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. 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**. If this doesn't work, something is broken in your setup, not your algorithm.

By the end, you will have:

1. A trained policy achieving >90% success rate (we got 100% in our test run)
2. An understanding of *why* PPO works (not just *that* it works)
3. Diagnostic skills to identify common training failures
4. Confidence to move to harder problems

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:

- Environment misconfiguration
- Wrong network architecture
- Bad hyperparameters
- Bug in your algorithm implementation
- The task simply needing more training time
- 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.

The Setup: At each timestep, our policy π (a neural network) sees the current state s and goal g, and outputs an action a. The environment responds with a new state and a reward.

The Objective: Find policy parameters θ that maximize total reward:

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

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(s, a) = Q(s, a) - V(s)
```

where:

- $Q(s, a)$ = expected return if you take action a in state s, then follow your policy

- $V(s)$ = expected return if you just follow your policy from state s

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

The Policy Gradient:

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

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**. Here's why:

Problem 1: Advantage estimates are noisy.

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

Problem 2: Big updates break everything.

Suppose we estimate that some action is great ($A >> 0$) and crank up its probability. But if our estimate was wrong, we've now committed to a bad action. Worse, the new policy visits different states, making our old advantage estimates invalid. 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 probability ratio:

```
r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)}
```

This measures how the action probability changed:

- $r = 1$: same probability as before
- $r = 2$: action is now twice as likely
- $r = 0.5$: action is now half as likely

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( r_t A_t, \text{clip}(r_t, 1-\epsi
```

What this does:

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

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: The old policy assigns probability 0.3 to action a in state s. 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 $\pi_\theta(a|s) = 0.6$.

Problem: The ratio $r = 0.6/0.3 = 2.0$. 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: $r \cdot A = 2.0 \times 2 = 4.0$
- 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.

Why this helps:

- Every action provides signal: "you got 2cm closer" or "you drifted 1cm away"
- The learning algorithm always has gradient information
- Exploration isn't a bottleneck--even random actions provide useful data

Compare to sparse rewards ($R = 0$ if success, -1 otherwise):

- You only learn when you succeed
- Random policy almost never succeeds
- Most of your data is uninformative

Dense rewards 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 $\pi_\theta(a|s, g)$: Given the state and goal, output a probability distribution over actions. For continuous actions, this is typically a Gaussian with learned mean and standard deviation.

Critic $V_\phi(s, g)$: Given the state and goal, 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? Three reasons:

1. **Different objectives.** The actor maximizes expected return (wants to find good actions). The critic minimizes prediction error (wants accurate value estimates). These gradients can conflict--improving one may hurt the other.
2. **Different output types.** The actor outputs a probability distribution (mean and variance for continuous actions). The critic outputs a single scalar. Forcing these through the same final layers creates unnecessary coupling.
3. **Stability.** The critic's value estimates are used to compute advantages, which then train the actor. If actor updates destabilize the critic, the advantages become noisy, which destabilizes the actor further--a vicious cycle.

A geometric perspective: There may be deeper structure here. Consider what we're learning: a mapping from states 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_{k=0}^{\infty} (\gamma \lambda)^k \delta_{t+k}
```

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the TD residual.

The parameter λ interpolates between:

- $\lambda = 0$: One-step TD (high bias, low variance)
- $\lambda = 1$: Monte Carlo (low bias, high variance)
- $\lambda = 0.95$: Typical default

Steps 3-4: Update Networks

Unlike supervised learning, we do multiple passes over the same data:

- `n_epochs = 10` is typical for PPO
- Each pass uses 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 3: WHAT -- Running the Experiment

1.5.1 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.

For a quick sanity check (~1 minute):

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

1.5.2 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
100k-200k	70-90%	Rapid improvement
200k-500k	95-100%	Fine-tuning, convergence

Our test run achieved:

- **100% success rate** after 500k steps
- **4.6mm average goal distance** (environment considers <50mm as success)
- **~1300 steps/second** throughput on NVIDIA GB10

1.5.3 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 decreasing (policy becoming more deterministic)

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

1.5.4 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 > 90%
- Mean return > -10
- Final distance < 0.02m

1.6 Part 4: Understanding What You Built

1.6.1 4.1 What the Policy Actually Learned

The trained policy maps observations to actions:

Input (25 dimensions total):

- End-effector position (3)
- End-effector velocity (3)
- Gripper state (4)
- Desired goal (3)
- Achieved goal (3)
- Various other features (9)

Output (4 dimensions):

- dx, dy, dz: Cartesian velocity commands
- 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.

1.6.2 4.2 The Clipping in Action

During training, you can observe the clipping mechanism working:

- `clip_fraction = 0.15` means 15% of updates were clipped
- This is healthy--some updates are constrained, preventing instability
- `clip_fraction = 0` would mean the policy isn't learning aggressively enough
- `clip_fraction = 1.0` would mean all updates are being constrained (too aggressive)

1.6.3 4.3 Why This Validates Your Pipeline

If PPO succeeds on dense Reach, you know:

1. **Environment is configured correctly** - Observations and actions have the right shapes and semantics
2. **Network architecture works** - MultiInputPolicy correctly processes dict observations
3. **GPU acceleration works** - Training completes in reasonable time
4. **Evaluation protocol is sound** - You can load checkpoints and run deterministic rollouts
5. **Metrics are computed correctly** - Success rate matches what you observe

Now you can add complexity (SAC, HER, harder tasks) with confidence that failures are algorithmic, not infrastructural.

1.7 Part 5: Exercises

1.7.1 Exercise 2.1: Reproduce the Baseline

Run training with seed 0 and verify you achieve > 90% success rate. Record:

- Final success rate
- Final mean return
- Training time
- Steps per second

1.7.2 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%.

1.7.3 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 probability 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?

1.7.4 Exercise 2.4: Ablation Study

Train with `clip_range` values of 0.1, 0.2, and 0.4. For each:

- Does final performance change?
- Does training stability change?
- How does `clip_fraction` in TensorBoard differ?

1.8 Part 6: Common Failures and Solutions

1.8.1 "Success rate stays at 0%"

Check 1: Environment shape

```
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

```
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.

1.8.2 ”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.

1.8.3 ”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
```

1.9 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.

Key takeaways:

1. **PPO prevents catastrophic updates** through clipped probability ratios
2. **Dense rewards provide continuous signal**, decoupling exploration from learning
3. **Diagnostics reveal problems early**--watch TensorBoard, not just final metrics
4. **Infrastructure bugs are silent killers**--validate before adding complexity

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.

1.10 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/