Contents

1 Chapter 1: The Anatomy of Goal-Conditioned Fetch Environments

1.1 Abstract

This chapter develops a precise understanding of what a reinforcement learning agent ”perceives” when interacting with a Gymnasium-Robotics Fetch environment. We formalize the observation space, action space, and reward function--not as implementation details to be glossed over, but as mathematical structures whose properties determine what algorithms can and cannot accomplish.

The central result of this chapter is that Fetch environments implement a specific mathematical structure: the *goal-conditioned Markov Decision Process*. This structure--characterized by observation dictionaries with explicit goal representations and reward functions that can be evaluated for arbitrary goals--is not merely a design choice; it is the mathematical substrate

that enables Hindsight Experience Replay and related techniques. A researcher who does not understand this structure cannot use those techniques correctly.

1.2 Part 0: The Practical Context

1.2.1 0.1 Where We Are

You have completed Chapter 0. You now have:

- A working Docker container with GPU access
- MuJoCo physics simulation running headlessly
- A verified training loop that produces checkpoints

You are running on a DGX cluster (or similar GPU workstation). Your environment is reproducible: anyone with the same Docker image and code can replicate your results.

How to Execute Commands. All commands in this curriculum run through the Docker wrapper:

```
bash docker/dev.sh <your-command>
```

For example:

```
# Run a specific script
bash docker/dev.sh python scripts/ch01_env_anatomy.py describe

# Start an interactive shell inside the container
bash docker/dev.sh
```

When you run `bash docker/dev.sh python some_script.py`, the following happens:

1. **Container launches** with GPU access (`--gpus all`) and your repository mounted at `/workspace`
2. **Virtual environment** is created (first run) or activated in `.venv/`
3. **Dependencies** from `requirements.txt` are installed (cached by hash--reinstalls only when requirements change)
4. **Your command executes** inside this isolated environment
5. **Container exits** when the command completes (or stays open for interactive shells)

The script preserves your host user ID, so files created inside the container are owned by you, not root.

First Run. On first invocation, `dev.sh` builds the Docker image `robotics-rl:latest` from `docker/Dockerfile`. This takes several minutes but only happens once. Subsequent runs start in seconds.

1.2.2 0.2 What We Are Simulating

We are training neural network policies to control a **simulated robot** in a **physics engine**. Let us be precise about what this means.

The Physics Engine: MuJoCo. MuJoCo (Multi-Joint dynamics with Contact) is a physics simulator designed for robotics and biomechanics research. It computes:

- Rigid body dynamics (how objects move under forces)
- Contact forces (what happens when the robot touches objects)
- Joint constraints (how the robot's links connect)

MuJoCo runs the physics at 500Hz internally; the environment exposes control at 25Hz (every 20 simulation steps). When you call `env.step(action)`, MuJoCo simulates 20 timesteps of physics, then returns the resulting state.

Why Simulation? Reinforcement learning requires millions of trials. A real robot would:

- Break from repeated collisions
- Take months to collect enough data
- Pose safety hazards during random exploration

In simulation, we collect a million timesteps in minutes. Policies trained in simulation can later transfer to real hardware (sim-to-real transfer), though that is beyond this curriculum's scope.

The Simulated Robot: Fetch. The Fetch robot is a real mobile manipulator manufactured by Fetch Robotics (now part of Zebra Technologies), designed for warehouse automation and research. The MuJoCo model in Gymnasium-Robotics replicates its kinematics.

Real Fetch Robot	Simulated FetchReach
Fetch Robot *Source: Robots Guide*	FetchReach *Source: Gymnasium-Robotics*

Real Robot Specifications (from Wevolver and Robots Guide):

Component	Specification
Arm	7 degrees of freedom, 940mm reach, 6kg payload
Gripper	Parallel-jaw, 245N grip force, swappable
Joints	Harmonic drives + brushless motors, 14-bit encoders
Height	1.09m - 1.49m (telescoping spine)
Weight	113 kg
Compute	Intel i5, Ubuntu + ROS

What We Simulate (the subset relevant for our tasks):

Component	Simulation Detail
Arm	7 DOF, matches real kinematics
Gripper	Parallel-jaw, 2 fingers
Workspace	~1m reach from base
Control mode	Cartesian velocity (internal IK solver)
Physics rate	500Hz internal, 25Hz control interface

The simulation includes a table with objects (for Push/PickAndPlace tasks) and a red sphere marking the goal position. The arm is mounted on a fixed base (we do not simulate the mobile platform).

Further Reading:

- Gymnasium-Robotics Fetch documentation -- official environment docs with action/observation specs
- Original OpenAI Gym Robotics paper -- Plappert et al., "Multi-Goal Reinforcement Learning" (introduces these environments)
- Fetch Robotics product page -- real robot specifications

What the Agent Controls. The agent does not control joint torques directly. Instead, it outputs 4D Cartesian velocity commands:

- `(vx, vy, vz)`: Desired end-effector velocity in world frame
- `gripper`: Open (<0) or close (>0) command

An internal controller (part of the MuJoCo model) converts these Cartesian commands to joint torques. This simplification means the agent does not need to learn inverse kinematics--it just says ”move left” and the controller figures out which joints to actuate.

1.2.3 0.3 What We Are Doing

Before we write any training code, we must understand *exactly* what the robot perceives and what commands it accepts.

This is not academic pedantry. Consider what happens if you misunderstand the interface:

Misunderstanding	Consequence
Wrong observation shape	Network architecture mismatch, cryptic shape errors
Wrong action semantics	Policy learns to output nonsense commands
Wrong reward interpretation	Hyperparameters tuned for wrong scale
Missing goal structure	Cannot use HER, forced to use inefficient methods

Every hour spent understanding the environment saves ten hours debugging training failures.

1.2.4 0.4 The Four Fetch Tasks

The Gymnasium-Robotics package provides four manipulation tasks of increasing difficulty:

Task	Goal	Difficulty
FetchReach	Move end-effector to target position	Easiest--no object interaction
FetchPush	Push object to target position	Medium--requires contact
FetchPickAndPlace	Pick up object, place at target	Hard--requires grasping
FetchSlide	Slide object to distant target	Hardest--requires throwing motion

Each task has two reward variants:

- **Dense** (e.g., `FetchReachDense-v4`): Reward = negative distance to goal. Provides continuous feedback.
- **Sparse** (e.g., `FetchReach-v4`): Reward = 0 if goal reached, -1 otherwise. Binary feedback only.

1.2.5 0.5 Why Environment Anatomy Matters

The Fetch environments are not arbitrary. They implement a specific interface designed for **goal-conditioned learning**. Understanding this interface is essential because it enables a technique called **Hindsight Experience Replay (HER)** that we will use in Chapter 4.

Here is the key insight, explained simply:

The Problem with Sparse Rewards. Imagine you tell the robot ”reach position (0.5, 0.3, 0.2)” but it ends up at (0.6, 0.4, 0.3). With sparse rewards, this trajectory gets reward = -1 at every step (failure). The robot learns nothing useful--it just knows it failed.

The HER Solution. What if we could say: ”You failed to reach (0.5, 0.3, 0.2), but you successfully demonstrated how to reach (0.6, 0.4, 0.3)!” We relabel the trajectory with the goal the robot *actually* achieved, recompute the rewards (now it's a success!), and learn from that.

Why the Interface Matters. For this relabeling trick to work, the environment must provide three things:

1. **Separate goal information in observations.** The environment returns a dictionary with three keys:
   - `observation`: Robot state (joint positions, velocities)
   - `desired_goal`: Where we wanted to go (the target)
   - `achieved_goal`: Where we actually are (current position)

   Without this separation, we couldn't know what goal was ”achieved” by a trajectory.

2. **A `compute_reward()` function that accepts any goal.**

   ```
   reward = env.unwrapped.compute_reward(achieved_goal, any_goal, info)
   ```

   This lets us ask ”what would the reward have been if the goal were X?” without re-running the simulation. This is how we recompute rewards after relabeling.

3. **A geometric success threshold.** Success means `distance(achieved_goal, desired_goal) < 0.05` (5 centimeters). This concrete definition lets us determine success for any goal we choose to relabel with.

Bottom Line. These three interface features--dictionary observations, recomputable rewards, geometric success--are not arbitrary design choices. They are the mathematical substrate that makes HER possible. If any feature were missing, HER would not work.

If you treat the environment as a black box--feeding observations to a network and hoping it learns--you will waste weeks on avoidable failures. This chapter makes the interface explicit so you can use HER correctly in Chapter 4 and debug intelligently when things go wrong.

1.2.6 0.6 What This Chapter Produces

By the end of this chapter, you will have:

1. **JSON schemas** documenting observation and action spaces exactly
2. **Verified reward consistency** between `env.step()` and `compute_reward()`
3. **Random baseline metrics** establishing the performance floor
4. **Complete understanding** of why Fetch environments enable HER

These are not optional artifacts. They are the foundation on which all subsequent training rests.

1.3 Part I: The Problem

1.3.1 1.1 WHY: The Goal-Conditioning Paradigm

Consider the problem of learning a robotic manipulation policy. A naive formulation might be:

Naive Problem. *Find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes expected cumulative reward on a manipulation task.*

This formulation is inadequate for two reasons.

First, it conflates the *policy* with the *task*. A policy trained to reach position $(0.5, 0.3, 0.2)$ cannot generalize to position $(0.6, 0.4, 0.3)$ without retraining. If we want a policy that can reach arbitrary positions, we need the policy to accept the target position as input.

Second, the naive formulation provides no mechanism for learning from failure. If the task is "reach position g" and the agent fails to reach g, the episode provides no useful learning signal--the agent knows it failed, but not how to improve. This is particularly problematic with sparse rewards, where the agent receives no feedback until it succeeds.

The goal-conditioned formulation resolves both issues:

Problem (Goal-Conditioned Policy Learning). *Let $(\mathcal{S}, \mathcal{A}, \mathcal{G}, P, R, \gamma)$ be a goal-conditioned MDP. Find a policy $\pi : \mathcal{S} \times \mathcal{G} \to \Delta(\mathcal{A})$ that maximizes:*

$$J(\pi) = \mathbb{E}_{g \sim p(g)} \mathbb{E}_{\tau \sim \pi(\cdot|\cdot, g)} \left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1}, g) \right]$$

The policy now takes both state and goal as input, enabling generalization across goals. Moreover, the explicit goal structure enables *relabeling*: a trajectory that fails to reach goal g can be relabeled as a successful trajectory for whatever goal it actually reached, manufacturing learning signal from failure.

This chapter examines how the Fetch environments implement this formulation.

1.3.2 1.2 The Specific Questions of This Chapter

We seek to answer four questions:

Q1 (Observation Structure). What is the precise structure of the observation returned by `env.step()`? What are the shapes, ranges, and semantics of each component?

Q2 (Action Semantics). What do actions mean? Are they joint torques, joint velocities, Cartesian velocities, or something else? What are their ranges and how are they clipped?

Q3 (Reward Computation). How is the reward computed? Is it dense (continuous feedback) or sparse (binary success/failure)? Can the reward be recomputed for arbitrary goals?

Q4 (Goal Achievement). How does the environment determine success? What is the `achieved_goal` and how does it relate to the `desired_goal`?

These questions are not merely technical curiosities. The answers determine what algorithms are applicable, what hyperparameters are sensible, and what performance is achievable.

1.4 Part II: The Mathematical Framework

1.4.1 2.1 HOW: The Goal-Conditioned MDP Formalism

We begin with precise definitions.

Definition (Goal-Conditioned MDP). *A goal-conditioned Markov Decision Process is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{G}, P, R, \phi, \gamma)$ where:*

- *$\mathcal{S}$ is the state space*
- *$\mathcal{A}$ is the action space*
- *$\mathcal{G}$ is the goal space*
- *$P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the transition kernel*

- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{G} \to \mathbb{R}$ *is the reward function*
- $\phi : \mathcal{S} \to \mathcal{G}$ *is the goal-achievement mapping*
- $\gamma \in [0, 1)$ *is the discount factor*

The goal-achievement mapping ϕ is crucial: it extracts from each state the "achieved goal"--the outcome that state represents. For a reaching task, $\phi(s)$ might be the end-effector position; for a pushing task, it might be the object position.

Definition (Goal-Conditioned Observation). *A goal-conditioned observation is a tuple* $o = (\bar{s}, g_a, g_d)$ *where:*

- $\bar{s} \in \mathbb{R}^{d_s}$ *is the proprioceptive state (joint positions, velocities, etc.)*
- $g_a = \phi(s) \in \mathcal{G}$ *is the achieved goal*
- $g_d \in \mathcal{G}$ *is the desired goal*

The separation of achieved and desired goals is what enables relabeling. Given a trajectory with observations $(o_0, \ldots, o_T)$, we can substitute any $g' \in \mathcal{G}$ for the desired goal and recompute rewards as $R(s_t, a_t, s_{t+1}, g')$.

Proposition (Reward Recomputation). *If the reward function* R *depends on the goal only through the distance* $\|g_a - g_d\|$*, then the reward for any achieved goal* g_a *and any hypothetical desired goal* g' *can be computed without re-simulating the trajectory:*

$$R(s, a, s', g') = f(\|\phi(s') - g'\|)$$

for some function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}$*.*

This proposition is the mathematical foundation of Hindsight Experience Replay. It requires that the environment expose both ϕ (the goal-achievement mapping) and R (the reward function) in a form that allows evaluation for arbitrary goals.

1.4.2 2.2 The Fetch Environment as Goal-Conditioned MDP

The Gymnasium-Robotics Fetch environments implement the goal-conditioned MDP structure as follows:

State Space $\mathcal{S}$. The underlying state includes joint positions, joint velocities, gripper state, and (for manipulation tasks) object positions and velocities. The proprioceptive portion exposed to the agent has dimension $d_s = 10$ for reaching tasks and $d_s = 25$ for manipulation tasks.

Action Space $\mathcal{A}$. Actions are 4-dimensional: $a = (v_x, v_y, v_z, g) \in [-1, 1]^4$. The first three components specify Cartesian velocity commands for the end-effector; the fourth controls the gripper (positive = close, negative = open).

Goal Space $\mathcal{G}$. Goals are 3-dimensional Cartesian positions: $g \in \mathbb{R}^3$. For reaching tasks, the goal is the target end-effector position; for manipulation tasks, it is the target object position.

Reward Function R. Two variants exist:

- *Dense:* $R(s, a, s', g) = -\|\phi(s') - g\|_2$
- *Sparse:* $R(s, a, s', g) = \begin{cases} 0 & \text{if } \|\phi(s') - g\|_2 < \epsilon \\ -1 & \text{otherwise} \end{cases}$

where $\epsilon = 0.05$ is the success threshold.

Goal-Achievement Mapping ϕ. For reaching: $\phi(s) =$ end-effector position. For manipulation: $\phi(s) =$ object position.

1.4.3 2.3 The Dictionary Observation Structure

Fetch environments return observations as Python dictionaries with three keys:

```
{
    'observation': np.ndarray,   # shape (d_s,), proprioceptive state
    'achieved_goal': np.ndarray, # shape (3,), φ(s)
    'desired_goal': np.ndarray   # shape (3,), g
}
```

Remark (Why Dictionaries?). *The dictionary structure serves two purposes. First, it makes the goal-conditioned structure explicit--the achieved and desired goals are not buried in a flat observation vector but clearly labeled. Second, it enables automatic handling by Stable Baselines 3's `MultiInputPolicy`, which processes each key through a separate encoder before concatenation.*

1.4.4 2.4 The compute_reward Function

Fetch environments expose a `compute_reward` method that allows reward evaluation for arbitrary goals:

```
reward = env.unwrapped.compute_reward(achieved_goal, desired_goal, info)
```

This method takes batches of achieved goals and desired goals and returns the corresponding rewards. It is the API through which HER implementations relabel trajectories.

Critical Invariant. *For any transition (s, a, s', g), the following must hold:*

$$\texttt{env.step(a)[1]} = \texttt{env.unwrapped.compute\_reward}(\phi(s'), g, \texttt{info})$$

where the left side is the reward returned by `step()` and the right side is the reward computed by `compute_reward()`. Violation of this invariant would cause HER to learn from corrupted reward labels.

This invariant is verified empirically in Section 3.3.

1.5 Part III: The Implementation

1.5.1 3.1 WHAT: Empirical Verification of the Mathematical Structure

We now verify that the Fetch environments implement the goal-conditioned MDP structure as described. All verifications are implemented in `scripts/ch01_env_anatomy.py`.

1.5.1.1 3.1.1 Enumerating Available Environments

First, we enumerate the available Fetch environments:

```
bash docker/dev.sh python scripts/ch01_env_anatomy.py list-envs
```

Expected Output. Environment IDs including:

- `FetchReach-v4` (sparse reaching)
- `FetchReachDense-v4` (dense reaching)
- `FetchPush-v4` (sparse pushing)
- `FetchPushDense-v4` (dense pushing)
- `FetchPickAndPlace-v4` (sparse pick-and-place)
- `FetchPickAndPlaceDense-v4` (dense pick-and-place)

- `FetchSlide-v4` (sparse sliding)
- `FetchSlideDense-v4` (dense sliding)

The naming convention encodes task and reward type: environments without "Dense" use sparse rewards; those with "Dense" use distance-based rewards.

1.5.1.2 3.1.2 Describing Observation and Action Spaces

```
bash docker/dev.sh python scripts/ch01_env_anatomy.py describe --json-out results/ch01_env
```

This command produces a JSON file documenting the precise structure of observations and actions.

Expected Structure (FetchReach-v4):

```
{
  "action_space": {
    "shape": [4],
    "low": [-1, -1, -1, -1],
    "high": [1, 1, 1, 1]
  },
  "observation_space": {
    "observation": {"shape": [10], "low": [...], "high": [...]},
    "achieved_goal": {"shape": [3], "low": [...], "high": [...]},
    "desired_goal": {"shape": [3], "low": [...], "high": [...]}
  }
}
```

Interpretation. The observation dictionary has three components as predicted by the theory. The action space is 4-dimensional and bounded by $[-1, 1]$. The goal spaces are 3-dimensional Cartesian coordinates.

1.5.2 3.2 Action Semantics

The 4-dimensional action vector is interpreted as follows:

Index	Semantic	Range	Effect
0	v_x	$[-1, 1]$	End-effector velocity in x
1	v_y	$[-1, 1]$	End-effector velocity in y
2	v_z	$[-1, 1]$	End-effector velocity in z
3	gripper	$[-1, 1]$	Gripper command (> 0 = close, < 0 = open)

Remark (Cartesian vs. Joint Control). *The Fetch environments use Cartesian velocity control, not joint torque control. This is a significant simplification: the agent does not need to learn inverse kinematics. The Cartesian commands are converted to joint commands by an internal controller that is part of the environment dynamics.*

Remark (Action Scaling). *Actions are scaled by a factor before being applied. The exact scaling depends on the environment configuration. The agent outputs values in $[-1, 1]$; the environment scales these to physical units.*

1.5.3 3.3 Reward Consistency Verification

We verify the critical invariant that `env.step()` rewards match `compute_reward()` rewards:

```
bash docker/dev.sh python scripts/ch01_env_anatomy.py reward-check --n-steps 500
```

Expected Output. A message confirming that all 500 reward comparisons matched within numerical tolerance.

Failure Mode. If rewards do not match, HER will learn from incorrect reward labels. This would be a critical bug in either the environment or our understanding of its API.

Remark (Why This Check Matters). *The reward consistency check is not paranoid caution. Different versions of Gymnasium-Robotics have had bugs affecting reward computation. API changes have altered the signature of `compute_reward`. Running this check ensures that your specific installation behaves correctly.*

1.5.4 3.4 Random Baseline Metrics

Before training any agent, we establish baseline performance with a random policy:

```
bash docker/dev.sh python scripts/ch01_env_anatomy.py random-episodes --n-episodes 10 --js
```

Expected Output (FetchReachDense-v4):

- `success_rate`: ~0.0-0.1 (random flailing occasionally reaches the goal)
- `mean_return`: ~−15 to −25 (negative because rewards are negative distances)
- `mean_episode_length`: 50 (environments truncate at 50 steps)

Expected Output (FetchReach-v4, sparse):

- `success_rate`: ~0.0-0.05 (very unlikely to reach by chance)
- `mean_return`: ~−50 (constant −1 per step when not at goal)

These baselines establish the performance floor. Any trained agent that does not significantly exceed random performance is not learning.

1.6 Part IV: Theoretical Implications

1.6.1 4.1 Why the Dictionary Structure Enables HER

The dictionary observation structure is not arbitrary; it is the interface through which HER operates.

Theorem (HER Applicability). *Hindsight Experience Replay is applicable to an environment if and only if:*

1. *The observation includes an explicit `achieved_goal` $g_a = \phi(s)$*
2. *The reward function $R(s, a, s', g)$ can be evaluated for arbitrary goals g*
3. *The reward depends on the goal only through $\|g_a - g\|$*

The Fetch environments satisfy all three conditions by construction.

Corollary. *Standard (non-goal-conditioned) environments cannot use HER without modification. An environment that returns flat observations with no goal separation does not expose the structure HER requires.*

1.6.2 4.2 The Sparse vs. Dense Reward Trade-off

The choice between sparse and dense rewards involves a fundamental trade-off:

Dense Rewards: $R = -\|g_a - g_d\|$

- Pro: Provides gradient signal at every timestep
- Pro: Standard policy gradient methods (PPO) can learn effectively
- Con: May encourage undesirable behaviors (e.g., hovering near the goal without reaching it)
- Con: Reward shaping may not align with true task objective

Sparse Rewards: $R = \mathbf{1}[\|g_a - g_d\| < \epsilon] - 1$

- Pro: Clearly defined success criterion
- Pro: No reward shaping artifacts
- Con: No gradient signal until goal is reached
- Con: Requires HER or similar techniques for sample-efficient learning

Remark (When to Use Each). *For initial development and debugging, use dense rewards--they make it easier to verify that the pipeline is working. For final experiments, consider sparse rewards, which more accurately reflect the true task objective. HER bridges the gap by enabling sample-efficient learning even with sparse rewards.*

1.6.3 4.3 The Observation Dimension and Policy Architecture

The observation dimensions have implications for policy architecture:

Environment	`observation` dim	`achieved_goal` dim	`desired_goal` dim	Total
FetchReach	10	3	3	16
FetchPush	25	3	3	31
FetchPickAndPlace	25	3	3	31

Stable Baselines 3's `MultiInputPolicy` handles dictionary observations by:

1. Processing each key through a separate MLP encoder
2. Concatenating the encoded representations
3. Passing the concatenation through shared layers

This architecture allows the policy to learn separate representations for state and goal, which may improve generalization across goals.

1.7 Part V: Deliverables

Upon completion of this chapter, the following must exist:

D1. The file `results/ch01_env_describe.json` containing the observation and action space schema.

D2. The reward consistency check (`bash docker/dev.sh python scripts/ch01_env_anatomy.py reward-check`) must pass.

D3. The file `results/ch01_random_metrics.json` containing random baseline metrics.

D4. The reader must be able to answer:

- What is the dimension of the `observation` component for FetchReach? *(Answer: 10)*
- What does action index 3 control? *(Answer: gripper open/close)*
- What is the success threshold ϵ? *(Answer: 0.05)*
- Why can HER relabel trajectories? *(Answer: because `compute_reward` can evaluate arbitrary goals)*

A reader who cannot produce these deliverables or answer these questions has not completed the chapter.

1.8 Part VI: Connections

1.8.1 6.1 Connection to Chapter 0

This chapter assumes that the experimental environment from Chapter 0 is functional. All commands are executed inside the container via `docker/dev.sh`.

1.8.2 6.2 Connection to Chapter 2

Chapter 2 will train a PPO baseline on `FetchReachDense-v4`. The observation structure documented here determines the policy architecture. The random baseline metrics establish the performance floor against which the trained agent is compared.

1.8.3 6.3 Connection to Chapter 4

Chapter 4 will introduce HER for sparse-reward tasks. The `compute_reward` function verified here is the API through which HER relabels goals. The theoretical analysis of HER applicability in Section 4.1 explains why Fetch environments are suitable for HER and what properties an environment must have.

1.9 Appendix A: Observation Component Details

1.9.1 A.1 FetchReach Observation (10 dimensions)

Index	Semantic
0-2	Gripper position (x, y, z)
3-5	Gripper velocity $(\dot{x}, \dot{y}, \dot{z})$
6-7	Gripper finger positions
8-9	Gripper finger velocities

1.9.2 A.2 FetchPush/PickAndPlace Observation (25 dimensions)

Index	Semantic
0-2	Gripper position
3-5	Object position
6-8	Object relative position (object − gripper)
9-11	Gripper velocity
12-14	Object velocity
15-17	Object relative velocity
18-21	Object rotation (quaternion)
22-24	Object angular velocity

1.10 Appendix B: Formal Verification of HER Requirements

Requirement 1: Explicit achieved goal.

- Verified: Observations include `achieved_goal` key.
- Test: `'achieved_goal' in env.observation_space.spaces`

Requirement 2: Computable reward for arbitrary goals.

- Verified: `env.unwrapped.compute_reward(ag, dg, info)` accepts arbitrary goal arrays.
- Test: Call with random goals, verify no errors.

Requirement 3: Reward depends only on goal distance.

- Verified: Dense reward is $-\|g_a - g_d\|$; sparse is threshold on same.
- Test: Verify `compute_reward(ag, dg, {})` equals `-np.linalg.norm(ag - dg)` for dense.

All three requirements are verified by `scripts/ch01_env_anatomy.py reward-check`.

Next. With the environment anatomy understood, proceed to Chapter 2 to establish PPO baselines on dense Reach.