5 HER on Sparse Reach and Push: Learning from Failure

Vlad Prytula

2026-02-28

Contents

5 HER on Sparse Reach and Push: Learning from Failure

This chapter covers:

- Why sparse rewards create a needle-in-a-haystack problem -- and why SAC alone achieves only 5% success on FetchPush-v4 when rewards are binary
- The HER insight: relabeling failed trajectories with goals that were actually achieved, turning every failure into a learning opportunity
- Implementing HER from scratch: goal sampling strategies (future, final, episode), transition relabeling with reward recomputation, and the full episode processing pipeline that amplifies data from 50 transitions to ~210
- Bridging from-scratch code to SB3's HerReplayBuffer: verifying the compute_reward invariant holds across both implementations
- Training SAC+HER on two sparse-reward tasks: FetchReach-v4 (marginal improvement, 96% -> 100%) and FetchPush-v4 (transformative improvement, 5% -> 99% success rate across 3 seeds)

In Chapter 4, you derived SAC from the maximum entropy objective, implemented it from scratch (replay buffer, twin Q-networks, squashed Gaussian policy, automatic temperature tuning), bridged to SB3, and achieved 100% success on FetchReachDense-v4. The entire off-policy stack is validated -- every transition goes into a replay buffer and gets reused across many gradient updates.

But dense rewards required us to hand-design a distance-based signal. The robot received continuous feedback proportional to how far it was from the goal: ”you're getting warmer” at every timestep. What if we only know whether the robot succeeded or failed? Sparse rewards -- $r = 0$ for success, $r = -1$ for failure -- are more natural because they do not require knowing the right distance metric, but they create a needle-in-a-haystack exploration problem. On FetchPush-v4 with sparse rewards, SAC alone achieves only 5% success: the puck almost never lands on the goal by chance, so the agent has no signal to learn from.

This chapter introduces Hindsight Experience Replay (HER), which transforms failures into learning signal by asking: ”what goal would this trajectory have achieved?” You will implement goal sampling, transition relabeling, and reward recomputation from scratch, verify each component, bridge to SB3's HerReplayBuffer, and watch success on FetchPush-v4 jump from 5% to 99%. HER uses the off-policy replay buffer from Chapter 4 -- relabeled transitions were not generated by the current policy, so only off-policy algorithms can use them.

One note on where this leads: HER solves the exploration problem for goal-conditioned tasks with known goal spaces. Chapter 6 applies the full SAC+HER stack to the hardest Fetch task -- PickAndPlace -- where the robot must lift an object off the table, requiring curriculum strategies and stress-testing to achieve reliability.

5.1 WHY: The sparse reward problem

Failure first: SAC without HER on FetchPush-v4

Before we introduce any new ideas, let's look at what happens when you train the SAC algorithm from Chapter 4 on a sparse-reward pushing task -- no HER, no tricks, just the off-policy stack you already built.

FetchPush-v4 is a step up from FetchReach. Instead of moving the gripper to a target, the robot must push a puck across a table to a goal position. The observation is 25-dimensional (gripper position, velocity, puck position, puck velocity, relative positions), the action is the same 4D Cartesian delta as before, and the reward is sparse: $r = 0$ if the puck is within 5cm of the goal, $r = -1$ otherwise.

Here are the results from training SAC without HER on FetchPush-v4 for 2 million steps (3 seeds, 100 evaluation episodes each):

Metric	SAC (no HER)
Success Rate	5.0% +/- 0.0%
Mean Return	-47.50 +/- 0.00
Final Distance	184.5mm +/- 0mm

A 5% success rate means the puck lands near the goal roughly once every 20 episodes -- essentially by accident. The mean return of -47.50 out of a maximum of 0 tells you the agent is failing at nearly every timestep of every episode. And the final distance of 184.5mm (the puck ends up about 18cm from the goal on average) shows the agent has not learned to push at all.

The three seeds agree almost perfectly, which tells us this is a structural limitation of SAC on sparse rewards, not a matter of unlucky initialization. Figure 5.1 makes this visible: a flat line at 5% for the entire training run.

SAC without HER on FetchPush-v4: success rate stuck at approximately 5 percent over 2M training steps, with three seeds overlaid showing consistent failure. A dashed line at 99 percent shows the HER target for contrast.

Figure 5.1: SAC without HER on FetchPush-v4. The success rate stays at approximately 5% across 2M training steps, consistent across 3 seeds. The dashed line at 99% shows

what SAC+HER achieves. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/sac/FetchPush-v4/seed{0,1,2}/`.)

Why does standard RL fail here?

The problem is not that SAC is a bad algorithm. SAC achieved 100% success on FetchReachDense in Chapter 4. The problem is the reward structure.

With sparse rewards, the agent receives $r = -1$ for almost every transition, and the consequences cascade through the entire learning pipeline. Initial exploration is random -- the gripper moves chaotically while the puck sits on the table -- so most episodes fail completely, since the gripper rarely contacts the puck and pushing it to exactly the right spot by chance is extremely unlikely. Because nearly every transition in the replay buffer carries reward -1, the critic learns that everything is equally bad, which means the Q-value function becomes flat with no state-action pair looking better than any other. Without reward variation, the policy has no direction to improve: the actor loss from Chapter 4 (Section 4.7) relies on Q-value differences to select actions, and when all Q-values are the same, the gradient points nowhere useful.

This is the needle-in-a-haystack problem. The sparse reward contains information -- it tells you whether you succeeded -- but it provides that information so rarely that the learning algorithm has nothing to work with.

Sparse rewards are the natural formulation

Despite this failure, sparse rewards are more honest than dense rewards. In Chapter 4, we used a dense reward proportional to distance:

$$R_{\text{dense}}(s, g) = -\|g_{\text{achieved}} - g_{\text{desired}}\|$$

This works, but it encodes a strong assumption: that the straight-line distance to the goal is the right measure of progress. For reaching tasks that assumption holds, and for pushing it partially holds (the puck should get closer to the goal), but for more complex tasks -- stacking, assembly, tool use -- designing the right distance metric is itself a hard problem, so you end up engineering the reward rather than solving the task.

Sparse rewards avoid this:

$$R_{\text{sparse}}(s, g) = \begin{cases} 0 & \text{if } \|g_{\text{achieved}} - g_{\text{desired}}\| < \epsilon \\ -1 & \text{otherwise} \end{cases}$$

where $\epsilon = 0.05$ meters (5cm) is the success threshold. The sparse reward asks a binary question -- ”did you succeed?” -- without assuming anything about how to measure progress, which means that if we can learn from sparse rewards, we have a more general solution.

The challenge is making that learning possible.

Why Push is harder than Reach

You might wonder: does SAC without HER also fail on sparse Reach? Not quite. Here are the Reach results:

Metric	SAC (no HER) on FetchReach-v4
Success Rate	96.0% +/- 7.0%

Reach achieves 96% without HER because random exploration occasionally succeeds. The gripper workspace is roughly 15cm across and the success threshold is 5cm, so random flailing in a 15cm cube has a reasonable chance of landing within 5cm of a random target -- roughly every few episodes, the gripper stumbles into the goal by accident. Those accidental successes provide enough reward signal for the critic to learn: ”being near the goal is good.”

Push is different. Success requires a coordinated sequence: approach the puck, make contact, push it in the right direction, and stop it at the goal. Even a generous estimate of the probability of achieving this by random exploration is vanishingly small. We call this the **effective horizon** -- the number of coordinated steps needed for success. For Reach, the effective horizon is roughly 2-5 steps (just move toward the target). For Push, it is 20-50 steps (approach + contact + push + stop). Random exploration at each step has perhaps a 10-20% chance of being correct, so the probability of 20 consecutive correct steps is $(0.15)^{20} \approx 10^{-16}$.

In 2 million training steps with 8 parallel environments, the agent experiences roughly 40,000 episodes. Even at 40,000 episodes, the chance of a single random success on Push is negligible. The agent needs a way to extract learning signal from all these failed episodes -- and that is exactly what HER provides.

5.2 HOW: Hindsight Experience Replay

The insight

Here is the idea that makes HER work, stated plainly:

> A trajectory that fails to reach goal g is a successful demonstration of how to reach wherever it ended up.

If the robot tried to push the puck to position $(0.3, 0.2)$ but the puck ended at $(0.5, 0.4)$, we have evidence that the executed actions push the puck to $(0.5, 0.4)$. We can **relabel** the trajectory: pretend the goal was $(0.5, 0.4)$ all along, and now we have a successful episode with reward $r = 0$.

This is Hindsight Experience Replay (Andrychowicz et al., 2017). The name captures the idea: in hindsight, we reinterpret what the agent was trying to do. We are not changing the physics or the actions -- we are changing the question. Instead of ”did you reach the intended goal?”, we ask ”what goal did you reach?”

The insight is simple, but it changes everything about how the agent learns. Every failed episode becomes a source of learning signal. The agent does not need to succeed by chance -- it learns from its failures by reinterpreting them as successes for different goals.

Formal definition

Definition (Hindsight Experience Replay).

Motivating problem. With sparse rewards, the replay buffer contains almost exclusively failed transitions ($r = -1$). The critic cannot distinguish promising actions from useless ones, because all actions look equally bad.

Intuitive description. After each episode, we add the original transitions to the replay buffer (with the real goal), and also add relabeled copies where the goal is replaced with a goal that was actually achieved during the episode. The reward is recomputed for the new goal. Since the achieved goal is exactly where the agent ended up, many of these relabeled transitions are successes ($r = 0$).

Formal definition. Given an episode of transitions $\{(s_t, a_t, r_t, s_{t+1}, g_d)\}_{t=0}^{T-1}$ with desired goal g_d and achieved goals $\{g_a^t\}_{t=0}^{T-1}$, HER adds to the replay buffer:

1. The original transitions (with goal g_d and reward r_t)
2. For each transition t, k relabeled copies where g_d is replaced by $g' \sim S(t)$ (a goal sampled according to a strategy S), and the reward is recomputed: $r'_t = R(g_a^t, g')$

The reward recomputation is critical. We do not assume the relabeled transition is a success -- we **recompute** the reward using the environment's `compute_reward` function. If $\|g_a^t - g'\| < \epsilon$, then $r'_t = 0$ (success); otherwise $r'_t = -1$ (failure).

Grounding example. A 50-step failed episode where the puck ends up 18cm from the goal. Every original transition has $r = -1$. With $k = 4$ relabeled copies per transition and a relabeling ratio of 0.8, we generate $50 + 50 \times 0.8 \times 4 = 210$ total transitions. Because the relabeled goals come from the trajectory itself (where the puck actually went), many relabeled transitions satisfy $\|g_a - g'\| < 0.05$, producing $r' = 0$. The success fraction in the augmented data jumps from ~0% to ~4-80%, depending on how closely the trajectory's achieved goals cluster -- random trajectories produce ~4-8%, while purposeful trajectories from a partially trained policy produce 60-80%.

Non-example. HER does not invent data. The observations, actions, and achieved goals are real -- they came from actual environment interaction. Only the desired goal and the reward are modified. HER also does not guarantee that relabeled transitions are successes: if the achieved goal at timestep t is far from the sampled goal g', the relabeled reward is still -1.

Goal sampling strategies

The sampling strategy $S(t)$ determines which achieved goals are used for relabeling. Andrychowicz et al. (2017, Section 3.3) proposed three strategies:

Strategy	Description	Formal
FUTURE	Sample from achieved goals at timesteps after t	$g' \sim$ Uniform$(\{g_a^{t'} : t' > t\})$
FINAL	Always use the last achieved goal	$g' = g_a^{T-1}$
EPISODE	Sample from any achieved goal in the episode	$g' \sim$ Uniform$(\{g_a^{t'} : t' \in [0, T-1]\})$

FUTURE is the most common choice and works best in practice, because goals from future timesteps are ”reachable” from the current state -- the agent demonstrated that it could get there from here. Goals from earlier timesteps are less useful since they represent states the agent has already passed through.

FINAL is simpler but less diverse, since every relabeled transition uses the same goal. The agent therefore only learns about reaching the endpoint of each trajectory, not the intermediate states along the way.

EPISODE is the most diverse but includes goals from before the current timestep. These backward goals are reachable in principle (the agent was there earlier), yet the actions taken to reach them point in the reverse direction, which can confuse the critic.

The parameter k: how many goals to sample

For each original transition, HER creates k relabeled copies. The default from Andrychowicz et al. (2017) is $k = 4$, which means each transition appears in the buffer once with the real goal and up to four times with relabeled goals.

Why $k = 4$ and not $k = 1$ or $k = 100$? With $k = 1$, you double the data but the success fraction stays relatively low. With $k = 100$, you generate 100x more data, but it is all from the same underlying trajectory -- the marginal value of each additional relabeled copy decreases. $k = 4$ provides a good balance between data amplification and diversity. Our 120-run hyperparameter sweep (detailed in the Reproduce It section) confirmed that $k = 4$ performs within noise of $k = 8$, consistent with the original paper's recommendation.

Data amplification: the quantitative effect

The arithmetic of HER reveals why it is so powerful. Consider a 50-step episode with $k = 4$ and `her_ratio = 0.8` (the probability that a given transition receives relabeled copies):

- **Original transitions:** $T = 50$
- **Relabeled transitions:** $T \times$ her_ratio $\times k = 50 \times 0.8 \times 4 = 160$
- **Total:** $50 + 160 = 210$

The **data amplification ratio** is $210/50 = 4.2$x. From a single episode, we extract over four times as many training transitions.

More importantly, the **success fraction** changes dramatically. In the original 50 transitions, all have $r = -1$ (the episode failed). In the 160 relabeled transitions, a significant fraction have $r = 0$ because the relabeled goal was achieved. The exact fraction depends on how closely the trajectory's achieved goals cluster -- with random trajectories it might be ~4-8%, with a partially trained policy it can reach 60-80%.

This is the core quantitative insight of HER: it manufactures dense reward signal from sparse feedback, not by changing the reward function, but by reinterpreting the data.

Why HER requires off-policy learning

HER only works with off-policy algorithms like SAC and TD3. The reason is structural, and understanding it helps clarify what HER actually does to the data. Here are the three conditions:

> **Definition (Off-policy requirement for HER).**
>
> HER requires off-policy learning because:
>
> 1. **Relabeled transitions were not generated by the current policy.** The policy that collected the data was trying to reach goal g_d, not goal g'. The actions were chosen based on observations conditioned on g_d. Using these transitions to learn about g' is inherently off-policy.
> 2. **The reward is retroactively changed.** The agent experienced $r = -1$, but HER stores $r' = 0$. An on-policy method like PPO computes policy gradients using the actual rewards the agent received. Substituting a different reward invalidates the gradient estimate.
> 3. **The goal is modified after collection.** On-policy methods require that the data distribution matches the current policy. Changing the goal changes the effective data distribution -- the transition now represents behavior under a different goal-conditioned policy than the one being optimized.

This is why the curriculum builds SAC (Chapter 4) before HER (this chapter). SAC's replay buffer stores transitions and reuses them regardless of which policy generated them. HER exploits this by modifying what the stored transitions mean. PPO cannot do this -- it must use each transition exactly as collected, then discard it. Figure 5.2 illustrates how relabeling transforms a single failed trajectory.

HER relabeling diagram showing a 5-step trajectory as a horizontal sequence of states. The original desired goal (red X) is far away. Achieved goals (blue dots) are along the path. Arrows show how HER substitutes the desired goal with an achieved goal from a future timestep, turning a failure into a success.

Figure 5.2: HER relabeling in action. A 5-step trajectory (blue dots showing achieved goals at each timestep) fails to reach the original desired goal (red X). HER relabels the transition at timestep 2 with the achieved goal at timestep 4. Because the puck

was at that position at timestep 4, the relabeled transition becomes a success ($r = 0$). The three strategies (future, final, episode) differ in which timesteps are eligible for goal sampling. (Illustrative diagram.)

5.3 Build It: Data structures

Before we implement relabeling, we need to define what a goal-conditioned transition looks like. Recall from Chapter 2 that Fetch environments produce dictionary observations with three keys: `observation` (robot state), `achieved_goal` (where the agent is or what it accomplished), and `desired_goal` (the target). HER operates on both goal fields -- it reads `achieved_goal` to know what happened and modifies `desired_goal` to change the question.

Each transition carries seven fields. This is the standard five-tuple from Chapter 4's replay buffer -- $(s, a, r, s', \text{done})$ -- extended with the two goal arrays:

```python
# (from scripts/labs/her_relabeler.py:data_structures)

class Transition(NamedTuple):
    """A single environment transition in a goal-conditioned setting."""
    obs: np.ndarray             # observation (e.g., robot state)
    action: np.ndarray          # action taken
    reward: float               # reward received
    next_obs: np.ndarray        # next observation
    done: bool                  # episode terminated?
    achieved_goal: np.ndarray # goal actually achieved at next_obs
    desired_goal: np.ndarray  # goal the agent was trying to reach


class Episode(NamedTuple):
    """A complete episode of transitions."""
    transitions: list[Transition]

    def __len__(self) -> int:
        return len(self.transitions)


class GoalStrategy(Enum):
    """HER goal sampling strategies (Andrychowicz et al., 2017)."""
    FINAL = "final"       # Use final achieved goal
    FUTURE = "future"     # Sample from future timesteps
    EPISODE = "episode"   # Sample from anywhere in episode
```

The `Transition` extends Chapter 4's five-tuple with `achieved_goal` and `desired_goal`. An `Episode` is a list of transitions from a single environment rollout. The `GoalStrategy` enum encodes the three sampling strategies we discussed in Section 5.2. (The source file also includes a `RANDOM` strategy that samples from the full replay buffer -- we omit it here because it requires buffer access and is rarely used in practice.)

Checkpoint. Construct a transition manually and verify all seven fields are accessible:

```
import numpy as np
t = Transition(
    obs=np.zeros(10), action=np.zeros(4), reward=-1.0,
    next_obs=np.zeros(10), done=False,
    achieved_goal=np.array([0.5, 0.4, 0.15]),
    desired_goal=np.array([0.3, 0.2, 0.10]),
)
assert t.reward == -1.0
assert t.achieved_goal.shape == (3,)
assert GoalStrategy.FUTURE.value == "future"
```

All seven fields should be present. `GoalStrategy.FUTURE.value` should return `"future"`.

5.4 Build It: Goal sampling

HER needs to choose which achieved goals to use as relabeled targets. The three strategies from Section 5.2 are formalized in `sample_her_goals`:

- **FUTURE:** Sample g' from $\{g_a^{t'} : t' > t\}$ -- achieved goals from future timesteps
- **FINAL:** Always use $g' = g_a^{T-1}$ -- the episode's last achieved goal
- **EPISODE:** Sample g' from $\{g_a^{t'} : t' \in [0, T-1]\}$ -- any achieved goal

```
# (from scripts/labs/her_relabeler.py:goal_sampling)

def sample_her_goals(
    episode: Episode,
    transition_idx: int,
    strategy: GoalStrategy = GoalStrategy.FUTURE,
    k: int = 4,
) -> list[np.ndarray]:
    """Sample alternative goals for HER relabeling."""
    n = len(episode)
    goals = []
    if strategy == GoalStrategy.FINAL:
        final_goal = episode.transitions[-1].achieved_goal
        goals = [final_goal.copy() for _ in range(k)]

    elif strategy == GoalStrategy.FUTURE:
        future_indices = list(range(transition_idx + 1, n))
        if len(future_indices) == 0:
            goals = [episode.transitions[-1].achieved_goal.copy()
                     for _ in range(k)]
        else:
            sampled_indices = np.random.choice(
                future_indices,
                size=min(k, len(future_indices)),
                replace=False,
            ).tolist()
            while len(sampled_indices) < k:
                sampled_indices.append(n - 1)
            goals = [episode.transitions[i].achieved_goal.copy()
                     for i in sampled_indices]
```

The EPISODE strategy follows the same pattern but samples from the full episode instead of only future timesteps:

```
    # (continued from sample_her_goals)

    elif strategy == GoalStrategy.EPISODE:
        all_indices = list(range(n))
        sampled_indices = np.random.choice(
            all_indices, size=k, replace=True
        ).tolist()
        goals = [episode.transitions[i].achieved_goal.copy()
                 for i in sampled_indices]

    return goals
```

The FUTURE strategy is the most nuanced. For transition at index t, it samples from indices $[t+1, T-1]$ without replacement (up to k samples). If fewer than k future indices exist (near the end of the episode), it pads with the final achieved goal. This ensures we always return exactly k goals.

FINAL is the simplest -- it repeats the same goal k times. All relabeled transitions from this strategy share the same target, which means less diversity but a strong signal about the trajectory's endpoint.

EPISODE samples with replacement from the full episode, providing the most diversity but including backward goals that may be less useful.

Checkpoint. Create a 20-step synthetic episode and verify each strategy:

```
episode = create_synthetic_episode(n_steps=20)

# FUTURE at idx=5: goals from indices [6, 19]
goals = sample_her_goals(episode, transition_idx=5,
                         strategy=GoalStrategy.FUTURE, k=4)
assert len(goals) == 4
assert all(g.shape == (3,) for g in goals)

# FINAL: all goals identical (last achieved_goal)
goals = sample_her_goals(episode, transition_idx=5,
                         strategy=GoalStrategy.FINAL, k=4)
assert all(np.array_equal(g, goals[0]) for g in goals)
```

FUTURE should return 4 goals each with shape `(3,)`. FINAL should return 4 identical goals.

5.5 Build It: Transition relabeling and reward recomputation

The core HER operation substitutes a new goal and recomputes the reward. Given a transition $(s_t, a_t, r_t, s_{t+1}, g_d)$ and a new goal g', the relabeled reward is:

$$r'_t = R(g_a^t, g') = \begin{cases} 0 & \text{if } \|g_a^t - g'\| < \epsilon \\ -1 & \text{otherwise} \end{cases}$$

where g_a^t is the achieved goal (unchanged) and $\epsilon = 0.05$ meters. The observation and action stay the same -- only the goal and reward change.

The **critical invariant** from Chapter 2 applies here: the reward must be recomputed, not assumed. We do not set $r' = 0$ because ”the goal was achieved” -- we call the environment's reward function and let it compute the answer. This matters because the achieved goal might be close to but not within the threshold of the new goal, in which case $r' = -1$ even after relabeling.

```python
# (from scripts/labs/her_relabeler.py:relabel_transition)

def relabel_transition(
    transition: Transition,
    new_goal: np.ndarray,
    compute_reward_fn: callable,
    threshold: float = 0.05,
) -> Transition:
    """Relabel a transition with a new goal and recompute reward."""
    new_reward = compute_reward_fn(
        transition.achieved_goal,
        new_goal,
        {"distance_threshold": threshold},
    )
    return Transition(
        obs=transition.obs,
        action=transition.action,
        reward=new_reward,
        next_obs=transition.next_obs,
        done=transition.done,
        achieved_goal=transition.achieved_goal,  # unchanged
        desired_goal=new_goal,  # replaced
    )
```

The `sparse_reward` function mirrors the Gymnasium-Robotics `compute_reward` API -- it takes achieved and desired goals plus an info dict, and returns the binary reward:

```python
# (from scripts/labs/her_relabeler.py:relabel_transition)

def sparse_reward(
    achieved_goal: np.ndarray,
    desired_goal: np.ndarray,
    info: dict,
) -> float:
    """Sparse reward: 0 if within threshold, -1 otherwise."""
    threshold = info.get("distance_threshold", 0.05)
    distance = np.linalg.norm(achieved_goal - desired_goal)
    return 0.0 if distance < threshold else -1.0
```

Notice that `achieved_goal` stays the same in the relabeled transition -- only `desired_goal` changes. HER makes the mostly-negative sparse reward learnable by turning failures into a mix of successes and failures through relabeling.

> **Checkpoint.** Relabeling with the transition's own achieved goal should always produce a success (reward = 0), because the distance is zero:
>
> ```python
> episode = create_synthetic_episode(n_steps=10)
> transition = episode.transitions[5]
>
> # Relabel with own achieved goal -> guaranteed success
> relabeled = relabel_transition(
>     transition, transition.achieved_goal, sparse_reward
> )
> assert transition.reward == -1.0   # original: failure
> assert relabeled.reward == 0.0     # relabeled: success
> assert np.array_equal(relabeled.achieved_goal,
>                       transition.achieved_goal)  # unchanged
>
> # Relabel with a distant goal -> still failure
> far_goal = transition.achieved_goal + np.array([1.0, 1.0, 1.0])
> relabeled_far = relabel_transition(
>     transition, far_goal, sparse_reward
> )
> assert relabeled_far.reward == -1.0  # distance >> 0.05m
> ```
>
> If `achieved_goal = [0.50, 0.40, 0.15]` and we relabel with `new_goal = [0.50, 0.40, 0.15]`, distance = 0.000m < 0.05m, so reward = 0 (success). If we relabel with `new_goal = [1.50, 1.40, 1.15]`, distance = 1.73m > 0.05m, so reward = -1 (failure).

5.6 Build It: Episode processing and wiring

The three components -- data structures, goal sampling, and transition relabeling -- wire together into a single function that processes an entire episode. For each transition, `process_episode_with_her` adds the original, then (with probability `her_ratio`) samples k alternative goals and creates relabeled copies:

```python
# (from scripts/labs/her_relabeler.py:her_buffer_insert)

def process_episode_with_her(
    episode: Episode,
    compute_reward_fn: callable,
    strategy: GoalStrategy = GoalStrategy.FUTURE,
    k: int = 4,
    her_ratio: float = 0.8,
) -> list[Transition]:
    """Process an episode: HER-augmented transitions."""
    all_transitions = []

    for idx, transition in enumerate(episode.transitions):
        all_transitions.append(transition)  # always keep original

        if np.random.random() < her_ratio:  # probabilistic relabeling
            her_goals = sample_her_goals(episode, idx, strategy, k)
            for new_goal in her_goals:
                relabeled = relabel_transition(
                    transition, new_goal, compute_reward_fn
                )
                all_transitions.append(relabeled)

    return all_transitions
```

A small helper tells us how many transitions in the augmented batch are successes:

```python
# (from scripts/labs/her_relabeler.py:her_buffer_insert)

def compute_success_fraction(
    transitions: list[Transition],
) -> float:
    """Fraction of transitions with non-negative reward."""
    if not transitions:
        return 0.0
    successes = sum(1 for t in transitions if t.reward >= 0)
    return successes / len(transitions)
```

The processing function iterates through each transition, always keeping the original, and probabilistically adding k relabeled copies. The success fraction is the metric that should jump from ~0% to a meaningful fraction after HER processing.

> **Checkpoint.** Process a 50-step synthetic episode and verify the data amplification arithmetic:
>
> ```python
> episode = create_synthetic_episode(n_steps=50)
>
> # Original: ~0% success (random trajectory, sparse rewards)
> original_success = compute_success_fraction(episode.transitions)
> ```

```
# With HER: ~210 transitions, success fraction > 0%
her_transitions = process_episode_with_her(
    episode, sparse_reward,
    strategy=GoalStrategy.FUTURE, k=4, her_ratio=0.8
)
her_success = compute_success_fraction(her_transitions)

n_original = len(episode)           # 50
n_total = len(her_transitions)      # ~210
amplification = n_total / n_original  # ~4.2x

assert n_total > n_original
assert her_success > original_success
```

You should see approximately 210 total transitions (50 original + ~160 relabeled) with a success fraction meaningfully above 0%. The exact success fraction depends on the random trajectory -- with random actions, expect ~4-8%. With a partially trained policy that moves more purposefully, this would be much higher (60-80%). HER amplifies competence; it does not create it from nothing.

Verification: the full lab

Run the from-scratch implementation's sanity checks end-to-end:

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

Expected output:

```
============================================================
HER Relabeler -- Verification
============================================================
Verifying goal sampling...
  final: sampled 4 goals
  future: sampled 4 goals
  episode: sampled 4 goals
  [PASS] Goal sampling OK

Verifying relabeling...
  Original reward: -1.0
  Relabeled reward: 0.0
  [PASS] Relabeling OK

Verifying HER processing...
  Original success rate: 0.0%
  HER success rate: ~4%
  Transitions: 50 -> ~218
  [PASS] HER processing OK
```

```
============================================================
[ALL PASS] HER implementation verified
============================================================
```

The demo: seeing amplification in action

Run the demo to see how different strategies compare:

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

This processes a 50-step synthetic episode with each of the three strategies and reports the transition count and success fraction for each. FUTURE typically produces the highest success fraction because it provides diverse goals that are naturally close to the achieved states along the trajectory. Figure 5.3 visualizes the amplification effect.

Data amplification visualization: bar chart showing original episode with 50 transitions and approximately 0 percent success compared to HER-processed episode with approximately 210 transitions and a meaningful fraction of successes, annotated with k=4 and her_ratio=0.8.

Figure 5.3: Data amplification effect of HER. A 50-step episode with all failures (left) is processed with $k = 4$ and `her_ratio = 0.8`, producing approximately 210 transitions with a significant fraction of successes (right). The exact success fraction depends on trajectory coherence -- random trajectories produce ~4-8%, while purposeful trajectories from a partially trained policy produce 60-80%. (Generated by `python scripts/labs/her_relabeler.py --demo`.)

This lab is not how we train policies -- SB3's HER wrapper handles that in the Run It section. The lab shows *what* relabeling does to your data, with every operation visible and verifiable. The goal sampling strategies from Section 5.4, the reward recomputation from Section 5.5, and the episode processing from this section are the same math that SB3's `HerReplayBuffer` computes internally. In the next span, we verify that claim directly by comparing our from-scratch output against SB3.

5.7 Bridge: From-scratch to SB3

We claimed that the goal sampling, transition relabeling, and reward recomputation you built in Sections 5.3-5.6 are the same operations that SB3's `HerReplayBuffer` performs internally. Let's verify that directly.

The bridging proof feeds the same synthetic episode through our `process_episode_with_her()` and through SB3's `HerReplayBuffer`, then checks the key invariant: **sampled rewards must equal `compute_reward(achieved_goal, desired_goal)`**. Because sparse rewards are binary (0 or -1), this comparison requires exact equality -- no floating-point tolerance needed.

Run the comparison:

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

Expected output:

```
HER Relabeler -- SB3 Comparison
Max abs reward diff: 0.000e+00
Success fraction:    0.799
n_sampled_goal:      4

[PASS] SB3 HER relabeling is reward-consistent (compute_reward invariant)
```

The reward difference is exactly zero -- every relabeled reward in SB3's buffer matches what `compute_reward(achieved_goal, desired_goal)` returns. The success fraction of 0.799 confirms that HER is creating successes from failures, and `n_sampled_goal=4` confirms both implementations use $k = 4$.

Mapping SB3 parameters to our concepts

SB3's `HerReplayBuffer` constructor takes the same conceptual parameters we built, under slightly different names:

SB3 parameter	Our concept	Section
`n_sampled_goal=4`	$k = 4$ goals per transition	5.2
`goal_selection_strategy="future"`	`GoalStrategy.FUTURE`	5.4
`online_sampling=True`	Relabel at sample time	5.6

The last parameter -- `online_sampling` -- reveals an implementation difference worth understanding. Our from-scratch code relabels at **insertion time**: when an episode ends, `process_episode_with_her()` generates all relabeled copies and stores them immediately. SB3 relabels at **sample time**: it stores original episodes as-is, then performs relabeling lazily when sampling a batch for training. The math is identical; the timing differs.

Online sampling is more memory-efficient -- SB3 stores each episode once and generates relabeled variants on the fly, avoiding the 4.2x storage overhead. For teaching, insertion-time relabeling makes the data amplification visible and verifiable. For production training, SB3's approach scales better.

What SB3 adds beyond our from-scratch code

Our lab processes one episode at a time with explicit Python loops. SB3 adds several engineering features that matter for real training. It provides vectorized episode storage across 8 parallel environments with automatic episode boundary detection, so that it knows where each episode starts and ends within the buffer. This storage integrates directly with SAC's training loop -- no manual episode collection is needed, since the replay buffer, goal relabeling, and gradient updates run as a unified pipeline. SB3 also stores `observation`, `achieved_goal`, and `desired_goal` arrays separately rather than bundling them into named tuples, which is more memory-efficient. And as noted above, online relabeling at sample time avoids the 4.2x storage overhead of materializing all relabeled copies up front.

The from-scratch code exists so you understand *what* relabeling does. SB3 handles the *how fast* and *at what scale*. They compute the same function; the bridging proof confirms it.

5.8 Run It: SAC+HER on sparse Reach and Push

> **A note on script naming.** The production script is called `ch04_her_sparse_reach_push.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, HER on sparse Reach/Push is chapter 4; in this book, it is chapter 5. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch04`. When you see `ch04` in a command or file path, you are running the right script for this chapter.

```
---------------------------------------------------------
EXPERIMENT CARD: SAC + HER on FetchPush-v4 (sparse)
---------------------------------------------------------
Algorithm:   SAC + HER (future strategy, n_sampled_goal=4, ent_coef=0.05, gamma=0.95)
Environments: FetchReach-v4 (sparse, validation), FetchPush-v4 (sparse, primary)
Fast path:    FetchPush-v4, 500,000 steps, seed 0
Time:         ~14 min (GPU) / ~60 min (CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
    --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05

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

Expected artifacts:
  checkpoints/sac_her_FetchPush-v4_seed0.zip
  checkpoints/sac_her_FetchPush-v4_seed0.meta.json
  results/ch04_sac_her_fetchpush-v4_seed0_eval.json
  runs/sac_her/FetchPush-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.90
  mean_return > -20.0
  final_distance_mean < 0.05

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

The Push environment

FetchPush-v4 is a step up from the reaching tasks you have seen so far. Instead of moving the gripper to a target position, the robot must push a puck across a table to a goal location. Figure 5.4 shows the environment layout.

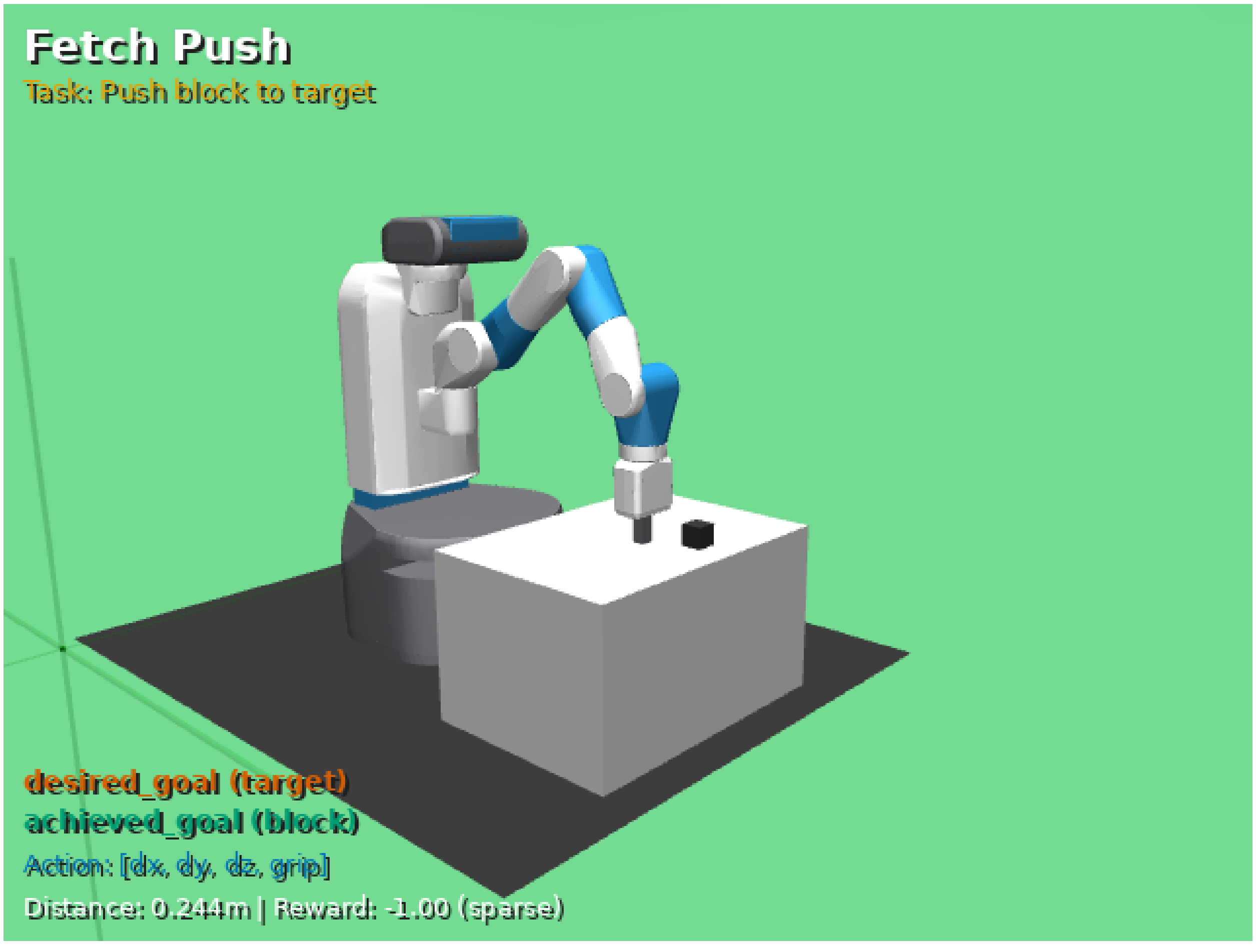
Fetch Push
Task: Push block to target
desired_goal (target)
achieved_goal (block)
Action: [dx, dy, dz, grip]
Distance: 0.244m | Reward: -1.00 (sparse)

Figure 5.4: FetchPush-v4 environment. A 7-DoF robotic arm must push a puck (cylindrical object on the table) to a target position (red marker). The observation is 25-dimensional (gripper state + puck state + relative positions), the action is 4D (Cartesian deltas + gripper), and the reward is sparse: 0 if the puck is within 5cm of the goal, -1 otherwise. (Generated by `python scripts/capture_proposal_figures.py env-setup`.)

The observation is 25-dimensional -- gripper position, gripper velocity, puck position, puck velocity, and relative positions between gripper, puck, and goal. The action is the same 4D Cartesian delta from Chapter 2 (dx, dy, dz, gripper). The complexity comes from indirect control: the gripper must contact the puck and push it in the right direction, requiring a coordinated sequence of approach, contact, push, and stop.

Two key hyperparameters for sparse Push

Before running the experiment, we need to explain two hyperparameter choices that differ from Chapter 4's defaults:

`ent_coef=0.05` (fixed, not auto-tuned). SAC's automatic entropy tuning (Chapter 4, Section 4.8) works well on dense rewards but fails on sparse tasks. The reason is that with sparse rewards, almost every transition has $r = -1$ early in training, so the critic sees no reward variation and Q-values remain nearly flat. The policy then becomes arbitrarily ”confident” (low entropy), and the auto-tuner interprets this as ”the policy knows what it is doing,” reducing α to near zero. Once that happens, exploration stops and the agent never discovers push behavior.

Fixed `ent_coef=0.05` bypasses this failure mode. It maintains a small but steady entropy bonus throughout training, ensuring the agent keeps trying diverse push strategies until it discovers effective behavior.

`gamma=0.95` (not 0.99). The discount factor controls how far into the future the agent ”looks,” with an effective horizon of $T_{\text{eff}} = 1/(1-\gamma)$: at $\gamma = 0.99$, that is 100 steps, while at $\gamma = 0.95$ it is 20 steps. Since a successful push takes roughly 15-25 coordinated steps, $\gamma = 0.95$ matches the task timescale, whereas $\gamma = 0.99$ forces the agent to optimize over steps where it is just waiting after the push is complete -- adding noise without useful signal.

There is also an interaction between these two parameters, because SAC's entropy bonus accumulates over the effective horizon: roughly $\alpha \times T_{\text{eff}} \times \bar{\mathcal{H}}$ total entropy contribution. At $\alpha = 0.05$ and $\gamma = 0.95$ ($T_{\text{eff}} = 20$), this contribution is moderate, but at $\alpha = 0.2$ and $\gamma = 0.98$ ($T_{\text{eff}} = 50$) it becomes 10x larger. In our sweep, we found that high entropy plus long horizon can overwhelm the sparse reward signal entirely.

These values come from a 120-run hyperparameter sweep (24 configurations, 5 seeds each) on FetchPush-v4. The sweep confirmed that γ is the dominant factor (+11 percentage points marginal effect) and that `ent_coef=0.05` outperforms higher values. The full sweep analysis is available in the tutorial repository; for the book, we present the winning configuration directly.

Running the experiments

The one-command version trains both SAC (no HER) and SAC+HER, evaluates both, and generates a comparison report:

```bash
# FetchReach-v4: validation task (~60 min for all 6 runs)
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-all \
  --env FetchReach-v4 --seeds 0,1,2 --total-steps 1000000

# FetchPush-v4: primary task (~3 hours for all 6 runs)
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-all \
  --env FetchPush-v4 --seeds 0,1,2 --total-steps 2000000 --ent-coef 0.05
```

For a quick validation on a single seed (about 14 minutes on GPU):

```bash
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05
```

If you are using the checkpoint track, evaluate the pretrained model directly:

```bash
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py eval \
  --ckpt checkpoints/sac_her_FetchPush-v4_seed0.zip
```

Training milestones for Push

Watch for these milestones during SAC+HER training on FetchPush-v4. The learning curve shows a characteristic pattern -- slow initial progress as HER builds up successful transitions in the buffer, then rapid improvement once the critic has enough signal:

Timesteps	Success Rate	What's happening
0-100k	0-5%	Collecting data, HER creating relabeled successes in buffer
100k-500k	5-40%	Critic learning from relabeled successes, policy improving
500k-1M	40-80%	Rapid improvement -- positive feedback loop: better policy -> better tr
1M-2M	80-99%	Convergence, policy refining push accuracy

The no-HER baseline stays at approximately 5% throughout. This flat line is exactly the failure we saw in Section 5.1.

Results: FetchReach-v4 (sparse)

Reach is the validation task. We run it to confirm that HER does not hurt on an easy task. The results (3 seeds, 100 evaluation episodes each, 1M training steps):

Metric	HER	No-HER	Delta
Success Rate	100% +/- 0%	96% +/- 7%	+4%
Mean Return	-1.68 +/- 0.02	-2.92 +/- 2.04	+1.24
Final Distance	17.0mm +/- 6mm	19.5mm +/- 8mm	-2.5mm

The separation is marginal -- both methods succeed. HER achieves a slightly higher success rate (100% vs 96%) and lower variance. This is expected: FetchReach has a short effective horizon (2-5 steps), so random exploration occasionally succeeds even without relabeling. The 96% no-HER result means roughly 4 out of 100 evaluation episodes fail -- usually when the goal spawns at the edge of the workspace.

Reach confirms that HER does not hurt on an easy task. To see where HER becomes essential, we turn to Push.

Results: FetchPush-v4 (sparse)

Push is where HER's effect becomes clear. The results (3 seeds, 100 evaluation episodes each, 2M training steps):

Metric	HER	No-HER	Delta
Success Rate	99% +/- 1%	5% +/- 0%	**+94%**
Mean Return	-13.20 +/- 1.48	-47.50 +/- 0.00	+34.30
Final Distance	25.7mm +/- 1mm	184.5mm +/- 0mm	-158.8mm

The improvement is transformative. Without HER, the puck ends up 184.5mm from the goal on average -- the agent has not learned to push at all. With HER, the puck ends up 25.7mm from the goal -- well within the 50mm success threshold. The success rate jumps from 5% to 99%, a 94 percentage point improvement that is consistent across all 3 seeds.

Figure 5.5 shows the learning curves for both environments, making the contrast visible.

HER vs no-HER learning curves on FetchReach-v4 and FetchPush-v4: left panel shows both methods reaching near-100 percent success on Reach, right panel shows no-HER stuck at 5 percent while HER climbs to 99 percent on Push.

Figure 5.5: HER vs no-HER learning curves. Left: FetchReach-v4 -- both methods succeed, with HER providing a marginal improvement (100% vs 96%). Right: FetchPush-v4 -- without HER, success stays flat at approximately 5%; with HER, it climbs to 99%. The shaded regions show +/- one standard deviation across 3 seeds. The contrast on Push makes HER's value concrete: a task that SAC alone cannot solve becomes a 99% success rate with goal relabeling. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/sac{,_her}/Fetch{Reach,Push}-v4/seed{0,1,2}/`.)

What the mean return tells you

The mean return of -13.20 for Push with HER deserves a brief interpretation. Each episode is 50 steps, and the reward is -1 for failure, 0 for success, so a return of -13.20 means the agent fails for roughly the first 13 steps (approaching and aligning the push) and then succeeds for the remaining 37 steps. This maps directly to the task structure: the puck needs to be contacted and pushed before the agent can ”succeed” at each timestep. A perfect policy that pushes immediately would have a return near -5 to -10 (a few steps to approach), whereas a return of -47.50 means failure at almost every step -- the no-HER agent never contacts the puck meaningfully.

Reading TensorBoard during training

Launch TensorBoard to watch the experiments:

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

The key metrics map directly to the concepts you built in the Build It sections:

SB3 TensorBoard key	What it measures	HER expected behavior
`rollout/success_rate`	Fraction of episodes reaching the goal	0% -> 99% over 2M steps
`rollout/ep_rew_mean`	Mean episode return	-50 -> -13 (closer to 0 is better)
`train/ent_coef`	Entropy coefficient α	Fixed at 0.05 (recommended for
`train/critic_loss`	Twin Q-network Bellman error	Should decrease and stabilize
`train/actor_loss`	Policy loss (entropy-regularized)	Should decrease as policy impro

The `rollout/success_rate` metric is what HER directly improves. Without HER, the replay buffer contains almost exclusively $r = -1$ transitions, so the critic cannot distinguish good from bad actions. With HER, the relabeled successes from Section 5.6 -- the same `process_episode_with_her` math that SB3's `HerReplayBuffer` computes -- provide the gradient signal the critic needs.

5.9 What can go wrong

Here are the failure modes we have encountered when training SAC+HER on sparse rewards. For each, we give the symptom, the likely cause, and a specific diagnostic.

Push success_rate stalls at approximately 5% with HER enabled

Likely cause. The entropy coefficient is too high. With `ent_coef=auto` or a value above 0.1, SAC over-explores and never exploits the successful relabeled transitions. The entropy bonus overwhelms the sparse reward signal.

Diagnostic. Check three things: (1) verify `ent_coef=0.05` (fixed, not auto-tuned); (2) verify the `--her` flag is actually passed -- check the checkpoint's `.meta.json` to confirm `"her": true`; (3) look at TensorBoard's `train/ent_coef` -- if it shows a value much higher than 0.05, the fixed setting did not take effect.

Push success_rate stalls at approximately 5% even with correct entropy

Likely cause. Wrong discount factor. With `gamma=0.99`, the effective horizon is 100 steps -- far beyond the 15-25 steps a push takes. The critic tries to estimate value over steps where the agent is idle, adding noise.

Diagnostic. Verify `gamma=0.95` (not 0.99). Also check that `total_steps >= 2000000` -- Push needs substantially more training than Reach. Finally, verify `n_envs=8` -- fewer environments mean slower data collection.

Reach success_rate is 0% for the first 100k steps then jumps to 100%

Likely cause. This is normal behavior, not a bug. SAC+HER on sparse Reach takes longer to start learning than dense Reach because the initial replay buffer contains only $r = -1$ transitions. Once enough relabeled successes accumulate, the critic rapidly learns the value function. This is the ”hockey-stick” learning curve pattern.

Diagnostic. No fix needed. Let it train. The jump is expected once the buffer has enough relabeled successes for the critic to distinguish promising states from hopeless ones.

`ep_rew_mean` stays at -50 throughout training

Likely cause. HER is not enabled, or the environment is wrong. A mean return of -50 on a 50-step episode means every single timestep is a failure ($r = -1$), which indicates the agent has not learned any push behavior at all.

Diagnostic. Verify the `--her` flag is set. Check that the environment is `FetchPush-v4` (not `FetchPushDense-v4`, which uses a different reward scale). Print the SB3 model to confirm `HerReplayBuffer` is being used as the replay buffer class.

`--compare-sb3` fails with reward mismatch

Likely cause. SB3 version incompatibility. The `HerReplayBuffer` API changed between SB3 versions, and older versions may handle batched `compute_reward` calls differently.

Diagnostic. Check that `stable-baselines3 >= 2.0` is installed. Verify that the `GoalEnvStub` used in the comparison handles batched inputs correctly -- `compute_reward(achieved_goal, desired_goal, info)` must work when `achieved_goal` and `desired_goal` are 2D arrays, not just 1D.

`--compare-sb3` shows success_fraction equal to 0.0

Likely cause. The synthetic episode's goals are not far enough apart. If the desired goal happens to be near the achieved goals, all original transitions are already successes, and HER relabeling has no failures to turn into successes.

Diagnostic. Verify that the stub environment places the desired goal at least 10 units from the achieved goals (well beyond the 0.05m success threshold). Check that `n_sampled_goal=4` -- a value of 0 would mean no relabeling occurs.

Training much slower than expected (below 300 fps on GPU)

Likely cause. HER adds overhead to replay buffer sampling. Each minibatch requires goal relabeling and reward recomputation, which adds approximately 20% overhead compared to plain SAC.

Diagnostic. Check that `gradient_steps=1` (the default). Approximately 500 fps is typical for SAC+HER on Fetch tasks with GPU. If throughput is below 200 fps, run `nvidia-smi` inside the container to verify the GPU is in use.

NaN in Q-values during Push training

Likely cause. Sparse reward combined with high gamma creates large Bellman targets. The entropy bonus accumulates over a long effective horizon, and the resulting target values can exceed the range of float32 precision.

Diagnostic. Reduce gamma from 0.99 to 0.95. Verify that `ent_coef` is not 0.0 (some exploration is needed to populate the buffer with diverse transitions). Check that rewards are the expected binary values (0 or -1), not something unexpected from a misconfigured environment.

Reach HER barely outperforms no-HER baseline

Likely cause. This is expected, not a failure. FetchReach-v4 has a short effective horizon (2-5 steps), so random exploration occasionally reaches the goal by chance. Without HER, those accidental successes are sufficient for learning. HER's benefit is marginal on tasks where random success is common.

Diagnostic. Run the Push experiment to see the dramatic improvement. HER's value scales with the difficulty of the exploration problem -- the harder the task, the larger the gap between HER and no-HER.

5.10 Summary

This chapter introduced Hindsight Experience Replay and demonstrated its transformative effect on sparse-reward goal-conditioned tasks. Here is what you accomplished:

- **The sparse reward problem.** You trained SAC without HER on FetchPush-v4 and observed a 5% success rate across 3 seeds -- consistent across seeds, pointing to a structural limitation rather than unlucky initialization. Sparse rewards provide almost no gradient signal because the agent rarely succeeds by chance. The effective horizon for Push (20-50 coordinated steps) makes random success vanishingly improbable.
- **The HER insight.** A trajectory that fails to reach its intended goal is a successful demonstration of reaching wherever it ended up. By relabeling the desired goal with an achieved goal and recomputing the reward, HER turns failures into learning signal.
- **From-scratch implementation.** You built four components: data structures (the 7-tuple `Transition`, `Episode`, `GoalStrategy` enum), goal sampling (FUTURE, FINAL, EPISODE strategies), transition relabeling with reward recomputation (the critical invariant -- never assume the reward, always recompute it), and the full episode processing pipeline. A 50-step episode with $k = 4$ and `her_ratio=0.8` produces approximately 210 transitions with a success fraction that jumps from 0% to 4-80% depending on trajectory coherence.
- **Bridge to SB3.** The bridging proof confirmed that our relabeled rewards match SB3's `HerReplayBuffer` output exactly (max absolute reward difference = 0.0). SB3 adds vectorized episode storage, automatic boundary detection, and online relabeling at sample time -- but computes the same relabeling math.
- **Production results.** SAC+HER on FetchPush-v4 achieved 99% +/- 1% success rate across 3 seeds, a 94 percentage point improvement over the no-HER baseline. On FetchReach-v4, HER provided a marginal improvement (100% vs 96%), confirming it does not hurt on easy tasks but showing that Push is where HER's value becomes visible.

Two hyperparameter choices matter for sparse Push: fixed `ent_coef=0.05` (bypassing auto-tuning that collapses on sparse rewards) and `gamma=0.95` (matching the 15-25 step task timescale). These came from a 120-run hyperparameter sweep that identified gamma as the dominant factor.

What comes next

HER solves the exploration problem for goal-conditioned tasks where the goal space is known and the agent can learn from achieved goals. But Push is a table-level task -- the puck stays on the surface. Chapter 6 applies the full SAC+HER stack to FetchPickAndPlace-v4, the hardest Fetch task, where the robot must lift an object off the table and place it at a 3D goal that can be above the surface. PickAndPlace introduces new challenges: the gripper must close on the object (a discrete-like decision within continuous actions), grasp stability matters (the object can slip), and the goal space is three-dimensional rather than two-dimensional. We will explore curriculum strategies that introduce goals at increasing difficulty and stress-test the trained policy to build confidence before deployment.

The off-policy machinery from Chapter 4 (SAC's replay buffer and value estimation) and the goal relabeling from this chapter (HER's data amplification) form the complete algorithmic stack. Chapter 6 is about applying that stack to a harder task and developing the engineering practices -- curriculum, evaluation, robustness testing -- that bridge the gap between ”it works on the benchmark” and ”it works reliably.”

Reproduce It

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

  # Sparse Reach: HER vs no-HER (3 seeds, 1M steps each)
  bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-
all \
    --env FetchReach-v4 --seeds 0,1,2 --total-steps 1000000

  # Sparse Push: HER vs no-HER (3 seeds, 2M steps each)
  bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py env-
all \
   --env FetchPush-v4 --seeds 0,1,2 --total-steps 2000000 --ent-coef 0.05

Hardware:    Any machine with Docker (GPU optional; tested on NVIDIA GB10)
Time:      ~28 min per seed for Reach (Linux GPU), ~56 min per seed for Push (Linux GPU
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/sac_her_FetchReach-v4_seed{0,1,2}.zip
  checkpoints/sac_her_FetchReach-v4_seed{0,1,2}.meta.json
 checkpoints/sac_FetchReach-v4_seed{0,1,2}.zip       (no-HER baselines)
  checkpoints/sac_her_FetchPush-v4_seed{0,1,2}.zip
  checkpoints/sac_her_FetchPush-v4_seed{0,1,2}.meta.json
 checkpoints/sac_FetchPush-v4_seed{0,1,2}.zip        (no-HER baselines)
  results/ch04_fetchreach-v4_comparison.json
  results/ch04_fetchpush-v4_comparison.json

Results summary (what we got):

  FetchReach-v4 (sparse):
                   HER            no-HER
   success_rate:  1.00 +/- 0.00   0.96 +/- 0.07   (3 seeds x 100 episodes)
    return_mean:  -1.68 +/- 0.02  -2.92 +/- 2.04
    final_dist:    17.0mm +/- 6mm  19.5mm +/- 8mm

  FetchPush-v4 (sparse):
                   HER            no-HER
   success_rate:  0.99 +/- 0.01   0.05 +/- 0.00   (3 seeds x 100 episodes)
    return_mean: -13.20 +/- 1.48  -47.50 +/- 0.00
    final_dist:    25.7mm +/- 1mm  184.5mm +/- 0mm

If your numbers differ by more than ~10%, check the
"What Can Go Wrong" section above.

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

Exercises

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

Run the fast path command and verify your results match:

```bash
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 500000 --ent-coef 0.05
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py eval \
  --ckpt checkpoints/sac_her_FetchPush-v4_seed0.zip
```

Check the eval JSON: success_rate should be >= 0.90, mean_return > -20.0. Compare training time to Chapter 4's SAC on FetchReachDense (expect approximately 20% overhead from HER relabeling).

2. (Tweak) Change n_sampled_goal.

The default is $k = 4$. Try $k = 2$ and $k = 8$:

```bash
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --her --seed 0 --total-steps 2000000 --ent-coef 0.05 \
  --n-sampled-goal 2
```

Compare success_rate at convergence. Expected: $k = 2$ may be slightly slower to converge; $k = 8$ may be slightly faster but with diminishing returns. The original paper

(Andrychowicz et al., 2017) found $k = 4$ to be a good balance, and our 120-run sweep confirmed that $k = 8$ provides only +1.2 percentage points -- within seed variance.

3. (Tweak) Replace FUTURE strategy with FINAL.

In the lab's `process_episode_with_her()`, change the default strategy from `GoalStrategy.FUTURE` to `GoalStrategy.FINAL`. Run `--demo` and compare the success fraction. Expected: FINAL produces slightly lower success fraction because all relabeled goals are the same (the episode's last achieved goal), providing less diversity. FUTURE provides more diverse relabeled goals from different timesteps, which helps the critic learn a richer value function.

4. (Extend) Visualize the data amplification effect.

Write a short script that processes 10 synthetic episodes with `process_episode_with_her()` and plots three things using matplotlib: (a) number of original vs relabeled transitions per episode, (b) success fraction before and after HER for each episode, (c) histogram of distances between achieved goals and relabeled desired goals. Use the lab's `create_synthetic_episode()` function. Expected: you should see approximately 4.2x amplification per episode, with success fractions varying based on how clustered each trajectory's achieved goals are.

5. (Challenge) SAC without HER on sparse Reach vs Push.

Run the no-HER baseline on both environments:

```bash
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchReach-v4 --seed 0 --total-steps 1000000
bash docker/dev.sh python scripts/ch04_her_sparse_reach_push.py train \
  --env FetchPush-v4 --seed 0 --total-steps 2000000 --ent-coef 0.05
```

Compare the two: Reach achieves approximately 96% without HER, Push achieves approximately 5%. Explain the difference in terms of the effective horizon T_{eff}: for Reach, random exploration reaches the goal within 2-5 steps (the gripper workspace is about 15cm and the success threshold is 5cm). For Push, the puck must be contacted AND pushed to the target, requiring 20-50 coordinated steps. The probability of a random success drops exponentially with the effective horizon. HER's benefit scales directly with this difficulty -- the harder the exploration problem, the more HER matters.