

## Contents

<b>1 Chapter 5: PickAndPlace -- From Push to Grasping</b>	<b>1</b>
1.1 WHY: What Makes PickAndPlace Harder . . . . .	2
1.1.1 5.1 Three Structural Differences from Push . . . . .	2
1.1.2 5.2 The Multi-Phase Control Problem . . . . .	2
1.1.3 5.3 The Air Goal Challenge . . . . .	3
1.1.4 5.4 Dense-First Debugging . . . . .	3
1.2 HOW: Transfer and Evaluation Methodology . . . . .	4
1.2.1 5.5 Starting from Proven Hyperparameters . . . . .	4
1.2.2 5.6 Definition: Goal Stratification . . . . .	5
1.2.3 5.7 Definition: Stress Evaluation . . . . .	5
1.2.4 5.8 Curriculum Learning (Conditional) . . . . .	6
1.3 BUILD IT: Curriculum Goal Wrapper (5.9) . . . . .	6
1.3.1 5.9.1 The Goal Sampling Mechanism . . . . .	7
1.3.2 5.9.2 CurriculumGoalWrapper . . . . .	7
1.3.3 5.9.3 Schedule Callback . . . . .	7
1.3.4 5.9.4 Integration with SB3 . . . . .	8
1.3.5 5.9.5 Verify: --verify Command . . . . .	8
1.3.6 5.9.6 Exercises . . . . .	9
1.4 WHAT: Experiments and Expected Results (Run It) . . . . .	9
1.4.1 5.10 Experiment 0: Dense Debug (~13 min) . . . . .	9
1.4.2 5.11 Experiment 1: SAC+HER Sparse PickAndPlace . . . . .	10
1.4.3 5.12 Stratified Evaluation . . . . .	11
1.4.4 5.13 Stress Evaluation . . . . .	11
1.4.5 5.14 Full Pipeline . . . . .	12
1.4.6 5.15 Results and Analysis . . . . .	12
1.4.7 5.16 CLI Parameter Reference . . . . .	13
1.4.8 Artifact Locations . . . . .	14
1.5 Summary . . . . .	14
1.5.1 Concepts Introduced in This Chapter . . . . .	14
1.5.2 Files Generated . . . . .	14
1.5.3 What Comes Next . . . . .	15

## 1 Chapter 5: PickAndPlace -- From Push to Grasping

**Week 5 Goal:** Transfer SAC+HER from Push to PickAndPlace, introducing dense-first debugging, stratified evaluation (air vs table goals), and stress testing.

---

In Chapter 4, we trained SAC+HER to push objects to sparse-reward goals with 99.4% success. Push involves a single contact behavior -- the gripper slides the object along the table, and at no point does it need to close its fingers. PickAndPlace changes this fundamentally: the robot must **grasp** the object, **lift** it off the table, **carry** it to the goal, and **place** it there. Some goals are on the table (similar to Push), but about half are in the air -- requiring the full grasp-lift-place sequence.

This creates a natural transfer experiment. Ch04 produced a winning hyperparameter configuration ( $\gamma = 0.95$ ,  $\text{ent\_coef} = 0.05$ ,  $n_{\text{sampled\_goal}} = 4$ ) through a 120-run sweep. Were those values Push-specific, or do they capture general properties of sparse goal-conditioned learning? If the same configuration works on a qualitatively harder task, that is strong evidence for the latter.

The question for this chapter: **Do our proven hyperparameters transfer to this harder task?** And when performance inevitably differs, what tools help us diagnose *where* and *why*?

This chapter introduces three diagnostic techniques -- dense-first debugging, goal stratification, and stress evaluation -- that help us answer "what went wrong" with numbers rather than guesswork. We also build a curriculum wrapper (the Build It lab) as a conditional fallback if direct transfer proves insufficient.

---

## 1.1 WHY: What Makes PickAndPlace Harder

### 1.1.1 5.1 Three Structural Differences from Push

PickAndPlace (FetchPickAndPlace-v4) differs from Push (FetchPush-v4) in three environment parameters that fundamentally change the task:

Parameter	Push	PickAndPlace	Effect
block_gripper	True	False	Gripper can close (grasp)
target_in_the_air	False	True	~50% of goals are elevated
gripper_extra_height	0.0	0.2	Gripper starts higher, can reach above table

These are not hyperparameter differences -- they change the **problem structure**. In Push, the gripper is always open (it cannot grasp), all goals are on the table, and the only skill required is pushing. PickAndPlace removes these constraints: the gripper can close, goals may be in the air, and the agent must learn to grasp, lift, carry, and place.

Note that the **action space** also changes in practice. While both environments have 4D Cartesian actions ( $dx$ ,  $dy$ ,  $dz$ , gripper), Push ignores the gripper dimension entirely (`block_gripper=True` clamps it). In PickAndPlace, the fourth action dimension controls finger position -- an entirely new degree of freedom the agent must learn to coordinate with the other three.

### 1.1.2 5.2 The Multi-Phase Control Problem

Reaching a goal in the air requires a sequence of coordinated behaviors:

- |                   |   |
|-------------------|---|
| Phase 1: Approach | - Move gripper above the object         |
| Phase 2: Descend  | - Lower gripper to object height        |
| Phase 3: Grasp    | - Close gripper fingers on the object   |
| Phase 4: Lift     | - Raise the grasped object              |
| Phase 5: Carry    | - Move horizontally to goal XY position |
| Phase 6: Place    | - Lower object to goal Z and release    |

Each phase has different action requirements. The agent must **discover** this sequence through trial and error -- we do not encode the phases explicitly.

Why is this hard for RL? Consider what needs to go right:

1. **Phase ordering is strict.** You cannot lift before grasping, or carry before lifting. An action that is correct in phase 4 (move up) is counterproductive in phase 2 (need to move down). The policy must condition on what phase it's in, which it infers entirely from the observation.
2. **Grasping is a precision skill.** The gripper must close at the right moment -- when the fingers straddle the object. Close too early (fingers above the object) and you miss.

Close too late (fingers past the object) and you push it away. This requires coordinating the gripper action with the position actions within a narrow spatial window.

3. **Sparse rewards provide no partial credit.** An agent that perfectly executes phases 1-4 but carries the object to the wrong position receives the same  $R = -1$  as an agent that does nothing. HER helps by relabeling: if the agent lifted the object but carried it to the wrong position, HER can retroactively say "pretend the goal was where you placed it."
4. **The behavior is temporally extended.** A successful pick-and-place takes most of the 50-step episode. Unlike Reach (which can succeed in  $\sim 10$  steps), there is little room for wasted actions.

### 1.1.3 5.3 The Air Goal Challenge

In FetchPickAndPlace-v4, approximately 50% of goals have their z-coordinate above the table surface. We call these **air goals**. The remaining  $\sim 50\%$  are **table goals** at the table height ( $z \approx 0.42\text{m}$ ).

**Definition (Goal Stratification).** We classify goals into two types based on their z-coordinate:

$$\text{type}(g) = \begin{cases} \text{air} & \text{if } g_z > z_{\text{table}} + 0.02 \\ \text{table} & \text{otherwise} \end{cases}$$

where  $z_{\text{table}} \approx 0.42\text{m}$  is the table surface height and  $0.02\text{m}$  is a small margin to avoid classifying table-surface goals as air.

This classification matters because the two goal types require **qualitatively different skills**. Table goals can sometimes be solved by pushing (similar to ch04's Push task), since the agent does not need to grasp -- sliding the object along the table surface to the goal position is sufficient. Air goals, by contrast, always require grasping and lifting because no pushing shortcut exists; the object must leave the table.

We expect air-goal success to lag table-goal success during training. An agent that has learned pushing but not grasping will score well on table goals and poorly on air goals. Reporting only the aggregate success rate would hide this important gap -- the agent might appear to have 50% success when it has actually mastered only one of the two sub-tasks.

Our dense-debug run (section 5.10) confirms this prediction empirically: even at 500k steps with dense rewards, table goals have mean return -9.78 while air goals have mean return -16.22 -- nearly 2x worse. The agent learns table-reaching behaviors first because they are similar to the simpler skills it discovers early in training.

### 1.1.4 5.4 Dense-First Debugging

Before committing to a 5M-step sparse training run ( $\sim 1\text{-}1.5$  hours), we validate the pipeline on FetchPickAndPlaceDense-v4 with 500k steps ( $\sim 13$  minutes).

**Why this matters:** Dense rewards provide continuous feedback proportional to goal distance. If the agent cannot learn *anything* with dense rewards, there is likely a bug in the pipeline -- not a hyperparameter issue. Dense-first debugging catches problems like wrong observation preprocessing, action clipping issues, HER misconfiguration, environment version mismatches, and rendering backend failures (which silently break physics).

**Why 500k steps, not 200k?** This is a lesson we learned the hard way. PickAndPlace is qualitatively harder than Reach or Push -- the agent must discover grasping, lifting, and placing.

Our first attempt used 200k steps and saw 0% eval success, which looked like a pipeline failure. But the training logs told a different story: reward had improved from -14.3 to -12.3 and rolling success reached 4%. The pipeline was fine -- the task genuinely needed more time.

At 500k steps we see enough learning signal to distinguish "pipeline works but task is hard" from "something is broken." The script uses a multi-level verdict rather than a simple pass/fail threshold:

- **PASS** (success  $\geq 20\%$ ): Pipeline validated, proceed confidently
- **LEARNING** (any success, or return above random baseline of  $\sim -15$ ): Pipeline works, PickAndPlace just needs more steps
- **MARGINAL** (return above worst-case -25 but no success): Some learning detected, may want to investigate
- **WARNING** (no improvement at all): Likely a pipeline bug -- fix before proceeding

If you see WARNING, fix the pipeline before proceeding. Dense debugging takes 13 minutes; discovering a bug after a 5M-step sparse run takes hours.

Our dense debug produced a LEARNING verdict: 0% eval success but mean return of -13.77, above the random baseline of  $\sim -15$ . The stratified data confirmed section 5.3's prediction -- table goals were nearly 2x closer to success than air goals, showing the agent learned pushing-like behavior before grasping. Full results and analysis are in section 5.10.

---

## 1.2 HOW: Transfer and Evaluation Methodology

### 1.2.1 5.5 Starting from Proven Hyperparameters

Chapter 4's 120-run sweep identified the winning configuration for sparse Push:

Parameter	Value	Source
gamma	0.95	Dominated all other factors (+11 pp vs 0.98)
ent_coef	0.05	Fixed prevents entropy collapse on sparse rewards
n_sampled_goal	4	Default, effective
learning_starts	1000	Early start helps on sparse tasks
batch_size	256	Default, works well

We start PickAndPlace with these exact values. The hypothesis is that these hyperparameters are not Push-specific -- they address general properties of sparse-reward goal-conditioned learning. Consider each in turn:  $\gamma = 0.95$  works because Fetch episodes are 50 steps, so the effective horizon  $T_{\text{eff}} = 1/(1-\gamma) = 20$  is long enough to propagate reward information across an episode but short enough to avoid vanishing gradients -- a property that depends on episode length, not task content. Fixed  $\text{ent\_coef} = 0.05$  prevents the entropy collapse that auto-tuning causes with sparse rewards (ch04, section 4.3), since sparse rewards create an uninformative loss landscape for the entropy coefficient where  $\alpha$  collapses to near-zero and kills exploration -- a problem that exists regardless of which Fetch task we train on. Finally, HER with future strategy and  $n_{\text{sampled\_goal}} = 4$  creates 5x data amplification, and the future strategy is particularly effective because PickAndPlace trajectories -- even failed ones -- visit many distinct positions that serve as valid relabeled goals.

The main change is `total_steps`: we increase from 2M (Push) to 5M (PickAndPlace), anticipating that the multi-phase control problem needs more experience. If Push required 2M steps for 99.4% success on a single-phase task, PickAndPlace's six phases should need at least 2-3x more.

### 1.2.2 5.6 Definition: Goal Stratification

Standard evaluation reports a single success rate across all episodes. For PickAndPlace, this conflates two fundamentally different sub-tasks.

**Motivating problem.** Our dense-debug run showed 0% aggregate success -- but the per-goal-type breakdown revealed the agent was nearly 2x closer to success on table goals (return -9.78) than air goals (return -16.22). Without stratification, we would have missed this signal entirely.

**Definition (Stratified Evaluation).** We run  $N$  evaluation episodes and partition them by goal type. For each partition, we compute separate metrics:

$$\text{SR}_{\text{air}} = \frac{|\{e : \text{type}(g_e) = \text{air} \wedge \text{success}(e)\}|}{|\{e : \text{type}(g_e) = \text{air}\}|}$$

$$\text{SR}_{\text{table}} = \frac{|\{e : \text{type}(g_e) = \text{table} \wedge \text{success}(e)\}|}{|\{e : \text{type}(g_e) = \text{table}\}|}$$

The **air gap** is  $\text{SR}_{\text{table}} - \text{SR}_{\text{air}}$ . A large air gap means the agent has learned pushing but not grasping. As the agent masters grasping, we expect the gap to narrow.

**Grounding example.** In our dense-debug data (50 episodes, 31 air / 19 table), stratification revealed:

Goal type	N	Mean return	Mean distance
Air	31	-16.22	0.324m
Table	19	-9.78	0.196m

The table goals have 40% shorter distance and 40% better return. Without stratification, we would see only the blended return of -13.77 and miss the fact that the agent is learning table-reaching behaviors first.

### 1.2.3 5.7 Definition: Stress Evaluation

Standard evaluation tests the policy under clean conditions -- the same physics and observations the agent saw during training. But real deployments introduce sensor noise, actuator imprecision, and small environmental changes. How much does performance degrade?

**Definition (Stress Evaluation).** Given a trained policy  $\pi(a|s)$ , stress evaluation modifies the evaluation loop:

1. **Observation noise:**  $\tilde{s}_t = s_t + \epsilon_s$ , where  $\epsilon_s \sim \mathcal{N}(0, \sigma_{\text{obs}}^2 I)$
2. **Action noise:**  $\tilde{a}_t = \text{clip}(a_t + \epsilon_a, a_{\text{low}}, a_{\text{high}})$ , where  $\epsilon_a \sim \mathcal{N}(0, \sigma_{\text{act}}^2 I)$

Default noise levels:  $\sigma_{\text{obs}} = 0.01$ ,  $\sigma_{\text{act}} = 0.05$ . Note: observation noise is applied only to the observation key of the dict observation, not to desired\_goal or achieved\_goal -- we are testing robustness to sensor noise, not changing the task.

**Why these noise levels?** The observation noise  $\sigma_{\text{obs}} = 0.01$  corresponds to roughly 1cm position error, which is realistic for camera-based state estimation. The action noise  $\sigma_{\text{act}} = 0.05$  corresponds to 5% of the action range, a reasonable model of actuator imprecision.

The **degradation** is  $\text{SR}_{\text{clean}} - \text{SR}_{\text{stress}}$ :

Degradation	Verdict	Interpretation
< 10%	Robust	Policy learned generalizable behavior
10-30%	Moderate	Sensitive but functional
> 30%	Fragile	Memorized precise trajectories

A fragile policy has memorized precise trajectories rather than learning robust behavior. We expect grasping to be more noise-sensitive than pushing: grasping requires precise finger-object alignment, while pushing tolerates sloppy contact. Stratified stress testing can reveal if air-goal success degrades more than table-goal success.

#### 1.2.4 5.8 Curriculum Learning (Conditional)

If sparse PickAndPlace success remains below 60% after 5M steps, we have a fallback: **curriculum learning**.

**Motivating problem.** The full PickAndPlace goal distribution includes goals that require the most complex behavior (grasping + lifting to a specific height). If the agent rarely encounters goals it can solve early in training, it collects few positive rewards and learning stalls. Curriculum learning addresses this by starting with goals the agent can already solve and gradually expanding the difficulty.

**Intuition.** Think of learning to juggle: you start with one ball before adding two and three. Curriculum learning applies the same principle -- give the agent easy goals first (nearby, on the table) so it experiences success, then gradually introduce harder goals (farther away, in the air) as it improves.

**Definition (Curriculum Learning).** Instead of sampling goals from the full distribution, we start with easy goals and gradually increase difficulty:

- **Difficulty 0:** Goals within 2cm of the object, all on the table (essentially a very short Push)
- **Difficulty 1:** Goals up to 15cm away, ~50% in the air (full distribution)

A schedule function  $d(t) : [0, T_{\text{train}}] \rightarrow [0, 1]$  maps training progress (timestep  $t$  out of  $T_{\text{train}}$  total training steps) to difficulty. We implement two modes:

1. **Linear:**  $d(t) = t/T_{\text{train}}$  -- difficulty increases steadily regardless of performance
2. **Success-gated:**  $d(t) = d(t - 1) + \Delta$  only when  $\text{SR}(t) > \theta$  over a sliding window -- difficulty advances only when the agent masters the current level

Curriculum learning is **not** in the default pipeline -- it is an option if the direct approach fails. We prefer testing the simpler approach first: if ch04's hyperparameters transfer directly, we don't need the added complexity. This follows a general principle: add complexity only when simpler approaches demonstrably fail, and keep the simpler approach as the baseline for comparison.

---

### 1.3 BUILD IT: Curriculum Goal Wrapper (5.9)

This section builds a curriculum wrapper from scratch. Even if we don't use it for training, it demonstrates a useful technique for goal-conditioned RL and illustrates how gym.Wrapper can modify environment behavior without touching environment internals.

### 1.3.1 5.9.1 The Goal Sampling Mechanism

In Gymnasium-Robotics Fetch environments, goals are sampled during `reset()`. The environment's internal `_sample_goal()` method determines the goal distribution. Our wrapper intercepts `reset()` and replaces the sampled goal with a difficulty-controlled one.

The wrapper does NOT modify environment dynamics -- only the goal distribution. Physics, rewards, and success criteria remain unchanged. This is important: it means any change in learning speed can be attributed to the goal distribution, not to a different reward function or different physics.

The design principle is **minimal intervention**: we override one method (`reset()`), replace one quantity (the goal), and ensure the rest of the environment behaves identically. This makes the wrapper composable with other wrappers (like SB3's `Monitor` and `DummyVecEnv`).

### 1.3.2 5.9.2 CurriculumGoalWrapper

The wrapper interpolates between easy and hard goal distributions based on a difficulty parameter  $d \in [0, 1]$ :

- **Goal range:**  $r(d) = r_{\min} + d \cdot (r_{\max} - r_{\min})$  where  $r_{\min} = 0.02\text{m}$  and  $r_{\max} = 0.15\text{m}$
- **Air probability:**  $p_{\text{air}}(d) = d \cdot p_{\max}$  where  $p_{\max} = 0.5$

At  $d = 0$ : goals are within 2cm and always on the table (trivial). At  $d = 1$ : goals span 15cm and 50% are in the air (full difficulty).

```
--8<-- "scripts/labs/curriculum_wrapper.py:curriculum_wrapper"
```

!!! lab "Checkpoint" Verify the wrapper at extreme difficulty levels:

```
```python
import gymnasium as gym
from scripts.labs.curriculum_wrapper import CurriculumGoalWrapper, _classify_goal

env = gym.make("FetchPickAndPlace-v4")
wrapped = CurriculumGoalWrapper(env, initial_difficulty=0.0)

# Easy: all goals on table, close to object
obs, _ = wrapped.reset()
goal_type = _classify_goal(obs["desired_goal"])
print(f"Easy goal type: {goal_type}" # "table"

wrapped.set_difficulty(1.0)
# Hard: goals can be in the air
n_air = sum(
    1 for _ in range(50)
    if _classify_goal(wrapped.reset()[0]["desired_goal"]) == "air"
)
print(f"Air goals at difficulty=1: {n_air}/50") # ~25
```
```

Expected: At `difficulty=0`, all goals are "table". At `difficulty=1`, roughly half are "air".

### 1.3.3 5.9.3 Schedule Callback

The callback advances difficulty during SB3 training. It implements the `__call__` protocol expected by SB3's `callback` parameter:

```
--8<-- "scripts/labs/curriculum_wrapper.py:curriculum_schedule"
!!! lab "Checkpoint" Verify linear scheduling:
```python
from scripts.labs.curriculum_wrapper import (
    CurriculumGoalWrapper, CurriculumScheduleCallback,
)

env = gym.make("FetchPickAndPlace-v4")
wrapped = CurriculumGoalWrapper(env, initial_difficulty=0.0)
cb = CurriculumScheduleCallback(wrapped, total_timesteps=1000, mode="linear",
                                check_freq=100, verbose=0)

# Simulate training
for _ in range(1000):
    cb({}, {})

print(f"Final difficulty: {wrapped.difficulty:.2f}") # ~1.0
```

```

### 1.3.4 5.9.4 Integration with SB3

```
--8<-- "scripts/labs/curriculum_wrapper.py:integration_example"
```

To use curriculum with `make_vec_env`, pass a factory function:

```
!!! lab "Illustrative" "python from stable_baselines3.common.env_util import make_vec_env
def make_curriculum_fn():
    env = gym.make("FetchPickAndPlace-v4")
    return CurriculumGoalWrapper(env, initial_difficulty=0.0)

vec_env = make_vec_env(make_curriculum_fn, n_envs=8)
```

```

Note a subtlety: when using vectorized environments, each sub-environment gets its own `CurriculumGoalWrapper` instance. The schedule callback should reference one of them (or a shared difficulty value) to advance difficulty. The chapter script handles this by accessing the wrapper through `vec_env.envs[0]`.

### 1.3.5 5.9.5 Verify: --verify Command

Run the full verification suite:

```
bash docker/dev.sh python scripts/labs/curriculum_wrapper.py --verify
```

Expected output:

```
=====
Curriculum Wrapper -- Verification
=====
Verifying wrapper basics...
[PASS] Wrapper basics OK

Verifying easy goals (difficulty=0.0)...
Air goals: 0% (expected: 0%)
Mean XY distance: ~0.01m (expected: <0.02m)
```

```

[PASS] Easy goals OK

Verifying hard goals (difficulty=1.0)...
  Air goals: ~50% (expected: ~50%)
  Mean XY distance: ~0.08m (expected: wider spread)
[PASS] Hard goals OK

Verifying schedule callback (linear mode)...
  Final difficulty: 1.00 (expected: ~1.0)
[PASS] Schedule callback OK

```

---

[ALL PASS] Curriculum wrapper verified

---

### 1.3.6 5.9.6 Exercises

#### **Exercise: Compare Linear vs Success-Gated Schedules**

Modify the demo to compare both modes. Linear schedules advance regardless of agent performance. Success-gated schedules wait until the agent masters each level.

*Question:* When would success-gated be preferable? When would linear be better? Consider: what happens if the agent is stuck at difficulty 0.3 with a linear schedule? What happens with a success-gated schedule if the threshold is set too high?

#### **Exercise: Custom Difficulty Dimensions**

The current wrapper controls goal range and air probability. What other difficulty dimensions could be useful?

*Hint:* Consider object size, friction, initial object position randomization, or episode length. Which of these can be controlled through gym.Wrapper and which would require modifying the environment source?

---

## 1.4 WHAT: Experiments and Expected Results (Run It)

All experiments run through Docker. Times measured on DGX with RTX A100, ~540-600 fps throughput.

### 1.4.1 5.10 Experiment 0: Dense Debug (~13 min)

```
bash docker/dev.sh python scripts/ch05_pick_and_place.py dense-debug
```

This trains SAC+HER on FetchPickAndPlaceDense-v4 for 500k steps and evaluates with stratified metrics. It validates that the pipeline works before committing to long sparse runs.

#### **What to look for:**

- **PASS** (success  $\geq$  20%): Pipeline validated, proceed to sparse training
- **LEARNING** (some success or return above random): Pipeline works, task is just harder than Push
- **WARNING** (no learning at all): Check for bugs, wrong env version, or rendering issues

In all cases, table-goal return should be better (less negative) than air-goal return, since the agent learns pushing-like behaviors before discovering grasping.

#### 1.4.1.1 Our Dense Debug Results

Metric	Value
Wall time	~15 min (540 fps)
Final rolling success	3-8% (fluctuating)
Eval success (50 ep)	0.0%
Mean return	-13.77
Mean final distance	0.2754m
Table return / distance	-9.78 / 0.196m (n=19)
Air return / distance	-16.22 / 0.324m (n=31)
Verdict	<b>LEARNING</b>

The verdict is LEARNING: the mean return of -13.77 is above the random-policy baseline of approximately -15, confirming the agent learned to approach objects.

**Why 0% eval success despite 3-8% training success?** This gap is common in RL and worth understanding. Training success is a rolling average across many episodes using stochastic (exploratory) actions, so occasional lucky grasps from random exploration inflate the number. Deterministic eval, by contrast, uses the policy's mode (no exploration noise), which means that at a ~4% true success rate, 50 episodes give an expected ~2 successes -- but the variance is high, and the probability of seeing exactly zero successes is  $0.96^{50} \approx 13\%$ , placing our result within statistical noise. Mean return is the more reliable signal because it averages over all episodes rather than thresholding a binary outcome, giving it much lower variance. The improvement from -15 (random) to -13.77 reliably indicates learning even when the success count happens to be zero.

The stratified breakdown reveals exactly the learning progression predicted in section 5.3. Table goals (return -9.78, distance 0.196m) are nearly **2x closer** to success than air goals (return -16.22, distance 0.324m). The agent has learned to move toward objects on the table -- a behavior similar to Push -- but has not yet discovered grasping and lifting. This is the expected order: pushing is a simpler behavior that the agent discovers first through random exploration.

Without stratification, we would see only the blended return of -13.77 and miss the fact that the agent is learning table-reaching behaviors while air goals remain essentially unimproved from random.

#### 1.4.2 5.11 Experiment 1: SAC+HER Sparse PickAndPlace

```
# Single seed (~1-1.5 hours)
bash docker/dev.sh python scripts/ch05_pick_and_place.py train --seed 0

# Faster test run to verify setup (~15 min)
bash docker/dev.sh python scripts/ch05_pick_and_place.py train --seed 0 --total-steps 5000
```

Training uses ch04's proven configuration:

Parameter	Value	Rationale
env	FetchPickAndPlace-v4	Sparse PickAndPlace
algo	SAC	Off-policy for HER
her	True	Required for sparse rewards
gamma	0.95	Ch04 sweep winner
ent_coef	0.05	Prevents entropy collapse
total_steps	5M	More than Push due to task complexity

Parameter	Value	Rationale
n_envs	8	Parallel data collection

#### 1.4.2.1 Long-Running Jobs with tmux

```
# Multi-seed training (~5 hours for 3 seeds)
tmux new -s week5
bash docker/dev.sh python scripts/ch05_pick_and_place.py all --seeds 0,1,2
# Detach: Ctrl-b d
# Reattach: tmux attach -t week5
```

#### 1.4.3 5.12 Stratified Evaluation

```
bash docker/dev.sh python scripts/ch05_pick_and_place.py eval \
  --ckpt checkpoints/sac_her_FetchPickAndPlace-v4_seed0.zip
```

The evaluation reports overall, air, and table success rates separately. Our actual output (seed 0):

```
=====
Evaluation Results: FetchPickAndPlace-v4
=====
Overall: 100.0% success (100 episodes)
Air goals: 100.0% success (56 episodes)
Table goals: 100.0% success (44 episodes)
Time to success: 10.7 steps (among successes)
Final distance: 0.0138m
Action smoothness: 0.196
```

Action smoothness measures how much consecutive actions differ (we will define this formally in Chapter 6). Lower values indicate smoother, more physically plausible motion.

**What to look for:** The overall success rate target is >60% (good) or >80% (strong) -- we achieved **100%**. The air gap (table success minus air success) measures whether the agent pushes well but grasps poorly; a gap >20% would signal incomplete learning, and ours is **0%**. Time to success should be shorter for table goals since they involve simpler behavior with fewer phases, which we confirmed at **8.2 vs 12.6 steps**. Finally, air goals achieve tighter final placement (0.009m) than table goals (0.018m) because grasping gives more precise control than pushing, where the object can slide unpredictably.

#### 1.4.4 5.13 Stress Evaluation

```
bash docker/dev.sh python scripts/ch05_pick_and_place.py stress \
  --ckpt checkpoints/sac_her_FetchPickAndPlace-v4_seed0.zip
```

Injects observation noise ( $\sigma = 0.01$ ) and action noise ( $\sigma = 0.05$ ) during evaluation using a NoisyEvalWrapper (a lightweight gym.Wrapper that adds Gaussian noise to observations and actions at each step). Our actual output (seed 0):

```
=====
Stress Test Results: FetchPickAndPlace-v4
=====
Condition | Success | Air | Table
-----
Clean     | 100.0% | 100.0% | 100.0%
```

Stress	100.0%	100.0%	100.0%
Degradation	+0.0%	+0.0%	+0.0%

Verdict: ROBUST -- less than 10% degradation under noise

**What to look for:** Degradation below 10% indicates a robust policy, and we achieved **1.7% mean degradation** -- well within the ROBUST threshold. As predicted, grasping is more noise-sensitive than pushing: air degradation reached **up to 7.1%** (seed 2) while table degradation remained at **0%**. A fragile policy would suggest reliance on precise trajectories, but our policies are robust -- even the worst seed under noise achieves 96% overall and 92.9% on air goals.

#### 1.4.5 5.14 Full Pipeline

```
# Complete ch05: dense-debug -> train (3 seeds) -> eval -> stress -> compare
bash docker/dev.sh python scripts/ch05_pick_and_place.py all --seeds 0,1,2
```

This runs all stages in sequence (~5 hours total for 3 seeds). The compare stage aggregates results across seeds and computes mean +/- std for each metric.

#### 1.4.6 5.15 Results and Analysis

##### 1.4.6.1 Hyperparameter Transfer from Push

Metric	Push (ch04)	PickAndPlace (ch05)	Transfer?
Overall success	99.4% +/- 0.9%	<b>100.0% +/- 0.0%</b>	Yes
Training steps	2M	5M	More steps needed
gamma	0.95	0.95	Same
ent_coef	0.05	0.05	Same

**The hyperparameters transfer completely.** Ch04's configuration ( $\gamma = 0.95$ ,  $\text{ent\_coef} = 0.05$ ,  $n_{\text{sampled\_goal}} = 4$ ) achieves 100% success on PickAndPlace across all 3 seeds -- matching or exceeding Push's 99.4%. This is strong evidence that these values capture general properties of sparse goal-conditioned learning in Fetch environments, not task-specific tuning.

The learning curve reveals when grasping "clicks":

Steps	Seed 0	Seed 1	Seed 2
1M	19%	17%	8%
2M	52%	42%	--
3M	96%	69%	--
4M	95%	99%	--
5M	<b>98%</b>	<b>98%</b>	--

The inflection around 2-3M steps marks where the agent transitions from primarily pushing (table goals) to reliable grasping (air goals). Before this point, success rates are 17-52% -- mostly table goals solved by pushing. After 3M steps, grasping clicks and success jumps to 69-98%.

##### 1.4.6.2 Dense Debug (Pipeline Validation)

Metric	Value	Interpretation
Success (50 ep)	0.0%	Within noise at ~4% true rate
Mean return	-13.77	Above random baseline (-15)
Table return	-9.78	Agent learns table-reaching first
Air return	-16.22	Grasping not yet discovered
Air gap (return)	6.44	Table goals ~2x easier
Verdict	LEARNING	Pipeline works, task needs more steps

#### 1.4.6.3 Stratified Breakdown (Sparse, 5M steps, 3 seeds)

Goal Type	Success Rate	Time to Success	Final Distance
Air	100.0% +/- 0.0%	~12.6 steps	0.0092m
Table	100.0% +/- 0.0%	~8.2 steps	0.0176m
Gap	0.0%	--	--

Success rates are mean +/- 95% CI across 3 seeds. Time-to-success and final distance are averaged across all episodes of seed 0 (100 episodes); cross-seed variation is negligible (< 0.3 steps).

The air gap is **zero** -- the agent has fully mastered grasping. Two observations are worth noting. Air goals take longer (12.6 vs 8.2 steps) because they require the full multi-phase sequence (approach, grasp, lift, carry, place) while table goals can be solved by shorter pushing trajectories. Perhaps counterintuitively, air goals also achieve tighter final distance (0.009m vs 0.018m), which makes sense once we consider that grasping gives the agent precise control over placement, whereas pushing is inherently less precise because the object can slide unpredictably.

#### 1.4.6.4 Stress Test Summary (3 seeds)

Condition	Overall	Air	Table
Clean	100.0%	100.0%	100.0%
Stress (seed 0)	100.0%	100.0%	100.0%
Stress (seed 1)	99.0%	98.2%	100.0%
Stress (seed 2)	96.0%	92.9%	100.0%
<b>Mean degradation</b>	<b>1.7% +/- 2.4%</b>	<b>3.0%</b>	<b>0.0%</b>
Verdict	<b>ROBUST</b>		

The +/- values are 95% CI across 3 seeds. The policies are robust under noise ( $\sigma_{\text{obs}} = 0.01$ ,  $\sigma_{\text{act}} = 0.05$ ), with mean degradation of only 1.7%. As predicted in section 5.7, **air goals are more noise-sensitive** than table goals (up to 7.1% degradation for seed 2 vs 0% for table). This confirms that grasping requires more precise finger-object coordination, but even the worst case (92.9% air success under noise) is well within the ROBUST threshold.

#### 1.4.7 5.16 CLI Parameter Reference

`scripts/ch05_pick_and_place.py <command> [options]`

Commands:

`dense-debug` Quick validation on dense PickAndPlace (~13 min)

```

train      Train SAC+HER on sparse PickAndPlace
eval       Evaluate with stratified metrics (air vs table)
stress    Stress test with noise injection
compare   Compare results across seeds
all       Full pipeline: dense-debug -> train -> eval -> stress -> compare

```

Key options:

--seed N	Random seed (default: 0)
--seeds 0,1,2	Seeds for multi-seed runs
--total-steps N	Training steps (default: 5M)
--her / --no-her	Enable/disable HER (default: on)
--curriculum	Enable curriculum learning
--gamma F	Discount factor (default: 0.95)
--ent-coef F	Entropy coefficient (default: 0.05)
--stress-obs-noise	Observation noise std (default: 0.01)
--stress-act-noise	Action noise std (default: 0.05)

#### 1.4.8 Artifact Locations

Artifact	Path
Dense debug checkpoint	checkpoints/sac_her_FetchPickAndPlaceDense-v4_dense_debug.zip
Sparse checkpoints	checkpoints/sac_her_FetchPickAndPlace-v4_seed{N}.zip
Dense debug eval	results/ch05_dense_debug_eval.json
Stratified eval	results/ch05_sac_her_fetchpickandplace-v4_seed{N}_eval.json
Stress reports	results/ch05_sac_her_fetchpickandplace-v4_seed{N}_stress.json
Comparison	results/ch05_fetchpickandplace-v4_comparison.json
TensorBoard	runs/sac_her_nsg4/FetchPickAndPlace-v4/seed{N}/

## 1.5 Summary

### 1.5.1 Concepts Introduced in This Chapter

Concept	Definition
Dense-first debugging	Validate pipeline on dense rewards before committing to sparse training
Multi-phase control	Sequential coordinated behaviors (approach, grasp, lift, carry, place)
Goal stratification	Separate evaluation of air goals vs table goals
Air gap	$SR_{table} - SR_{air}$ : quantifies grasp learning vs push learning
Stress evaluation	Noise injection to quantify policy robustness
NoisyEvalWrapper	gym.Wrapper that injects observation and action noise during eval
Degradation	Performance drop under noise: robust (<10%), moderate (10-30%), fragile (>30%)
Curriculum learning	Gradually increasing goal difficulty during training
Difficulty schedule	Mapping from training progress to goal distribution
CurriculumGoalWrapper	gym.Wrapper that replaces goal sampling with difficulty-controlled distribution
Air goal / table goal	Goal classification based on z-coordinate relative to table surface

### 1.5.2 Files Generated

File	Purpose
<code>scripts/labs/curriculum_wrapper.py</code>	Pedagogical curriculum wrapper (Build It)
<code>scripts/ch05_pick_and_place.py</code>	Chapter script: train, eval, stress, compare
<code>tutorials/ch05_pick_and_place.md</code>	This tutorial

### 1.5.3 What Comes Next

Chapter 6 treats trained policies as **controllers** -- not just as RL outputs, but as components in a control system. We will quantify action smoothness, compare against classical baselines, and study how action scaling affects both performance and physical plausibility.

The transition from "can the robot succeed?" (chapters 2-5) to "how well does the robot behave?" (chapters 6-7) is where RL meets robotics engineering.