Contents

1 Chapter 2: PPO on Dense Reach -- The Pipeline Truth Serum

1.1 What This Chapter Is Really About

Before we dive into math, let's be honest about what we're doing here.

You're about to train a neural network to control a robot arm. The arm will learn to reach arbitrary 3D positions--not because someone programmed the kinematics, but because it tried millions of times and gradually figured out what works.

The result: A neural network that figured out how to reach any point in 3D space.

Robot reaching goals

No inverse kinematics. No trajectory planning. The robot learned this through 500,000 training steps--watch the distance counter drop to zero.

That's remarkable. But here's the uncomfortable truth: **most RL implementations don't work on the first try.** The field has a reproducibility crisis (Henderson et al., 2018), hyperparameters matter more than they should, and small bugs can silently cause complete failure.

This chapter is about building confidence that your infrastructure is correct *before* you add complexity. We use **Proximal Policy Optimization (PPO)**--a widely-used RL algorithm that learns by repeatedly trying actions and adjusting based on outcomes--on a dense-reward task as a **diagnostic**, so that failure becomes informative: if this doesn't work, the problem is in your setup, not your algorithm.

By the end, you will have a trained policy achieving >90% success rate (we got 100% in our test run), an understanding of *why* PPO works (not just *that* it works), diagnostic skills to identify common training failures, and the confidence to move to harder problems.

If you want to run first and read later, jump to Part 3:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all --seed 0
```

1.2 Part 0: Setting the Stage

1.2.1 0.1 The Problem We're Solving

Imagine you want to teach a robot arm to touch a target. You could:

Option A: Program it explicitly. Compute inverse kinematics, plan a trajectory, execute. This works but requires knowing the robot's geometry precisely and doesn't generalize to new situations.

Option B: Let it learn. Show the robot where the target is, let it flail around, reward it when it gets close. Over time, it figures out how to reach any target.

Option B is reinforcement learning. It's harder to get working, but once it works, the same algorithm can learn to push objects, pick things up, even walk--without you programming each behavior explicitly.

1.2.2 0.2 Why Start Here?

The Fetch robot in simulation has 7 joints and a gripper. The full task hierarchy looks like:

Task	Difficulty	What Makes It Hard
Reach	Easiest	Just move the end-effector to a point
Push	Medium	Must contact and move an object
Pick & Place	Hard	Must grasp, lift, and place accurately

We start with Reach because it isolates the core RL problem: learning a mapping from observations to actions. No object dynamics, no contact physics, no grasp planning. Just: "see goal, move there."

And we use **dense rewards** (negative distance to goal) rather than sparse rewards (success/failure only). This gives the learning algorithm continuous feedback--every action either improves or worsens the situation.

1.2.3 0.3 The Diagnostic Mindset

Here's a scenario that happens more often than anyone admits:

> You implement an advanced algorithm on a difficult task. Train for 10 hours. Success rate: 0%. What went wrong?

The honest answer: **you have no idea.** It could be an environment misconfiguration, a wrong network architecture, bad hyperparameters, a bug in your algorithm implementation, the task simply needing more training time, or a subtle numerical issue. You're debugging in the dark with too many variables.

The solution: Establish a baseline where failure is informative. Start with the simplest algorithm (PPO) on the easiest task (Reach with dense rewards). If this doesn't work, the problem is in your infrastructure, not your algorithm choice.

1.3 Part 1: WHY -- Understanding the Learning Problem

1.3.1 1.1 What Are We Actually Optimizing?

Let's build up the math from intuition, defining each symbol as we introduce it.

The Setup: At each timestep t, our **policy** π (a neural network with parameters θ) sees the current state s_t and the goal g. It is convenient to bundle these into a single goal-conditioned input:

```
x_t := (s_t, g)
```

The policy outputs an action $a_t \sim \pi_\theta(\cdot \mid x_t)$. The environment responds with a new state s_{t+1} and a **reward** r_t--a single number indicating how good that transition was.

Before stating the objective, we need three definitions:

Definition (Reward). The reward $r_t \in \mathbb{R}$ is the immediate feedback signal at timestep t. In FetchReachDense, $r_t = -\|p_t - g\|_2$ where p_t is the gripper position and g is the goal. More negative means farther from the goal; zero means perfect.

Definition (Discount Factor). The discount factor $\gamma \in [0, 1)$ determines how much we value future rewards relative to immediate rewards. A reward of magnitude r received k steps in the future contributes $\gamma^k r$ to our objective. With $\gamma = 0.99$, a reward 100 steps away is worth $0.99^{100} \approx 0.37$ as much as an immediate reward. This captures two intuitions: (1) sooner is better than later, and (2) distant rewards are more uncertain.

Definition (Time Horizon). The horizon T is the maximum number of timesteps in an episode. For FetchReach, $T = 50$ steps (we index $t = 0, \ldots, T-1$).

The Objective: Find policy parameters θ that maximize the **expected discounted return**:

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

Here $J(\theta)$ is the objective function we seek to maximize--it measures how good a policy with parameters θ is, averaged over many episodes. The sum $\sum_{t=0}^{T-1} \gamma^t r_t$ is called the **return**: the total reward accumulated over an episode, with future rewards discounted by γ.

The expectation is over trajectories--different runs give different outcomes because actions sample from the policy distribution and the environment may be stochastic.

The Challenge: How do you take a gradient of this? The expectation depends on θ in a complicated way--θ determines the policy, which determines the actions, which determines the states visited, which determines the rewards.

1.3.2 1.2 The Policy Gradient Theorem (Intuition First)

Here's the key insight, stated informally:

> **To improve the policy, increase the probability of actions that led to better-than-expected outcomes, and decrease the probability of actions that led to worse-than-expected outcomes.**

The "better-than-expected" part is crucial. An action that got reward +10 isn't necessarily good--if you typically get +15 from that state, it was actually a bad choice.

This is captured by the **advantage function**:

```
A(x, a) = Q(x, a) - V(x)
```

where:

- $Q(x, a)$ = expected return if you take action a in goal-conditioned state x, then follow your policy
- $V(x)$ = expected return if you just follow your policy from goal-conditioned state x

So $A(x, a) > 0$ means action a was better than average; $A(x, a) < 0$ means it was worse.

The Policy Gradient:

```
\nabla_\theta J(\theta) = \mathbb{E}\left[ \sum_t \nabla_\theta \log \pi_\theta(a_t \mid x
```

Read this as: "Adjust θ to make good actions more likely and bad actions less likely, weighted by how good/bad they were."

1.3.3 1.3 The Problem: Policy Gradient Is Unstable

In theory, you can just follow this gradient and improve. In practice, **vanilla policy gradient is notoriously unstable**, for two reinforcing reasons.

First, advantage estimates are noisy. We don't know the true advantage--we estimate it from sampled trajectories, and with a finite batch these estimates have high variance. Sometimes they're way off, which means we make bad updates.

Second, big updates break everything. Suppose we estimate that some action is great ($A >> 0$) and crank up its probability. If our estimate was wrong, we've now committed to a bad action. Worse, the new policy visits different states, which makes our old advantage estimates invalid, so the whole thing can spiral into catastrophic collapse.

This isn't hypothetical--it happens all the time. Training curves that look promising suddenly crash to zero and never recover.

1.3.4 1.4 PPO's Solution: Constrained Updates

PPO's key idea: **don't change the policy too much in one update.**

But "too much" in what sense? Not in parameter space (a small parameter change can cause large behavior change). Instead, in **probability space**.

Define the likelihood ratio:

```
\rho_t(\theta) = \frac{\pi_\theta(a_t \mid x_t)}{\pi_{\theta_{\text{old}}}(a_t \mid x_t)}
```

This measures how the action likelihood changed: $\rho = 1$ means the likelihood is unchanged, $\rho = 2$ means the action is now twice as likely, and $\rho = 0.5$ means it is now half as likely.

Note (continuous actions). For Fetch, actions are continuous, so $\pi_\theta(a \mid x)$ is a probability density. The ratio is still well-defined as a likelihood ratio, and implementations compute it via log-probabilities: $\rho_t = \exp(\log \pi_\theta(a_t \mid x_t) - \log \pi_{\text{old}}(a_t \mid x_t))$.

PPO clips this ratio to stay in $[1 - \epsilon, 1 + \epsilon]$ (typically $\epsilon = 0.2$):

```
L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[ \min\left( \rho_t A_t, \text{clip}(\rho_t, 1
```

What this does:

Advantage	Gradient wants to...	Clipping effect
$A > 0$ (good action)	Increase ρ (make action more likely)	Stops at $\rho = 1.2$
$A < 0$ (bad action)	Decrease ρ (make action less likely)	Stops at $\rho = 0.8$

The policy can improve, but only within a "trust region" around its current behavior. This prevents the catastrophic updates that kill vanilla policy gradient.

1.3.5 1.5 A Concrete Example

Let's trace through one update to make this concrete.

Setup (discrete intuition): Imagine a discrete action space. The old policy assigns probability 0.3 to action a in goal-conditioned state x. We estimate the advantage is $A = +2$ (this was a good action).

Naive approach: The gradient says "make this action more likely!" So we update and now $\pi_\theta(a \mid x) = 0.6$.

Problem: The ratio $\rho = 0.6/0.3 = 2.0$. We doubled the probability in one update. If our advantage estimate was wrong, we've made a big mistake.

PPO's approach: The clipped objective computes:

- Unclipped: $\rho \cdot A = 2.0 \times 2 = 4.0$
- Clipped: $\text{clip}(2.0, 0.8, 1.2) \times 2 = 1.2 \times 2 = 2.4$
- Objective: $\min(4.0, 2.4) = 2.4$

The gradient only flows through the clipped version. We still increase the action probability, but the update is bounded. We can't go from 0.3 to 0.6 in one step--we'd need multiple updates, each constrained.

1.3.6 1.6 Why Dense Rewards Matter Here

FetchReachDense-v4 gives reward $R = -\|achieved - goal\|_2$ at every step. This is the negative distance to the goal.

This helps because every action provides signal ("you got 2cm closer" or "you drifted 1cm away"), which means the learning algorithm always has gradient information and exploration isn't a bottleneck--even random actions provide useful data. Compare this to sparse rewards ($R = 0$ if success, -1 otherwise), where you only learn when you succeed, a random policy almost never succeeds, and most of your data is uninformative.

Dense rewards therefore decouple the exploration problem from the learning problem. If PPO fails on dense Reach, the issue is definitely in your implementation, not in insufficient exploration.

1.4 Part 2: HOW -- The Algorithm in Detail

1.4.1 2.1 The Actor-Critic Architecture

PPO maintains two neural networks:

Actor $\pi_\theta(a \mid x)$: Given the goal-conditioned input $x = (s, g)$, output a probability distribution over actions. For continuous actions, this is typically a Gaussian with learned mean and standard deviation.

Critic $V_\phi(x)$: Given the same input, estimate the expected return. This helps compute advantages.

Why two networks, not one? A natural question: why not have a single network output both actions and value estimates? The short answer is that the two objectives pull in different directions. The actor maximizes expected return (it wants to find good actions), while the critic minimizes prediction error (it wants accurate value estimates), and these gradients can conflict so that improving one hurts the other. The output types also differ--the actor produces a probability distribution (mean and variance for continuous actions) whereas the critic produces a single scalar, so forcing both through the same final layers creates unnecessary coupling. Finally, separation improves stability: since the critic's value estimates feed into advantage computation, which in turn trains the actor, any destabilization of the critic makes advantages noisy, which destabilizes the actor further--a vicious cycle that separate networks help prevent.

> "A geometric interpretation involves representation learning and function factorization. For our purposes, the property we need is simple: sharing early layers but splitting actor/critic heads improves stability by reducing destructive interference between objectives."

Consider what we're learning: a mapping from goal-conditioned inputs to "optimal behavior," encompassing both *what to do* (policy) and *how good is this state* (value). Naively, this is a single map:

$$F : \mathcal{S} \times \mathcal{G} \to \mathcal{P}(\mathcal{A}) \times \mathbb{R}$$

Actor-critic separates this into two maps:

$$\pi : \mathcal{S} \times \mathcal{G} \to \mathcal{P}(\mathcal{A}) \quad \text{and} \quad V : \mathcal{S} \times \mathcal{G} \to \mathbb{R}$$

This *suggests* a factorization--perhaps recognizing product structure in the target space, or projecting through a lower-dimensional "behaviorally-relevant" manifold. The shared backbone makes this more concrete: we learn $\phi : \mathcal{S} \times \mathcal{G} \to \mathcal{Z}$ (a representation), then compose with separate heads. This is genuinely factoring through an intermediate space.

However, the analogy is imperfect. Unlike clean mathematical factorizations, V depends on π (it's V^π), so the components are coupled. The precise geometric interpretation--if one exists--remains to be clarified. What's clear is that the separation has practical benefits; whether it reflects deep structure or is merely a useful engineering heuristic is an open question.

In practice, implementations often share early layers (a "backbone") with separate final layers ("heads"). This captures shared features while keeping the objectives separate. Stable Baselines 3 uses this approach by default.

1.4.2 2.2 The Training Loop

```
repeat:
    1. Collect N steps using current policy
    2. Compute advantages using critic
    3. Update actor using clipped objective (multiple epochs)
    4. Update critic using MSE loss on returns
    5. Discard data, go to 1
```

Step 1: Collect Data

Run the policy for `n_steps` in each of `n_envs` parallel environments. This gives us `n_steps * n_envs` transitions to learn from.

Step 2: Compute Advantages

We use Generalized Advantage Estimation (GAE), which balances bias and variance:

```
\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma \lambda)^l \delta_{t+l}
```

where the TD residual (with termination masking) is:

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

Here $d_t \in \{0, 1\}$ indicates whether the episode terminated at timestep t (if it did, we do not bootstrap from x_{t+1}). The value terms are computed when collecting the rollout and treated as constants while optimizing this batch.

The parameter λ interpolates between two extremes: $\lambda = 0$ gives one-step TD estimates (high bias, low variance), while $\lambda = 1$ recovers Monte Carlo returns (low bias, high variance). The typical default of $\lambda = 0.95$ sits close to Monte Carlo but gains some variance reduction from the value baseline.

Compact equation summary (goal-conditioned PPO)

```
x_t := (s_t, g)
J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1}\gamma^t r_t\right]
\delta_t = r_t + \gamma(1-d_t)V_{\text{rollout}}(x_{t+1}) - V_{\text{rollout}}(x_t)
\hat{A}_t = \sum_{l=0}^{T-t-1}(\gamma\lambda)^l\delta_{t+l}
\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)
\rho_t(\theta) = \pi_\theta(a_t \mid x_t)\,/\,\pi_{\theta_{\text{old}}}(a_t \mid x_t)
L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(\rho_t\hat{A}_t,\text{clip}(\rho_t,1
```

Steps 3-4: Update Networks

Unlike supervised learning, we do multiple passes over the same data--`n_epochs = 10` is typical for PPO, with each pass using minibatches of size `batch_size`. This reuses our expensive-to-collect trajectory data while the clipping prevents us from overfitting to it.

Step 5: Discard and Repeat

PPO is **on-policy**: we can only use data from the current policy. After updating, our data is "stale" and must be discarded.

This is inefficient compared to off-policy methods (which reuse old data), but simpler and more stable.

1.4.3 2.3 Key Hyperparameters

Parameter	Our Setting	What It Controls
`n_steps`	1024	Trajectory length before update
`n_envs`	8	Parallel environments (throughput)
`batch_size`	256	Minibatch size for gradient updates
`n_epochs`	10	Passes over data per update
`learning_rate`	3e-4	Gradient step size
`clip_range`	0.2	PPO clipping parameter (ϵ)
`gae_lambda`	0.95	Advantage estimation bias-variance
`ent_coef`	0.0	Entropy bonus (exploration)

For FetchReachDense-v4, Stable Baselines 3 defaults work well. Don't tune hyperparameters until you've verified the baseline works.

1.5 Part 2.5: BUILD IT -- From Equations to Code

This section shows how the math above maps to code. We build PPO piece by piece, verifying each component before moving to the next. We use pedagogical implementations from `scripts/labs/ppo_from_scratch.py` -- these are for understanding, not production.

1.5.1 2.5.1 The Actor-Critic Network

Before we can compute losses, we need the network they operate on. PPO uses an actor-critic architecture with a shared backbone and separate heads (discussed in Section 2.1):

```
--8<-- "scripts/labs/ppo_from_scratch.py:actor_critic_network"
```

!!! lab "Checkpoint" Instantiate the network and verify shapes:

```
```python
model = ActorCritic(obs_dim=16, act_dim=4, hidden_dim=64)
obs = torch.randn(1, 16)
dist, value = model(obs)
print(f"Action mean shape: {dist.mean.shape}")   # (1, 4)
print(f"Value shape:       {value.shape}")        # (1,)
print(f"Parameters:     {sum(p.numel() for p in model.parameters()):,}") # ~5,577
```

The network is small by design -- Fetch tasks use MLPs, not CNNs. Most of the ~5.6k parameters a
```

1.5.2 2.5.2 GAE: Computing Advantages

The advantage formula from Section 2.2:

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma(1-d_t)V_{\text{rollout}}(x_{t+1}) - V_{\text{rollout}}(x_t)$ is the TD residual with termination masking. In code, we compute this backwards through the trajectory:

--8<-- "scripts/labs/ppo_from_scratch.py:gae_computation"

Key mapping:

Math	Code	Meaning
δ_t	`delta`	TD residual: was this transition better than expected?
γ	`gamma`	Discount factor (0.99)
λ	`gae_lambda`	Bias-variance tradeoff (0.95)
$\hat{A}_t$	`advantages[t]`	How much better was this action vs. average?
d_t	`dones[t]`	Episode terminated at this step?

!!! lab "Checkpoint" Test with a simple trajectory where reward arrives only at the end:

```python
T = 10
rewards = torch.zeros(T); rewards[-1] = 1.0    # reward only at step 10
values = torch.linspace(0, 0.5, T)              # increasing value estimates
next_value = torch.tensor(0.0)
dones = torch.zeros(T); dones[-1] = 1.0         # episode ends

advantages, returns = compute_gae(rewards, values, next_value, dones)
print(f"advantages[-1]: {advantages[-1]:.3f}")   # > 0 (got unexpected reward)
print(f"All finite: {torch.isfinite(advantages).all()}")  # True
```

The last advantage should be positive because the agent received a reward it didn't fully predi

1.5.3 2.5.3 The Clipped Surrogate Loss

The PPO objective from Section 1.4:

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

In code:

--8<-- "scripts/labs/ppo_from_scratch.py:ppo_loss"

Key mapping:

Math	Code	Meaning
$\rho_t = \frac{\pi_\theta(a\|x)}{\pi_{\theta_{old}}(a\|x)}$	`ratio`	How much did action likelihood change?
ϵ	`clip_range`	Maximum allowed ratio change (0.2)
A_t	`advantages`	Advantage estimates from GAE

!!! lab "Checkpoint" When the policy hasn't changed yet, all ratios should be 1.0 and no clipping should occur:

```python
model = ActorCritic(obs_dim=4, act_dim=2)
obs = torch.randn(32, 4); actions = torch.randn(32, 2)
with torch.no_grad():
    dist, _ = model(obs)
    old_log_probs = dist.log_prob(actions).sum(dim=-1)

dist, _ = model(obs)  # same model, same params
loss, info = compute_ppo_loss(dist, old_log_probs, actions, torch.randn(32))
print(f"clip_fraction: {info['clip_fraction']:.3f}")  # 0.000 (nothing clipped)
print(f"ratio_mean:   {info['ratio_mean']:.3f}")    # 1.000 (unchanged policy)
print(f"approx_kl:     {info['approx_kl']:.6f}")      # ~0.000 (no divergence)
```

1.5.4 2.5.4 The Value Loss

The critic learns to predict expected returns. We minimize the mean squared error between the critic's predictions $V_\phi(x_t)$ and the computed return targets $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$:

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

--8<-- "scripts/labs/ppo_from_scratch.py:value_loss"

!!! lab "Checkpoint" At initialization, the critic predicts near-zero for all states, so explained variance should be near zero (predictions are no better than predicting the mean):

```python
values = torch.randn(64) * 0.01   # near-zero predictions at init
returns = torch.randn(64)          # target values with actual variance
loss, info = compute_value_loss(values, returns)
print(f"value_loss:        {info['value_loss']:.4f}")      # ~0.5 (large error)
print(f"explained_variance: {info['explained_variance']:.3f}") # ~0.0 (no prediction skill)
```

1.5.5 2.5.5 Wiring It Together: The PPO Update

The individual components above are combined into a single update step. PPO's total loss weaves together three terms: a clipped policy objective (which we maximize), a value prediction loss (which we minimize), and an entropy bonus (which we maximize to encourage exploration).

One common convention is to *minimize* the following total loss:

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

where $c_1 = 0.5$ (value coefficient) and c_2 is the entropy coefficient (typically 0.0 for Fetch tasks, where exploration isn't the bottleneck).

--8<-- "scripts/labs/ppo_from_scratch.py:ppo_update"

!!! lab "Checkpoint" Run 10 updates on a mock batch and verify the value loss decreases (the critic is learning to predict returns):

```python
model = ActorCritic(obs_dim=4, act_dim=2)
optimizer = torch.optim.Adam(model.parameters(), lr=3e-4)
# ... create batch with observations, actions, advantages, returns ...

initial_vloss = None
for i in range(10):
    info = ppo_update(model, optimizer, batch)
    if initial_vloss is None:
        initial_vloss = info["value_loss"]

print(f"Value loss: {initial_vloss:.4f} -> {info['value_loss']:.4f}") # decreasing
print(f"Approx KL:  {info['approx_kl']:.4f}")  # small, < 0.05
```

1.5.6 2.5.6 Verify the Full Lab

Run the from-scratch implementation's sanity checks -- this exercises all the components above end-to-end:

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

Expected output:

```
============================================================
PPO From Scratch -- Verification
============================================================
Verifying GAE computation...
  Advantages: [...]                    # finite values, last > 0
  Returns: [...]                       # finite values
  [PASS] GAE computation OK

Verifying PPO loss...
  Loss: -0.XXXX                        # finite
  Approx KL: 0.0000                    # near zero (same policy)
  Clip fraction: 0.0000                # no clipping (same policy)
  [PASS] PPO loss OK

Verifying PPO update...
  Initial value loss: X.XXXX
  Final value loss: X.XXXX             # lower than initial
  Approx KL: 0.XXXX                   # small, bounded
  [PASS] PPO update OK

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

This lab is **not** how we train policies -- that's what SB3 is for. The lab shows *what* SB3 is doing internally, with every tensor operation visible.

1.5.7 2.5.7 Verify vs SB3 (Optional)

SB3 is the scaling engine we use in the Run It track. This optional check confirms that our from-scratch GAE implementation matches SB3's reference implementation (`RolloutBuffer.compute_returns_and_advantage`):

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

Expected output:

```
============================================================
PPO From Scratch -- SB3 Comparison
============================================================
Max abs advantage diff: 0.000e+00
Max abs returns diff:   0.000e+00
Tolerance (atol):       1.0e-06

[PASS] Our GAE matches SB3 RolloutBuffer
```

1.5.8 2.5.8 Exercises: Modify and Observe

These exercises help you develop intuition by changing the code and seeing what happens.

Exercise 2.5.1: GAE Lambda Ablation

Modify `gae_lambda` in the verification test and observe the effect:

```
# In scripts/labs/ppo_from_scratch.py, find verify_gae() and try:
# gae_lambda = 0.0  (one-step TD, high bias)
# gae_lambda = 1.0  (Monte Carlo, high variance)
# gae_lambda = 0.95 (default balance)
```

Question: How do the advantage values change? Why does $\lambda = 0$ produce smaller magnitude advantages?

Exercise 2.5.2: Clipping Effect

Modify `clip_range` in `verify_ppo_loss()`:

```
# Try clip_range = 0.0, 0.1, 0.2, 0.5, 1.0
```

Question: What happens to `clip_fraction` as you increase `clip_range`? At what point does clipping become ineffective?

Exercise 2.5.3: Train on CartPole

Run the full demo to see PPO learn a simple task from scratch:

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

Actual results (CPU, ~30 seconds):

Iteration	Steps	Avg Return	Value Loss	What's Happening
1	2k	22	7.6	Random behavior
5	10k	45	25.9	Starting to balance
10	20k	64	57.6	Improving steadily
15	31k	129	32.0	Getting close

Iteration	Steps	Avg Return	Value Loss	What's Happening
18	37k	**254**	14.2	**[SOLVED]**

Key observations: The 195+ threshold (solved) is crossed around iteration 18 at roughly 37k steps. The value loss starts high as the critic learns and decreases as predictions improve, while the approximate KL divergence stays bounded (typically < 0.05), confirming that the clipping mechanism is working as intended. This is the same algorithm SB3 uses--the from-scratch version just makes every step explicit.

Exercise 2.5.4: Record a GIF of the Trained Policy

After training, you can record a GIF showing the learned behavior:

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

This trains the policy and saves a GIF to `videos/ppo_cartpole_demo.gif`.

Note: `docker/dev.sh` uses `docker run -it` and will fail without a TTY. If you need non-interactive execution (CI), run `docker run --rm` with the same env vars and mounts as `docker/dev.sh` (use that script as the authoritative reference), but drop `-it`, then execute `python scripts/labs/ppo_from_scratch.py --demo --record` inside the container.

Trained policy balancing the pole:

PPO CartPole Demo

1.6 Part 3: WHAT -- Running the Experiment (Run It)

1.6.1 3.1 The One-Command Version

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all --seed 0
```

This runs training, evaluation, and generates a report. Takes ~6-10 minutes on a GPU.

Artifacts (for `--seed 0`):

Artifact	Path
Checkpoint	`checkpoints/ppo_FetchReachDense-v4_seed0.zip`
Metadata	`checkpoints/ppo_FetchReachDense-v4_seed0.meta.json`
Eval JSON	`results/ch02_ppo_fetchreachdense-v4_seed0_eval.json`
TensorBoard logs	`runs/ppo/FetchReachDense-v4/seed0/`

For a quick sanity check (~1 minute):

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py train --total-steps 50000
```

1.6.2 3.2 What to Expect

Training progress (watch for these milestones):

Timesteps	Success Rate	What's Happening
0-50k	5-10%	Random exploration
50k-100k	30-50%	Policy starting to learn

Timesteps	Success Rate	What's Happening
100k-200k	70-90%	Rapid improvement
200k-500k	95-100%	Fine-tuning, convergence

Our test run achieved **100% success rate** after 500k steps, with a **4.6mm average goal distance** (the environment considers <50mm as success) and **~1300 steps/second** throughput on an NVIDIA GB10.

1.6.3 3.3 Reading the TensorBoard Logs

Launch TensorBoard:

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

Healthy training looks like:

Metric	Expected Behavior
`rollout/ep_rew_mean`	Steadily increasing (less negative)
`rollout/success_rate`	0 -> 1 over training
`train/value_loss`	High initially, decreases, stabilizes
`train/approx_kl`	Small (< 0.03), occasional spikes OK
`train/clip_fraction`	0.1-0.3 (some updates clipped, not all)
`train/entropy_loss`	Slowly moves toward 0 (SB3 logs -entropy; entropy magnitude decreases)

Warning signs:

Symptom	Likely Problem	Fix
`ep_rew_mean` flatlines at start	Environment misconfigured	Check obs/action shapes
`value_loss` explodes	Reward scale wrong	Check reward range
`approx_kl` consistently > 0.05	Learning rate too high	Reduce to 1e-4
`clip_fraction` near 1.0	Updates too aggressive	Reduce LR or clip_range
`entropy_loss` immediately 0	Policy collapsed	Increase ent_coef

1.6.4 3.4 Verifying Your Results

After training completes, check the evaluation report:

```
cat results/ch02_ppo_fetchreachdense-v4_seed0_eval.json | python -m json.tool | head -20
```

Key fields:

```
{
  "aggregate": {
    "success_rate": 1.0,
    "return_mean": -0.40,
    "final_distance_mean": 0.0046
  }
}
```

Passing criteria: success rate above 90%, mean return above -10, and final distance below 0.02m.

1.7 Part 4: Understanding What You Built

1.7.1 4.1 What the Policy Actually Learned

The trained policy maps observations to actions:

Input (goal-conditioned dict observation): `observation` with shape (10,), `achieved_goal` with shape (3,), and `desired_goal` with shape (3,). Stable Baselines 3 uses `MultiInputPolicy` to flatten and concatenate these inputs before feeding them to an MLP.

Output (4 dimensions): three Cartesian velocity commands (dx, dy, dz) plus a gripper open/close command.

The network learned that to reach a goal, it should output velocities that point toward the goal. This seems obvious, but the network discovered it purely from trial and error.

1.7.2 4.2 The Clipping in Action

During training, you can observe the clipping mechanism working. A `clip_fraction` of 0.15 means 15% of updates were clipped, which is healthy--it shows that some updates are being constrained, preventing instability. By contrast, `clip_fraction = 0` would mean the policy isn't learning aggressively enough to trigger any clipping, while `clip_fraction = 1.0` would mean all updates are being constrained, indicating that the policy is changing too aggressively between updates.

1.7.3 4.3 Why This Validates Your Pipeline

If PPO succeeds on dense Reach, you know that your environment is configured correctly (observations and actions have the right shapes and semantics), that the network architecture works (MultiInputPolicy correctly processes dict observations), that GPU acceleration works (training completes in reasonable time), that your evaluation protocol is sound (you can load checkpoints and run deterministic rollouts), and that metrics are computed correctly (success rate matches what you observe). With all of these validated, you can add complexity--SAC, HER, harder tasks--with confidence that failures are algorithmic, not infrastructural.

1.8 Part 5: Exercises

1.8.1 Exercise 2.1: Reproduce the Baseline

Run training with seed 0 and verify you achieve > 90% success rate. Record the final success rate, final mean return, training time, and steps per second.

1.8.2 Exercise 2.2: Multi-Seed Validation

Run with seeds 0-4 and compute mean and standard deviation:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py multi-seed --seeds 5
```

A robust result should have std < 5%.

1.8.3 Exercise 2.3: Explain the Clipping (Written)

Answer these questions in your own words:

1. What problem does vanilla policy gradient have that PPO fixes?

2. Why does PPO clip in likelihood ratio space rather than parameter space?
3. If $\epsilon = 0$ (no change allowed), what would happen?
4. If $\epsilon = 1$ (large changes allowed), what would happen?

1.8.4 Exercise 2.4: Ablation Study

Train with `clip_range` values of 0.1, 0.2, and 0.4. For each setting, note whether final performance changes, whether training stability changes, and how `clip_fraction` in TensorBoard differs across the three runs.

1.9 Part 6: Common Failures and Solutions

1.9.1 "Success rate stays at 0%"

Check 1: Environment shape

```python
import gymnasium as gym
env = gym.make("FetchReachDense-v4")
obs, _ = env.reset()
print("Obs shape:", {k: v.shape for k, v in obs.items()})
print("Action shape:", env.action_space.shape)
```

Expected: obs has keys with shapes (10,), (3,), (3,); action shape (4,).

Check 2: Reward range

```python
for _ in range(100):
    action = env.action_space.sample()
    obs, reward, _, _, _ = env.step(action)
    print(f"Reward: {reward:.3f}")
```

Expected: rewards in roughly [-1, 0] range.

1.9.2 "Value loss explodes"

Usually means reward scale is wrong. FetchReachDense returns rewards in [-1, 0]. If you're seeing rewards in [-1000, 0] or similar, something is misconfigured.

1.9.3 "Training is slow (<500 fps)"

Check that GPU is being used:

```
nvidia-smi  # Should show python process using GPU memory
```

Check that you're running in Docker with `--gpus all`:

```
bash docker/dev.sh nvidia-smi
```

Note: Even with a GPU, Fetch training is often CPU-bound on MuJoCo simulation. Low GPU utilization is normal; use steps/sec and learning curves as your primary signals.

1.10 Conclusion

This chapter established something more important than a trained policy: **confidence in your infrastructure**.

The ”truth serum” principle says: before tackling hard problems, verify that easy problems work. PPO on dense Reach is that easy problem. With 100% success rate achieved, we know our environment, training loop, evaluation protocol, and GPU setup are correct.

The key takeaways are that PPO prevents catastrophic updates through clipped likelihood ratios, that dense rewards provide continuous signal by decoupling exploration from learning, that diagnostics reveal problems early (which is why watching TensorBoard matters more than waiting for final metrics), and that infrastructure bugs are silent killers--validating before adding complexity saves far more time than it costs.

What's Next:

Chapter 3 introduces SAC (Soft Actor-Critic) on the same dense Reach task. SAC is off-policy, meaning it reuses old data through a replay buffer. This is more sample-efficient but adds complexity (target networks, entropy tuning). By running SAC on dense Reach, we validate the off-policy machinery before adding HER for sparse rewards in Chapter 4.

1.11 References

1. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.
2. Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-Dimensional Continuous Control Using Generalized Advantage Estimation. arXiv:1506.02438.
3. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep Reinforcement Learning that Matters. AAAI.
4. Stable Baselines3 Documentation: https://stable-baselines3.readthedocs.io/
5. Spinning Up in Deep RL: https://spinningup.openai.com/