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

Vlad Prytula

2026-02-28

Contents

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

This chapter covers:

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

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

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

This chapter introduces SAC (Soft Actor-Critic), an off-policy algorithm that stores every

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

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

4.1 WHY: The sample efficiency problem

The standard RL objective (review)

Recall from Chapter 3 that standard RL maximizes expected return:

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

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

Why determinism is a problem

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

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

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

Problem 3: training instability. When the policy is nearly deterministic, small changes in value estimates cause large behavioral changes (the ”winning” action flips).

This amplifies noise in the training process and can lead to oscillating or diverging training curves.

The maximum entropy objective

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

Definition (Maximum entropy objective).

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

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

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

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

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

Grounding example. Consider a state where two actions have Q-values $Q(s, a_1) = -0.1$ and $Q(s, a_2) = -0.3$. Under the standard objective, the policy assigns all probability to a_1. Under maximum entropy with $\alpha = 0.2$, the policy assigns probabilities proportional to $\exp(Q/\alpha)$: $\pi(a_1) \propto \exp(-0.1/0.2) = \exp(-0.5) \approx 0.61$ and $\pi(a_2) \propto \exp(-0.3/0.2) = \exp(-1.5) \approx 0.22$. After normalizing: $\pi(a_1) \approx 0.73$ and $\pi(a_2) \approx 0.27$. The better action is more likely, but the worse action is not eliminated -- the policy retains the ability to discover it was wrong about a_1.

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

The Boltzmann policy

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

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

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

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

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

Off-policy learning: reusing experience

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

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

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

Here is what this means concretely. PPO collects 8,192 transitions, uses them for 10 gradient epochs, then throws them all away. SAC stores every transition in a replay buffer (capacity 1,000,000) and samples from it repeatedly. A transition collected at step 10,000 might be sampled again at step 500,000 -- the same data contributes to learning hundreds of times.

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

Automatic temperature tuning

Choosing α manually requires care. Too low and the policy stops exploring early, settling on a suboptimal solution. Too high and the policy spends time exploring randomly

instead of exploiting what it has learned. The right value depends on the task, the training stage, and the action dimensionality.

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

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

The temperature loss function is:

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

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

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

Why this matters for robotics

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

4.2 HOW: The SAC algorithm

The components

SAC maintains five networks:

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

Network	Purpose	Updates
Target Critic 2 $Q_{\bar{\phi}_2}$	Stable target for critic updates	Polyak averaging

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

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

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

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

The training loop

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

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

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

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

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

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

PPO versus SAC

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

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

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

Key hyperparameters

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

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

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

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

Figure 4.1: The SAC architecture. Transitions flow from the environment into a circular replay buffer. At each update step, a minibatch is sampled and used to update three components: the twin Q-networks (minimizing Bellman error against slow-moving target networks), the actor (maximizing Q-value plus entropy), and the temperature α

(maintaining target entropy). Target networks track the main critics via Polyak averaging ($\tau = 0.005$). (Illustrative diagram.)

4.3 Build It: Replay buffer

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

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

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

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

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

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

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

    def add(self, obs, action, reward, next_obs, done):
        self.obs[self.ptr] = obs
        self.actions[self.ptr] = action
        self.rewards[self.ptr] = reward
        self.next_obs[self.ptr] = next_obs
        self.dones[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)

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

```
                "rewards": to_t(self.rewards[idx]),
                "next_obs": to_t(self.next_obs[idx]),
                "dones": to_t(self.dones[idx])}
```

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

> **Checkpoint.** Add 100 transitions with `obs_dim=10, act_dim=4` and sample a batch of 32. You should see: `obs shape (32, 10)`, `actions shape (32, 4)`, `rewards shape (32,)`, `buf.size == 100`, `buf.ptr == 100`. If shapes are wrong, check that the numpy arrays were initialized with the correct dimensions.

> **Note on the MLP helper.** The `TwinQNetwork` and `GaussianPolicy` classes use an `MLP` helper -- a standard feedforward network with ReLU activations. It lives in the lab file (`scripts/labs/sac_from_scratch.py`) alongside the snippet regions and takes three arguments: `input_dim`, `output_dim`, and `hidden_dims` (defaulting to `[256, 256]`). If you are copying code from the chapter, you will need this class. Run `--verify` to confirm everything assembles correctly.

4.4 Build It: Twin Q-network

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

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

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

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

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

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

    def forward(self, obs, action):
```

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

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

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

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

4.5 Build It: Squashed Gaussian policy

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

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

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

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

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

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

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

class GaussianPolicy(nn.Module):
```

```
    LOG_STD_MIN, LOG_STD_MAX = -20, 2

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

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

A few design choices in this code deserve attention. Unlike PPO in Chapter 3 (which used a state-independent `log_std` parameter), SAC learns a **state-dependent standard deviation**, so the policy can be more exploratory in unfamiliar states and more precise in well-understood ones. The `LOG_STD_MIN = -20` and `LOG_STD_MAX = 2` **clamps** prevent numerical instability, since without them `log_std` can go to $-\infty$ (zero variance, division by zero) or $+\infty$ (infinite variance, useless actions). We use `rsample()` instead of `sample()` -- the **reparameterization trick** -- so that gradients flow through the sampling operation, which is essential because without it we cannot backpropagate through the actor loss. Finally, the **numerically stable log-prob** correction `2 * (log(2) - x_t - softplus(-2 * x_t))` is mathematically equivalent to `log(1 - tanh^2(x_t))` but avoids the issue where `tanh^2(x_t)` rounds to exactly 1.0 for large `x_t`, which would produce `log(0) = -inf`.

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

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

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

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

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

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

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

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

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

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

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

    with torch.no_grad():  # no gradient through targets
        next_actions, next_log_probs = policy(next_obs)
        target_q1, target_q2 = target_q_network(next_obs, next_actions)
        target_q = torch.min(target_q1, target_q2)  # pessimistic
        # Soft Bellman backup
        target = rewards + gamma * (1.0 - dones) * (
            target_q - alpha * next_log_probs)

    q1_loss = F.mse_loss(q1, target)
    q2_loss = F.mse_loss(q2, target)
```

```
    info = {"q1_loss": q1_loss.item(),
            "q2_loss": q2_loss.item(),
            "q1_mean": q1.mean().item(),
            "q2_mean": q2.mean().item(),
            "target_q_mean": target.mean().item()}
    return q1_loss + q2_loss, info
```

The `torch.no_grad()` block is critical -- we do not backpropagate through the target computation. The target networks and the policy provide fixed targets for this update step. If gradients flowed through the target, the optimization would chase a moving target, defeating the purpose of having separate target networks.

Math	Code	Meaning
$Q_{\phi_i}(s, a)$	`q1, q2`	Current Q-value predictions
$\min_j Q_{\bar{\phi}_j}(s', a')$	`torch.min(target_q1, target_q2)`	Pessimistic target (reduces overestimati
$-\alpha \log \pi(a' \mid s')$	`-alpha * next_log_probs`	Entropy bonus in target
y	`target`	The Bellman target
$(1 - d)$	`(1.0 - dones)`	Zero bootstrap at episode end

> **Checkpoint.** Create a `TwinQNetwork`, copy it to make a `target_q_network`, create a `GaussianPolicy`, and compute the loss on a random batch. You should see: `q1_loss` finite and typically below 1.0 at initialization (random networks with zero-centered targets produce small errors), `q1_mean` near 0, `target_q_mean` near 0. If the loss is very large (above 100), check that rewards are bounded and that the done mask is applied correctly.

4.7 Build It: Actor loss with entropy

The policy update maximizes Q-values while maintaining entropy. The actor loss is:

$$L(\theta) = \mathbb{E}_{s \sim \mathcal{B},\, a \sim \pi_\theta} \left[\alpha \log \pi_\theta(a \mid s) - \min_{i=1,2} Q_{\phi_i}(s, a) \right]$$

Read this as: "find actions that have high Q-values ($-Q$ is minimized) while keeping the policy spread out ($\alpha \log \pi$ penalizes low entropy)." The balance between these two objectives is controlled by α.

```
# Actor loss with entropy regularization
# (from scripts/labs/sac_from_scratch.py:actor_loss)

def compute_actor_loss(
    policy, q_network, obs, alpha=0.2,
):
    # Sample actions from current policy
    actions, log_probs = policy(obs)
```

```
    # Evaluate with Q-networks (use min for pessimism)
    q1, q2 = q_network(obs, actions)
    q_value = torch.min(q1, q2)

    # Loss = E[alpha * log_pi - Q]  (minimize)
    actor_loss = (alpha * log_probs - q_value).mean()

    info = {
        "actor_loss": actor_loss.item(),
        "entropy": -log_probs.mean().item(),  # H = -E[log pi]
        "log_prob_mean": log_probs.mean().item(),
    }
    return actor_loss, info
```

Notice that actions are sampled fresh from the current policy (not taken from the replay buffer). The Q-networks evaluate these new actions using their current weights. This is different from the critic update, which evaluated the actions that were actually taken when the transition was collected. The actor asks: "given this state, what action would I take NOW, and how good would it be?"

The entropy diagnostic `H = -E[log pi]` tracks how stochastic the policy is. At initialization, entropy is high (the policy has not learned any preferences). As training proceeds and α decreases, entropy drops and the policy becomes more deterministic. If entropy drops to near zero too early, that is worth investigating -- see What Can Go Wrong for diagnostics.

> **Checkpoint.** Compute the actor loss on random observations with `alpha=0.2`. You should see: `actor_loss` finite and positive at initialization (the policy has not learned to select high-Q actions yet), `entropy` positive (the random-initialized policy is stochastic), and `log_prob_mean` negative (probabilities are less than 1). If `actor_loss` is NaN, check that the squashed Gaussian log-prob computation in Section 4.5 is numerically stable.

4.8 Build It: Automatic temperature tuning

The actor loss in Section 4.7 uses a fixed $\alpha = 0.2$. That works for a single verification step, but in real training the right value is hard to choose ahead of time. Too low and exploration dies early. Too high and the policy wastes time on random actions. The right α depends on the task, the training stage, and the action dimensionality -- it changes as training progresses.

SAC solves this by learning α alongside the policy and critics. The idea (introduced in Section 4.1): define a target entropy $\bar{\mathcal{H}} = -\dim(\mathcal{A})$ -- for Fetch's 4D action space, $\bar{\mathcal{H}} = -4$ -- and adjust α to keep the policy's actual entropy near that target.

The temperature loss function is:

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

where $\log \pi(a \mid s)$ comes from the current policy's output (the same log-probabilities we computed in Section 4.5). The gradient pushes α in the right direction: if entropy is below the target, α increases to encourage more exploration; if entropy is above the target, α decreases to allow more exploitation.

In practice, we optimize $\log \alpha$ rather than α directly. Since $\exp(\cdot)$ is always positive, this ensures $\alpha > 0$ without needing a constrained optimizer.

```python
# Automatic temperature tuning
# (from scripts/labs/sac_from_scratch.py:temperature_loss)

def compute_temperature_loss(
    log_alpha,          # learnable log(alpha)
    log_probs,          # log pi(a|s) from policy
    target_entropy,     # target entropy (-dim(A))
):
    alpha = log_alpha.exp()

    # If log_pi > -target (entropy too low): loss positive, alpha increases
    # If log_pi < -target (entropy too high): loss negative, alpha decreases
    alpha_loss = -(alpha * (log_probs.detach()
                            + target_entropy)).mean()

    info = {"alpha_loss": alpha_loss.item(),
            "alpha": alpha.item()}
    return alpha_loss, info
```

Notice the `.detach()` on `log_probs` -- we do not want the temperature gradient to flow back through the policy network. The temperature update adjusts only α, taking the current policy's entropy as a given. The policy has its own optimizer (Section 4.7) that handles its parameters.

Math	Code	Meaning
α	`log_alpha.exp()`	Temperature (always positive)
$\log \pi(a \mid s)$	`log_probs`	Policy log-probabilities (detached)
$\bar{\mathcal{H}} = -\dim(\mathcal{A})$	`target_entropy`	Target entropy (-4 for Fetch)

Checkpoint. Initialize `log_alpha = torch.tensor(0.0, requires_grad=True)` so that α starts at 1.0. After a few optimizer steps on random log-probs, α should have changed from 1.0 -- the direction depends on whether the random policy's entropy is above or below the target. The key check is that α remains positive (it always will, since we optimize $\log \alpha$). If `alpha_loss` is NaN, check that `log_probs` contains finite values.

Figure 4.2 shows what automatic temperature tuning looks like over a full training run. Early in training, when the policy is uncertain and entropy is naturally high, α is relatively large (around 0.47 at 30k steps). As the policy learns and becomes more deterministic, α drops -- reaching 0.0004 by 1M steps. The policy transitions from exploratory to exploitative without any manual schedule.

Entropy coefficient alpha over training steps, starting around 0.47 and decreasing to near 0.0004 over 1M steps, with annotations showing exploration phase (high alpha) and exploitation phase (low alpha)

Figure 4.2: Entropy coefficient α over training on FetchReachDense-v4. The automatic temperature tuning starts α around 0.47 (exploration phase) and drives it to 0.0004 (exploitation phase) as the policy converges. This happens without any manual schedule -- the dual gradient descent adjusts α to maintain the target entropy of $\bar{\mathcal{H}} = -4$. (Generated from TensorBoard logs in `runs/sac/FetchReachDense-v4/seed0/`, metric `replay/ent_coef`.)

4.9 Build It: SAC update -- wiring it together

We now have six components: replay buffer, twin Q-networks, squashed Gaussian policy, critic loss, actor loss, and temperature loss. The SAC update function wires them into a single gradient step. The sequence matters:

1. **Update Q-networks:** minimize Bellman error (uses current policy for next-action sampling)
2. **Update policy:** maximize Q-value plus entropy (uses updated Q-networks)
3. **Update temperature:** maintain target entropy (uses updated policy's log-probs)
4. **Soft-update target networks:** Polyak averaging $\bar{\phi} \leftarrow \tau\phi + (1-\tau)\bar{\phi}$

The first listing shows the three gradient updates -- Q-networks, policy, and temperature -- each using the loss functions we built in Sections 4.6-4.8:

```
# SAC update: gradient steps
# (from scripts/labs/sac_from_scratch.py:sac_update)

def sac_update(
    policy, q_network, target_q_network, log_alpha,
    policy_optimizer, q_optimizer, alpha_optimizer,
    batch, target_entropy, gamma=0.99, tau=0.005,
):
    alpha = log_alpha.exp().item()

    # 1. Q-network update
    q_loss, q_info = compute_q_loss(
        q_network, target_q_network, policy,
        batch, gamma, alpha)
    q_optimizer.zero_grad()
    q_loss.backward()
    q_optimizer.step()

    # 2. Policy update
    actor_loss, actor_info = compute_actor_loss(
        policy, q_network, batch["obs"], alpha)
    policy_optimizer.zero_grad()
```

```
    actor_loss.backward()
    policy_optimizer.step()

    # 3. Temperature update
    with torch.no_grad():
        _, log_probs = policy(batch["obs"])
    alpha_loss, alpha_info = compute_temperature_loss(
        log_alpha, log_probs, target_entropy)
    alpha_optimizer.zero_grad()
    alpha_loss.backward()
    alpha_optimizer.step()
```

Ordering matters. The Q-networks update first because the actor uses Q-values to evaluate its actions. The temperature updates last because it needs the updated policy's log-probs. Each component has its own Adam optimizer with learning rate 3e-4 -- coupling them into one optimizer would mix gradients that serve different objectives. We re-run the policy under `torch.no_grad()` to get fresh log-probs for the temperature update, because the policy parameters changed in step 2.

After the three gradient steps, the target networks blend in the new critic weights via Polyak averaging:

```
    # 4. Target network update (Polyak averaging)
    with torch.no_grad():
        for p, tp in zip(q_network.parameters(),
                         target_q_network.parameters()):
            tp.data.copy_(tau * p.data + (1 - tau) * tp.data)

    return {**q_info, **actor_info, **alpha_info}
```

This is a direct weighted blend of parameters, not a gradient step. It runs in `torch.no_grad()` and takes negligible time. The target network inherits 0.5% of the main network's weights per update ($\tau = 0.005$), providing the stable optimization targets that the critic loss (Section 4.6) depends on.

> **Checkpoint.** Initialize all networks, create three Adam optimizers, and run 20 updates on a random batch. You should see: `alpha` has changed from 1.0 (it may go up or down depending on the random policy's entropy relative to the target), `q1_loss` is finite, `actor_loss` is finite, and all values in the info dictionary are finite. If any value is NaN after 20 updates, check that each optimizer is attached to the correct parameters and that no gradients flow where they should not (target computation, temperature log-probs).

Quick verification: all seven components at once

Before running the full demo, verify that all seven components assemble and produce sane values:

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

Expected output:

```
============================================================
SAC From Scratch -- Verification
============================================================
Q-network:  Q1 mean ~0.04, Q2 mean ~0.06 (near zero at init)
Policy:     actions in [-0.99, 0.97], log_prob mean -2.63
SAC update: Q1 loss 2.35, actor loss -4.19, alpha 1.00 -> 0.99
            All values finite after 20 updates.
[PASS] All checks passed.
============================================================
```

This runs in under 30 seconds on CPU. If any value is NaN or if shapes are wrong, go back to the component that failed -- the error messages point to the specific check.

The demo: SAC solves Pendulum from scratch

The wired-up implementation can solve a real continuous control task. Run (~5 minutes on CPU, ~2 minutes on GPU):

```
bash docker/dev.sh python scripts/labs/sac_from_scratch.py --demo --steps 50000
```

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

Step	Avg Return	Alpha	What's happening
5,000	-843	0.415	Learning started, still mostly random
10,000	-190	0.197	Solved (threshold: -200)
25,000	-141	0.048	Refining swing-up behavior
50,000	-134	0.017	Stable, nearly deterministic policy

The policy solves Pendulum by step 10,000 -- well within 50k. The temperature drops from 0.415 to 0.017, showing the automatic tuning at work: early exploration gives way to late exploitation without any manual schedule. Figure 4.3 shows the learning curve.

This is the same algorithm SB3 runs. The difference is pedagogical clarity, not algorithmic. Every gradient step you see in SB3's TensorBoard traces back to the functions you just built.

Learning curve from the SAC from-scratch demo on Pendulum-v1: average return on the y-axis improves from around -1200 to above -200 over 50k training steps on the x-axis, with a dashed line at -200 marking the solved threshold

Figure 4.3: Learning curve from the from-scratch SAC implementation on Pendulum-v1. The average return improves from random (~-1200) to solved (above -200) within 10k steps, then refines further to ~-134 by 50k steps. The dashed line marks the -200 solved threshold. This validates that all seven components -- replay buffer, twin

Q-networks, squashed Gaussian policy, critic loss, actor loss, temperature tuning, and the wiring -- are correct. (Generated by `python scripts/labs/sac_from_scratch.py --demo --steps 50000`.)

4.10 Bridge: From-scratch to SB3

We have built SAC from scratch and verified each component. SB3 implements the same math in a production library. Before we use SB3 for the real training run, let's confirm that the two implementations agree on the core computation.

The key quantity to compare is the squashed Gaussian log-probability -- the mathematical object that appears in the actor loss (Section 4.7), the critic target (Section 4.6), and the temperature loss (Section 4.8). If the log-probs match, the losses match, and the gradients point in the same direction.

The bridging proof feeds the same random observations and pre-squash samples through our `GaussianPolicy.forward()` and through SB3's `SquashedDiagGaussianDistribution`. Both apply tanh squashing and compute log-probabilities, but with slightly different numerical formulas:

- **Our implementation:** `log_prob = dist.log_prob(x_t) - 2*(log(2) - x_t - softplus(-2*x_t))`
- **SB3's implementation:** `log_prob = dist.log_prob(x_t) - log(1 - tanh(x_t)^2 + 1e-6)`

The formulas are mathematically equivalent for the squashing correction. The ~0.02 nat typical difference comes from SB3's `1e-6` epsilon safety term, which prevents `log(0)` near the boundaries but introduces a small bias.

Run the comparison:

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

Expected output:

```
============================================================
SAC From Scratch -- SB3 Comparison
============================================================
Max abs log_prob diff: 2.055e-02
Tolerance (atol):      5.0e-02

[PASS] Our squashed Gaussian log_prob matches SB3
```

The match is within tolerance (0.02 < 0.05). For non-saturated actions -- the vast majority during training -- both formulas agree much more closely. The epsilon matters only when $\left|\tanh(u)\right| \to 1$, which happens at the action boundaries.

What SB3 adds beyond our from-scratch code

Our implementation handles one environment and stores transitions in a simple numpy buffer. SB3 adds engineering features that matter for real training:

- **Vectorized environments.** `n_envs=1` (SAC default in SB3) with `SubprocVecEnv` or `DummyVecEnv`. Unlike PPO which benefits heavily from parallel environments, SAC's off-policy nature means a single environment can fill the replay buffer adequately.
- **Efficient replay buffer.** SB3's replay buffer handles dictionary observations natively (the `observation`, `achieved_goal`, and `desired_goal` keys from Fetch), stores them efficiently in numpy arrays, and supports batched sampling.
- **`MultiInputPolicy`.** SB3 builds separate encoders for each key in the dictionary observation and concatenates them. Our from-scratch code assumed a flat observation vector.
- **Gradient clipping.** SB3 clips gradient norms by default, preventing a single bad batch from causing an explosively large update.
- **Automatic entropy coefficient** with constrained optimization -- the same math as our `compute_temperature_loss`, but with additional numerical safeguards.

Mapping SB3 TensorBoard metrics to our code

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

SB3 TensorBoard key	Our function	What it measures
`train/actor_loss`	`compute_actor_loss`	Policy loss (entropy-regularized)
`train/critic_loss`	`compute_q_loss`	Twin Q-network Bellman error
`replay/ent_coef`	`log_alpha.exp()`	Temperature α (auto-tuned)
`replay/q1_mean`	`TwinQNetwork` Q1 output	Average Q1 value in sampled batch
`replay/q2_mean`	`TwinQNetwork` Q2 output	Average Q2 value in sampled batch
`replay/q_min_mean`	`torch.min(q1, q2)`	Pessimistic Q-value estimate
`rollout/ep_rew_mean`	(environment)	Mean episode return
`rollout/success_rate`	(environment)	Fraction of episodes where goal is reached

We find this mapping useful for debugging. When you see `replay/ent_coef = 0.08` in TensorBoard, that is $\alpha = 0.08$ -- the output of the same `log_alpha.exp()` from Section 4.8. When you see `train/critic_loss = 0.34`, that is the MSE Bellman error from `compute_q_loss` in Section 4.6. Having built each component yourself, you can trace any TensorBoard metric back to the exact computation that produced it.

4.11 Run It: Training SAC on FetchReachDense-v4

> **A note on script naming.** The production script is called `ch03_sac_dense_reach.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, SAC on dense Reach is chapter 3; in this book, it is chapter 4. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch03`. This is intentional -- renaming would break the tutorial infrastructure. When you see `ch03` in a command or file path, you are running the right script for this chapter.

```
------------------------------------------------------------
EXPERIMENT CARD: SAC on FetchReachDense-v4
------------------------------------------------------------
Algorithm:    SAC (soft actor-critic, off-policy, auto-tuned entropy)
Environment:  FetchReachDense-v4
Fast path:    500,000 steps, seed 0
Time:         ~14 min (GPU) / ~60 min (CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
    --seed 0 --total-steps 500000

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

Expected artifacts:
  checkpoints/sac_FetchReachDense-v4_seed0.zip
  checkpoints/sac_FetchReachDense-v4_seed0.meta.json
  results/ch03_sac_fetchreachdense-v4_seed0_eval.json
  runs/sac/FetchReachDense-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.95
  mean_return > -5.0
  final_distance_mean < 0.03

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

Running the experiment

The one-command version:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all --seed 0
```

This runs training, evaluation, and comparison to PPO. It takes about 14 minutes on a GPU. For a quick sanity check that finishes in about 2 minutes:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py train --total-steps 1000
```

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

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py eval \
  --ckpt checkpoints/sac_FetchReachDense-v4_seed0.zip
```

Training milestones

Watch for these milestones during training. Note that `learning_starts=10000` means no training happens during the first 10k steps -- the agent collects random data to seed the replay buffer:

Timesteps	Success Rate	What's happening
0-10k	0%	Collecting random data (no training yet)
10k-50k	0-5%	Training begins, Q-values adjusting
50k-150k	20-60%	Rapid improvement, entropy decreasing
150k-400k	80-99%	Policy converging
400k-1M	100%	Fine-tuning, entropy near zero

Our test run achieved 100% success rate after approximately 300k steps, with an average goal distance of 18.6mm (the success threshold is 50mm) and throughput of approximately 594 steps per second on an NVIDIA GB10.

Reading TensorBoard

Launch TensorBoard to watch training in real time:

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

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

Metric	Expected behavior	What to watch for
`rollout/ep_rew_mean`	Steadily increasing (less negative)	Should move from around -20
`rollout/success_rate`	0 -> 1 over training	The primary success metric
`replay/ent_coef`	Starts high (~0.47), decreases gradually	Should NOT drop to near-zero
`replay/q_min_mean`	Stabilizes in a reasonable range	Should NOT grow unbounded
`train/actor_loss`	Fluctuates, generally decreases	NaN means numerical instabil
`train/critic_loss`	Starts high, decreases and stabilizes	Tracks learning progress

These are the same quantities from the Build It sections: `replay/ent_coef` maps to `log_alpha.exp()`, `train/critic_loss` maps to `compute_q_loss`, and `replay/q_min_mean` maps to `torch.min(q1, q2)`. The TensorBoard metric mapping table in Section 4.10 has the full correspondence.

> **Tip.** Low GPU utilization (~5-10%) is expected. The bottleneck is CPU-bound MuJoCo simulation, not neural network operations. With small networks (two 256-unit hidden layers) and batch sizes (256), GPU operations complete in microseconds while the CPU runs physics. This is normal for state-based RL -- pixel-based policies in later chapters will use the GPU much more heavily.

SAC versus PPO: results comparison

Both algorithms solve FetchReachDense-v4, but with different tradeoffs. Figure 4.4 compares their learning curves.

Metric	PPO (Chapter 3)	SAC (this chapter)
Success Rate	100%	100%
Mean Return	-0.40	-1.06
Final Distance	4.6mm	18.6mm
Action Smoothness	1.40	1.68
Training Time	~6 min	~28 min
Throughput	~1,300 fps	~594 fps

Both achieve 100% success -- the off-policy stack is validated. SAC has a higher final distance (18.6mm vs 4.6mm), though still well within the 50mm threshold, and it is slower in wall-clock time because it updates more networks per step (actor + two critics + two targets versus PPO's single shared network). The slightly rougher actions (smoothness 1.68 vs 1.40) reflect the entropy bonus encouraging a spread of actions during training.

Both algorithms solve this task completely, so the interesting comparison is structural: PPO used each transition once and discarded it, while SAC stored every transition and reused it across many updates. On FetchReachDense, where signal is plentiful, this difference is a mild efficiency trade. On sparse-reward tasks (Chapter 5), the replay buffer becomes the key enabler for learning.

Learning curve comparison: PPO and SAC success rate over timesteps on FetchReachDense-v4, both reaching 100% but PPO reaching it somewhat earlier in wall-clock time while SAC has a different convergence profile

Figure 4.4: Learning curve comparison -- PPO (Chapter 3) versus SAC (this chapter) on FetchReachDense-v4. Both reach 100% success rate. PPO converges somewhat earlier measured by timesteps because its per-step updates are lighter. SAC converges to a stable solution with the replay buffer machinery that Chapter 5 requires. (Generated by extracting `rollout/success_rate` from TensorBoard logs in `runs/ppo/FetchReachDense-v4/seed0/` and `runs/sac/FetchReachDense-v4/seed0/`.)

Verifying results

After training completes, check the evaluation JSON:

```
cat results/ch03_sac_fetchreachdense-v4_seed0_eval.json | python -m json.tool | hea
```

Key fields to verify:

```
{
  "aggregate": {
    "success_rate": 1.0,
```

```
    "return_mean": -1.06,
    "final_distance_mean": 0.019
  }
}
```

The passing criteria are: success rate above 95%, mean return above -5.0, and final distance below 0.03 meters. Our runs consistently exceed these thresholds. If yours do not, see What Can Go Wrong below.

4.12 What can go wrong

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

`replay/q_min_mean` grows unbounded (above 100 and still rising)

Likely cause. Overestimation feedback loop: Q-targets use overestimated Q-values, which train the critic to overestimate further, which produces even higher targets. The twin Q-network minimum (Section 4.4) is supposed to prevent this, but it can still happen if target networks update too fast or rewards are misconfigured.

Diagnostic. Check three things: (1) rewards are bounded -- FetchReachDense should produce rewards in $[-1, 0]$; (2) target networks update with small τ (0.005, not 0.5 or 1.0) -- verify the Polyak averaging is actually running; (3) try reducing the learning rate from 3e-4 to 1e-4 to slow down the overestimation spiral.

`replay/ent_coef` drops to less than 0.01 within the first 10k steps

Likely cause. The policy collapsed to near-deterministic before learning anything useful. This can happen if the target entropy is too low or if the initial policy variance is too small.

Diagnostic. Check the target entropy setting -- it should be $-\dim(\mathcal{A}) = -4$ for Fetch. Try temporarily using a fixed entropy coefficient (`ent_coef=0.2` in SB3) to isolate whether the issue is auto-tuning or something else. If the fixed coefficient works, the auto-tuning's initial learning rate may be too aggressive -- try reducing the alpha learning rate to 1e-4.

Success rate stalls below 50% for more than 200k steps

Likely cause. Insufficient exploration or replay buffer sampling issues.

Diagnostic. Check the entropy coefficient -- if it is very low (below 0.01) early in training, the policy may have stopped exploring. Verify that `learning_starts` is not set too high (default 10,000 is appropriate). Check that `batch_size` is reasonable (256) -- too small means high-variance updates, too large means slow updates.

Training much slower than expected (below 200 fps on GPU)

Likely cause. GPU not in use, or excessive gradient steps per environment step.

Diagnostic. Check `nvidia-smi` inside the container -- a python process should be using GPU memory. Verify `gradient_steps=1` (the default, meaning one gradient step per environment step). Note that approximately 594 fps is typical for SAC on FetchReach with a GPU -- the bottleneck is CPU-bound MuJoCo simulation, not the neural networks.

Q1 and Q2 diverge significantly (difference greater than 10x)

Likely cause. One Q-network is stuck or has a different effective learning rate.

Diagnostic. Verify that both Q-networks share the same optimizer -- our `compute_q_loss` returns `q1_loss + q2_loss` and both should receive gradients through a single `q_optimizer.step()`. If you accidentally created separate optimizers for Q1 and Q2, check that both are being stepped.

`ep_rew_mean` flatlines near -20 for the entire run

Likely cause. Wrong environment or the policy is not receiving goal information.

Diagnostic. Check the env_id is `FetchReachDense-v4` (not the sparse variant `FetchReach-v4`, which produces rewards of exactly 0 or -1). Verify that SB3 is using `MultiInputPolicy` -- this handles the dictionary observation structure. Print `obs.keys()` to confirm you see `observation`, `achieved_goal`, and `desired_goal`.

`--compare-sb3` shows log-prob difference above 0.05

Likely cause. The squashing correction formula differs in a way that matters.

Diagnostic. Verify that your implementation uses the numerically stable formula: `2*(log(2) - x_t - softplus(-2*x_t))`. This is mathematically equivalent to `log(1 - tanh^2(x_t))` but avoids `log(0)` at the boundaries. If the difference is between 0.02 and 0.05, it is likely harmless -- the discrepancy comes from SB3's epsilon term and only affects near-saturated actions.

Build It `--verify` reports NaN in Q-loss or actor loss

Likely cause. Numerical instability in the squashed Gaussian log-probability computation.

Diagnostic. Check the `LOG_STD_MIN` and `LOG_STD_MAX` bounds in `GaussianPolicy`. Without them, the log standard deviation can go to $-\infty$ (producing zero variance and division by zero) or $+\infty$ (producing infinite variance and meaningless actions). The safe range is $[-20, 2]$. Also verify that the softplus-based squashing correction is used instead of the naive `log(1 - tanh^2(x))`, which produces $-\infty$ when `tanh(x)` rounds to exactly ± 1.

4.13 Summary

This chapter derived SAC from the maximum entropy objective and built it from the ground up. Here is what you accomplished:

- **The maximum entropy objective.** Instead of maximizing reward alone, SAC maximizes reward plus an entropy bonus: $J_{\text{MaxEnt}} = \mathbb{E}[\sum_t \gamma^t (r_t + \alpha \mathcal{H}(\pi))]$. This keeps the policy exploratory during training and lets it become deterministic as it converges. The optimal policy assigns probabilities proportional to exponentiated Q-values -- a Boltzmann distribution.
- **Off-policy learning.** SAC stores every transition in a replay buffer and reuses it across many updates, unlike PPO which discards data after each update. This makes SAC more sample-efficient and -- crucially -- enables the goal relabeling that Chapter 5 requires.
- **From-scratch implementation.** You built seven components: the replay buffer, twin Q-networks, squashed Gaussian policy, twin Q-network loss (soft Bellman backup), actor loss with entropy, automatic temperature tuning, and the full update step. Each was verified with concrete checks before wiring them together.
- **Bridge to SB3.** The bridging proof confirmed that our squashed Gaussian log-probability matches SB3's distribution implementation within 0.02 nats. SB3 adds engineering features (efficient replay storage, dictionary observation handling, gradient clipping) but computes the same underlying math.
- **Pipeline validation.** SAC achieved 100% success rate on FetchReachDense-v4, matching PPO's performance. The entire off-policy stack -- replay buffer, twin critics, automatic temperature tuning, Polyak averaging -- is validated and working.

That last point is the foundation for what comes next. Chapter 5 introduces Hindsight Experience Replay (HER), which tackles sparse rewards -- first on Reach (where the improvement is marginal) and then on Push (where it is transformative, lifting success from 5% to 99%). With sparse rewards, the agent receives a signal of either 0 (success) or -1 (failure) -- no distance information, no gradient toward the goal. Almost every episode is a failure, and failures carry no information about how to improve.

HER solves this by asking a simple question after each failed episode: ”I did not reach the desired goal, but what goal DID I reach?” It then relabels the transitions in the replay buffer with the achieved goal, turning a failure into a (synthetic) success. This goal relabeling requires two things that you now have: (1) an off-policy algorithm that can learn from modified transitions, and (2) a replay buffer where those transitions are stored and accessible for relabeling. The off-policy machinery you built in this chapter is the foundation HER needs.

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
```

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

  for seed in 0 1 2; do
    bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
      --seed $seed --total-steps 1000000
  done

Hardware:    Any machine with Docker (GPU optional; tested on NVIDIA GB10)
Time:         ~28 min per seed (Linux GPU), ~120 min per seed (Mac/CPU)
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/sac_FetchReachDense-v4_seed{0,1,2}.zip
  checkpoints/sac_FetchReachDense-v4_seed{0,1,2}.meta.json
  results/ch03_sac_fetchreachdense-v4_seed{0,1,2}_eval.json
  runs/sac/FetchReachDense-v4/seed{0,1,2}/

Results summary (what we got):
  success_rate: 1.00 +/- 0.00  (3 seeds x 100 episodes)
  return_mean:  -0.93 +/- 0.13
  final_distance_mean: 0.016 +/- 0.003

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

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

Exercises

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

Run the fast path command and verify your results match:

```
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all \
  --seed 0 --total-steps 500000
```

Check the eval JSON: `success_rate` should be >= 0.95, `mean_return` > -5.0. Record your training time and steps per second for comparison with Chapter 3 (PPO). Expected: approximately 594 fps (versus PPO's approximately 1,300 fps), due to more network updates per step.

2. (Tweak) Twin Q-network ablation.

In the lab's `compute_q_loss()`, the target uses `torch.min(target_q1, target_q2)`. Change it to use only one Q-network: `target_q = target_q1`. Run `--verify` and observe: does `q1_mean` grow larger without the min clipping? Run `--demo --steps`

`50000` and compare final returns. Expected: without the min trick, Q-values may overestimate, leading to slightly worse or unstable training.

3. (Tweak) Fixed versus auto-tuned temperature.

In the lab's `sac_update()`, replace `alpha = log_alpha.exp().item()` with a fixed value: `alpha = 0.2`. Run `--demo --steps 50000` with fixed versus auto-tuned. Compare: (a) final return, (b) convergence speed, (c) policy entropy at the end. Expected: auto-tuned reaches better final performance because it reduces α as the policy improves; fixed $\alpha = 0.2$ may over-explore late in training.

4. (Extend) Add Q-value divergence monitoring.

In the verification code, add a check that tracks `|q1_mean - q2_mean|` over updates. Plot this divergence over the 20 verification steps. Expected: the two Q-networks should stay close (within approximately 0.5 of each other) because they see the same targets. Large divergence would indicate a problem -- for instance, if only one Q-network receives gradient updates.

5. (Challenge) SAC on Pendulum with different target entropies.

Modify the demo to try target entropy values of -0.5, -1.0 (default for 1D), and -2.0. For each, train for 50k steps and record: (a) final average return, (b) final alpha value, (c) steps to reach -200 return. Expected: lower target entropy leads to earlier exploitation (faster convergence but potentially less robust). Higher target entropy leads to more exploration (slower convergence but the policy may discover better strategies). This trade-off is the heart of the maximum entropy objective.