crank up its probability. But if our estimate was wrong, we have committed to a bad action. Worse, the new policy visits different states, making our old advantage estimates invalid. The whole thing can spiral into catastrophic collapse.

This is not hypothetical. Training curves that look promising can crash to zero and never recover. In our experience, unclipped policy gradient on Fetch tasks fails roughly half the time -- you get lucky with some seeds and unlucky with others. That is not a reliable foundation to build on.

PPO's solution: constrained updates

PPO's key idea: 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, 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:

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.

Why this helps PPO:

- **Every action provides signal.** ”You got 2cm closer” or ”you drifted 1cm away” -- the algorithm always has gradient information.
- **Exploration is not a bottleneck.** Even random actions produce useful data, because every distance tells you something.
- **Learning problems are algorithm problems.** If PPO fails on dense Reach, the issue is in your implementation, not in insufficient exploration. Dense rewards decouple the exploration problem from the learning problem.

Compare to sparse rewards ($R = 0$ if success, -1 otherwise): 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 three practical questions (from Chapter 1):

1. **Can this be solved?** Yes -- the policy network has 16 inputs and 4 outputs, with ~5,600 parameters. 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.
2. **Is the solution reliable?** We expect yes -- the task is low-dimensional, the reward is smooth, and the goal distribution is bounded. Different seeds should converge to qualitatively similar policies (move toward the goal), even if the exact parameters differ.
3. **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 are more interesting for harder tasks. For dense Reach, the answers are reassuring -- which is precisely the point. We start here because failure would be unambiguous evidence of a bug.

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? Three reasons:

1. **Different objectives.** The actor maximizes expected return (wants to find good actions). The critic minimizes prediction error (wants accurate value estimates). These gradients can conflict -- improving one may hurt the other.
2. **Different output types.** The actor outputs a probability distribution (mean and variance for continuous actions). The critic outputs a single scalar. Forcing these through the same final layers creates unnecessary coupling.
3. **Stability.** The critic's value estimates feed into advantages, which train the actor. If actor updates destabilize the critic, advantages become noisy, which destabilizes the actor further -- a vicious cycle.

In practice, implementations often share early layers (a ”backbone”) with separate final layers (”heads”). This captures 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. This gives 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. Each pass uses minibatches 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 is where PPO's fundamental limitation appears.

> **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). You use these transitions for `n_epochs` gradient steps, then discard all of them -- even though many transitions contain useful information that could improve the policy further. This is sample-inefficient: 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 -- 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. We recommend against tuning hyperparameters until you have verified that the baseline works with these values.

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.

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

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

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

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

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

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

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

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. The critic outputs a single scalar -- the estimated value $V(x)$. Both share the backbone features but have independent output layers.

The network is small by design -- Fetch tasks use MLPs, not 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 $\gamma\lambda$. The `(1 - done)` term resets the accumulation at episode boundaries -- when an episode ends, future advantages from a different episode should not bleed in.

The **returns** are computed as `advantages + values`. These 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:

```
# 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. 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:

```
    # 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)
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.

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 $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$:

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

```
# 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. 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:

- **Advantage normalization.** Before computing the policy loss, we subtract the mean and divide by the standard deviation of the advantages. This is standard practice -- it reduces the sensitivity to reward scale and makes the gradient magnitudes more consistent across batches.
- **Gradient clipping.** We clip the gradient norm at `max_grad_norm=0.5`. This prevents a single bad batch from causing an explosively large update. This is separate from the PPO ratio clipping -- gradient clipping limits the step size in parameter space, while PPO clipping limits the step size in probability space.
- **The entropy term.** 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:

```
# 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):

```
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.

> **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. Before we use SB3 for the real training run, let's 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`. Both use 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). This means 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 important. When you see `train/clip_fraction = 0.15` in TensorBoard, you know what that means -- 15% of the probability ratios exceeded the $[0.8, 1.2]$ clip range, and those updates were constrained. That number came from the same calculation as the `clip_fraction` in our `compute_ppo_loss`. You are not reading opaque metrics from a black box; you are reading quantities you have implemented and verified.

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 artifacts. It takes 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, these are the same quantities you implemented in the Build It sections. `train/value_loss` is the output of your `compute_value_loss`. `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 -- 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.

What does this look like in practice? The robot arm starts at a default position. At each timestep, the policy sees the current gripper position (in `achieved_goal`) and the target position (in `desired_goal`), and 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 -- 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:

1. **Environment is configured correctly** -- observations and actions have the right shapes and semantics
2. **Network architecture works** -- `MultiInputPolicy` correctly processes dictionary observations
3. **GPU acceleration works** -- training completes in reasonable time
4. **Evaluation protocol is sound** -- you can load checkpoints and run deterministic rollouts
5. **Metrics are computed correctly** -- success rate matches what you observe

This is the pipeline baseline. Every future chapter builds on this infrastructure. 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 normal -- RL training on Fetch is CPU-bound on MuJoCo simulation, not GPU-bound on neural network operations. This is expected behavior, not a bug. 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:

- **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. This means 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 modern GPU (tested on NVIDIA GB10; CPU works but ~6x slower)
Time:          ~8 min per seed (GPU), ~45 min per seed (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.