3 PPO on Dense Reach: Your First Trained Policy

Vlad Prytula

2026-02-17

Contents

3 PPO on Dense Reach: Your First Trained Policy

This chapter covers:

- Deriving the PPO clipped surrogate objective from the policy gradient theorem -- why constraining the likelihood ratio prevents the catastrophic updates that plague vanilla policy gradient
- Implementing PPO from scratch: actor-critic network, Generalized Advantage Estimation (GAE), clipped policy loss, value loss, and the full update loop
- Verifying each component with concrete checks (tensor shapes, expected values, learning curves) before assembling the complete algorithm
- Bridging from-scratch code to Stable Baselines 3 (SB3): confirming that both implementations compute the same GAE advantages, and mapping SB3 TensorBoard metrics to the code you wrote
- Training PPO on FetchReachDense-v4 to 100% success rate, establishing the pipeline baseline that every future chapter builds on

In Chapter 2, you dissected the Fetch environment -- observation dictionaries, action semantics, dense and sparse reward computation, goal relabeling, and the random-policy baseline (0% success, mean return around -20). You understand what the agent sees and what the numbers mean.

But understanding the environment is necessary and not sufficient. A random policy achieves 0% success. You need an algorithm that converts observations into intelligent actions -- one that improves through experience. The question is: which algorithm, and how do you verify it is working?

This chapter introduces PPO (Proximal Policy Optimization), an on-policy algorithm that learns by clipping likelihood ratios to prevent destructive updates. You will derive the PPO objective, implement it from scratch (actor-critic network, GAE, clipped loss, value loss), verify each component, bridge to SB3, and train a policy that reaches 100% success on FetchReachDense-v4. This validates your entire training pipeline.

One note before we begin: PPO works here because dense rewards provide continuous gradient signal. But PPO is on-policy -- it discards all data after each update, wasting expensive simulation time. Chapter 4 introduces SAC, an off-policy algorithm that

stores and reuses experience in a replay buffer. That off-policy machinery is what Chapter 5 (HER) requires when we tackle sparse rewards.

3.1 WHY: The learning problem

What are we optimizing?

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

At each timestep t, our **policy** π -- a neural network with parameters θ -- sees the current state s_t and the goal g. We 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.

Reward. The reward $r_t \in \mathbb{R}$ is the immediate feedback signal at timestep t. In FetchReachDense-v4, $r_t = -\|p_t - g\|_2$ where p_t is the gripper position and g is the goal. More negative means farther from the goal; zero means perfect.

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 things: sooner is better than later, and distant rewards are more uncertain.

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

Now the objective. We want to 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 $J(\theta)$ measures how good a policy with parameters θ is, averaged over many episodes. The sum $G_t = \sum_{t=0}^{T-1} \gamma^t r_t$ 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.

The policy gradient theorem

Here is 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 is not 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:

- $Q(x, a)$ is the **Q-function**: expected return if you take action a in goal-conditioned state x, then follow your policy
- $V(x)$ is the **value function**: expected return if you 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.

A concrete example helps here. If $V(x) = -0.3$ and $Q(x, a_{\text{left}}) = -0.1$, then $A(x, a_{\text{left}}) = +0.2$ -- moving left is better than the policy's average from this state. But notice: a positive advantage does NOT mean the action leads to a good outcome in absolute terms. The expected return is still $Q = -0.1$, which is negative. The advantage tells you the action is better *relative to what you normally do*, not that it is good in any absolute sense.

The **policy gradient theorem** (Sutton & Barto, 2018, Ch13.1) tells us how to differentiate the objective:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) \cdot A(x_t, a_t)\right]$$

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

The instability problem

In theory, you can follow this gradient and improve. In practice, vanilla policy gradient is notoriously unstable (Henderson et al., 2018). Two things go wrong.

Advantage estimates are noisy. We do not know the true advantage -- we estimate it from sampled trajectories. With a finite batch, these estimates have high variance. Sometimes they are way off, and we make bad updates.

Big updates break everything. Suppose we estimate that some action is great ($A \gg 0$) and crank up its probability. But if our estimate was wrong, we have 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 is not hypothetical. Training curves that look promising can crash to zero and never recover. In our experience, unclipped policy gradient on Fetch tasks fails roughly half the time -- you get lucky with some seeds and unlucky with others. That is not a reliable foundation to build on.

PPO's solution: constrained updates

PPO's key idea: do not change the policy too much in one update (Schulman et al., 2017).

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** (also called 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:

- $\rho = 1$: same likelihood as before
- $\rho = 2$: action is now twice as likely
- $\rho = 0.5$: action 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: $\rho_t = \exp\big(\log \pi_\theta(a_t \mid x_t) - \log \pi_{\text{old}}(a_t \mid x_t)\big)$.

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

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

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.

A concrete example

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

Imagine a discrete action space. The old policy assigns probability 0.3 to action a in 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 $\pi_\theta(a \mid x) = 0.6$. The ratio $\rho = 0.6/0.3 = 2.0$. We doubled the probability in one update. If our advantage estimate was wrong, we have 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 flows through the clipped version. We still increase the action probability, but the update is bounded. We cannot go from 0.3 to 0.6 in one step -- we would need multiple updates, each constrained.

Why dense rewards matter here

FetchReachDense-v4 gives reward $r_t = -\|g_a - g_d\|_2$ at every step -- the negative distance to the goal.

Why this helps PPO:

- **Every action provides signal.** "You got 2cm closer" or "you drifted 1cm away" -- the algorithm always has gradient information.
- **Exploration is not a bottleneck.** Even random actions produce useful data, because every distance tells you something.
- **Learning problems are algorithm problems.** If PPO fails on dense Reach, the issue is in your implementation, not in insufficient exploration. Dense rewards decouple the exploration problem from the learning problem.

Compare to sparse rewards ($R = 0$ if success, -1 otherwise): most of your data carries no information about which direction to improve. We address that challenge in Chapter 5 with HER.

The well-posedness check

Before we train, it is worth asking three practical questions (from Chapter 1):

1. **Can this be solved?** Yes -- the policy network has 16 inputs and 4 outputs, with ~5,600 parameters. The mapping from "see goal, move toward it" is well within the capacity of a small MLP. A hand-coded controller solves this task with a few lines of code, so a learned one certainly can.
2. **Is the solution reliable?** We expect yes -- the task is low-dimensional, the reward is smooth, and the goal distribution is bounded. Different seeds should converge to qualitatively similar policies (move toward the goal), even if the exact parameters differ.

3. **Is the solution stable?** With dense rewards and PPO's clipping, small hyperparameter changes should not break convergence. We verify this empirically: three seeds with the same hyperparameters all reach 100% success (see Reproduce It).

These questions are more interesting for harder tasks. For dense Reach, the answers are reassuring -- which is precisely the point. We start here because failure would be unambiguous evidence of a bug.

3.2 HOW: The actor-critic architecture and training loop

The actor-critic architecture

PPO maintains two neural networks (or two heads of one network):

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 a Gaussian with learned mean and standard deviation.

Critic $V_\phi(x)$: Given the same input, estimate the expected return. This is what we use to compute advantages.

Why two networks, not one? 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 feed into advantages, which train the actor. If actor updates destabilize the critic, advantages become noisy, which destabilizes the actor further -- a vicious cycle.

In practice, implementations often share early layers (a ”backbone”) with separate final layers (”heads”). This captures shared features while keeping the objectives separate. SB3 uses this approach by default.

The training loop

Here is the PPO training loop in pseudocode:

```
repeat:
    1. Collect N steps using current policy
    2. Compute advantages using critic (GAE)
    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 via a parameter λ. The full equation appears in Section 3.4 where we implement it.

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. This is where PPO's fundamental limitation appears.

> **Definition (on-policy learning).** An algorithm is **on-policy** if it can only learn from data collected by the current policy π_θ. Every time you update θ, all transitions collected with the old parameters become invalid for computing unbiased gradients. You must throw away the data and collect fresh transitions with the new policy.

Here is what this means concretely. Each update cycle in PPO produces `n_steps * n_envs` transitions (8,192 with our settings). You use these transitions for `n_epochs` gradient steps, then discard all of them -- even though many transitions contain useful information that could improve the policy further. This is sample-inefficient: millions of simulation steps are thrown away after a single use.

Why does this matter for what comes next? The on-policy constraint is the main reason we move to SAC in Chapter 4. Off-policy methods store transitions in a replay buffer and reuse them across many updates, so every simulation step contributes to learning not once but repeatedly. More importantly, the replay buffer is what makes Hindsight Experience Replay (Chapter 5) possible -- you cannot relabel goals in data you have already discarded.

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
`gamma`	0.99	Discount factor
`ent_coef`	0.0	Entropy bonus (exploration incentive)

For FetchReachDense-v4, SB3 defaults work well. We recommend against tuning hyperparameters until you have verified that the baseline works with these values.

Compact equation summary

For reference, here are all the PPO equations in one place. Each appears again in the Build It sections alongside its implementation.

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

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

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l} \qquad \hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$$

$$\rho_t(\theta) = \pi_\theta(a_t \mid x_t) \;/\; \pi_{\theta_{\text{old}}}(a_t \mid x_t)$$

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

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

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

3.3 Build It: The actor-critic network

Before we can compute losses, we need the network they operate on. We build PPO piece by piece, verifying each component before moving to the next. The implementations live in `scripts/labs/ppo_from_scratch.py` -- these are for understanding, not production.

The actor-critic network has a shared backbone (two hidden layers with tanh activations) and separate heads for the actor and critic:

```python
# Actor-critic network with shared backbone
# (from scripts/labs/ppo_from_scratch.py:actor_critic_network)

class ActorCritic(nn.Module):
    def __init__(self, obs_dim: int, act_dim: int,
                 hidden_dim: int = 64):
        super().__init__()
        self.backbone = nn.Sequential(
            nn.Linear(obs_dim, hidden_dim), nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim), nn.Tanh(),
```

```
        )
        # Actor: mean of Gaussian policy
        self.actor_mean = nn.Linear(hidden_dim, act_dim)
        # Learnable log std (state-independent)
        self.actor_log_std = nn.Parameter(torch.zeros(act_dim))
        # Critic: scalar value estimate
        self.critic = nn.Linear(hidden_dim, 1)

    def forward(self, obs):
        features = self.backbone(obs)
        mean = self.actor_mean(features)
        std = self.actor_log_std.exp()
        dist = Normal(mean, std)
        value = self.critic(features).squeeze(-1)
        return dist, value
```

The actor outputs a Gaussian distribution parameterized by a learned mean and a state-independent log standard deviation. The critic outputs a single scalar -- the estimated value $V(x)$. Both share the backbone features but have independent output layers.

The network is small by design -- Fetch tasks use MLPs, not CNNs. For FetchReach, the input dimension is 16 (10D observation + 3D achieved goal + 3D desired goal, concatenated by SB3's `MultiInputPolicy`), and the output is 4D (dx, dy, dz, gripper).

> **Checkpoint.** Instantiate the network with `obs_dim=16, act_dim=4` and run a forward pass with a random input. You should see: action mean shape `(1, 4)`, value shape `(1,)`, total parameters around 5,577. All outputs should be finite. If you get a shape error, check that `obs_dim` matches your concatenated observation size.

3.4 Build It: GAE computation

The advantage formula from Section 3.2 tells us how much better an action was compared to the policy's average behavior. Generalized Advantage Estimation (Schulman et al., 2015) computes this efficiently using a parameter λ that trades off 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} -- there are no future rewards to estimate. The value

terms V_{rollout} are computed when collecting the rollout and treated as constants during optimization.

The λ parameter controls the bias-variance tradeoff:

- $\lambda = 0$: One-step TD. High bias (only looks one step ahead), low variance.
- $\lambda = 1$: Monte Carlo. Low bias (uses full trajectory), high variance.
- $\lambda = 0.95$: The typical default, and what we use.

In code, we compute GAE backwards through the trajectory:

```python
# GAE computation (backward pass through trajectory)
# (from scripts/labs/ppo_from_scratch.py:gae_computation)

def compute_gae(rewards, values, next_value, dones,
                gamma=0.99, gae_lambda=0.95):
    T = len(rewards)
    advantages = torch.zeros(T, device=rewards.device)
    last_gae = 0.0

    for t in reversed(range(T)):
        if t == T - 1:
            next_val = next_value
        else:
            next_val = values[t + 1]
        next_val = next_val * (1.0 - dones[t])

        # TD residual: was this transition better than expected?
        delta = rewards[t] + gamma * next_val - values[t]

        # GAE recursion with episode boundary masking
        last_gae = (delta + gamma * gae_lambda
                    * (1.0 - dones[t]) * last_gae)
        advantages[t] = last_gae

    returns = advantages + values
    return advantages, returns
```

The key line is the GAE recursion: `last_gae = delta + gamma * gae_lambda * (1 - done) * last_gae`. This accumulates TD residuals backwards, decaying each by $\gamma\lambda$. The `(1 - done)` term resets the accumulation at episode boundaries -- when an episode ends, future advantages from a different episode should not bleed in.

The **returns** are computed as `advantages + values`. These serve as the target for the value function: $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$.

Math	Code	Meaning
δ_t	`delta`	TD residual: was this transition better than expected?
γ	`gamma`	Discount factor (0.99)

Math	Code	Meaning
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?

Checkpoint. Test with a trajectory where reward arrives only at the end: `rewards[-1] = 1.0`, all other rewards zero, `dones[-1] = 1.0`. The last advantage should be positive because the agent received a reward the value function did not fully predict. All advantages and returns should be finite. If the last advantage is negative or zero, check that the done mask is applied correctly -- the bootstrap value should be zeroed when the episode terminates.

3.5 Build It: The clipped surrogate loss

The PPO objective from Section 3.1:

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

In code, this maps directly to ratio computation, clipping, and a pessimistic (minimum) bound:

```
# PPO clipped surrogate loss (part 1: ratio and clipping)
# (from scripts/labs/ppo_from_scratch.py:ppo_loss)

def compute_ppo_loss(dist, old_log_probs, actions,
                     advantages, clip_range=0.2):
    # Log prob under CURRENT policy
    new_log_probs = dist.log_prob(actions).sum(dim=-1)

    # Probability ratio via logs (numerically stable)
    log_ratio = new_log_probs - old_log_probs
    ratio = log_ratio.exp()

    # Clipped ratio: keep within [1-eps, 1+eps]
    clipped_ratio = torch.clamp(
        ratio, 1.0 - clip_range, 1.0 + clip_range)

    # Pessimistic bound: take the minimum
    surr1 = ratio * advantages
    surr2 = clipped_ratio * advantages
    policy_loss = -torch.min(surr1, surr2).mean()
```

The ratio is computed in log-space (`exp(log_new - log_old)`) for numerical stability. The `min` operation is the pessimistic bound -- it takes the more conservative of the clipped and unclipped surrogate, ensuring we never overestimate the benefit of a policy change.

The function also returns diagnostics that track training health:

```
    # PPO loss diagnostics (part 2)
    with torch.no_grad():
        approx_kl = ((ratio - 1) - log_ratio).mean()
        clip_fraction = (
            (ratio - 1.0).abs() > clip_range
        ).float().mean()

    info = {"policy_loss": policy_loss.item(),
            "approx_kl": approx_kl.item(),
            "clip_fraction": clip_fraction.item(),
            "ratio_mean": ratio.mean().item()}
    return policy_loss, info
```

The `clip_fraction` tells you what fraction of updates were clipped -- this is the same metric you will see in TensorBoard under `train/clip_fraction`. The `approx_kl` measures how much the policy diverged from the old policy in this update step.

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

> **Checkpoint.** When the policy has not changed yet (same model, same parameters), all ratios should be 1.0 and no clipping should occur. Create a model, compute `old_log_probs` with `torch.no_grad()`, then immediately compute the loss with the same model. You should see: `clip_fraction = 0.000`, `ratio_mean = 1.000`, `approx_kl` near 0.000. If `clip_fraction` is nonzero, something is wrong -- the policy should not have changed between the two forward passes.

3.6 Build It: 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 $\hat{G}_t = \hat{A}_t + V_{\text{rollout}}(x_t)$:

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

```
# Value function loss (critic update)
# (adapted from scripts/labs/ppo_from_scratch.py:value_loss)

def compute_value_loss(values, returns):
    value_loss = 0.5 * (values - returns).pow(2).mean()
```

```
    # Explained variance: how much of the return variance
    # does the critic explain?
    with torch.no_grad():
        var_target = returns.var()
        if var_target < 1e-8:
            ev = 0.0
        else:
            ev = (1.0 - (returns - values).var()
                  / var_target).item()

    info = {"value_loss": value_loss.item(),
            "explained_variance": ev}
    return value_loss, info
```

The **explained variance** is a useful diagnostic. It measures how much of the return variance the critic captures:

- EV = 1: perfect predictions
- EV = 0: predictions are no better than predicting the mean
- EV < 0: predictions are worse than predicting the mean

At initialization, the critic predicts near-zero for everything, so explained variance starts near 0. As training progresses, it should climb toward 0.5-0.9. If it stays at 0 or goes negative, the critic is not learning -- check that the optimizer is attached to the critic's parameters.

> **Checkpoint.** Create near-zero value predictions and random returns. You should see `value_loss` around 0.5 (the predictions are wrong, so the squared error is roughly the variance of the returns) and `explained_variance` near 0.0 (the critic has no prediction skill yet). If `value_loss` is exactly 0, the critic and return tensors may be the same object -- check that you are using `.detach()` or separate computations.

3.7 Build It: PPO update (wiring)

The individual components above are combined into a single update step. PPO minimizes a combined 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), c_2 is the entropy coefficient (0.0 for Fetch tasks -- exploration is not the bottleneck with dense rewards), and $\mathcal{H}[\pi]$ is the entropy of the policy distribution (higher entropy means more exploration).

```
# PPO update step (part 1: compute losses)
# (from scripts/labs/ppo_from_scratch.py:ppo_update)

def ppo_update(model, optimizer, batch,
```

```
            clip_range=0.2, value_coef=0.5,
            entropy_coef=0.0, max_grad_norm=0.5):
    dist, values = model(batch.observations)

    # Normalize advantages (reduces variance)
    adv = batch.advantages
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)

    # Individual losses
    policy_loss, p_info = compute_ppo_loss(
        dist, batch.old_log_probs, batch.actions,
        adv, clip_range)
    value_loss, v_info = compute_value_loss(
        values, batch.returns)
    entropy = dist.entropy().mean()
    entropy_loss = -entropy
```

This computes all three loss components independently. The advantage normalization (subtracting mean, dividing by standard deviation) is standard practice -- it reduces sensitivity to reward scale.

The combined loss and gradient step:

```
    # PPO update step (part 2: backprop and step)
    total_loss = (policy_loss
                  + value_coef * value_loss
                  + entropy_coef * entropy_loss)

    optimizer.zero_grad()
    total_loss.backward()
    grad_norm = nn.utils.clip_grad_norm_(
        model.parameters(), max_grad_norm)
    optimizer.step()

    return {**p_info, **v_info,
            "entropy": entropy.item(),
            "total_loss": total_loss.item(),
            "grad_norm": grad_norm.item()}
```

A few things to notice:

- **Advantage normalization.** Before computing the policy loss, we subtract the mean and divide by the standard deviation of the advantages. This is standard practice -- it reduces the sensitivity to reward scale and makes the gradient magnitudes more consistent across batches.
- **Gradient clipping.** We clip the gradient norm at `max_grad_norm=0.5`. This prevents a single bad batch from causing an explosively large update. This is separate from the PPO ratio clipping -- gradient clipping limits the step size in parameter space, while PPO clipping limits the step size in probability space.

- **The entropy term.** We negate the entropy because we minimize the total loss but want to *maximize* entropy. With `entropy_coef=0.0`, this term has no effect -- for FetchReachDense, the dense reward signal is enough to drive learning without an explicit exploration bonus.

  Checkpoint. Run 10 updates on a mock batch (random observations, actions, advantages, returns). The value loss should decrease from its initial value -- the critic is learning to predict returns. The `approx_kl` should stay small (below 0.05) -- the clipping is preventing overly large policy changes. If the value loss does not decrease, verify that `optimizer.step()` is being called and that the model parameters are actually changing.

3.8 Build It: The training loop

The training loop wires together data collection, GAE computation, and the PPO update. The `collect_rollout` function runs the policy in the environment, and `transitions_to_batch` assembles the collected data into a training batch:

```python
# Rollout collection and batch assembly
# (from scripts/labs/ppo_from_scratch.py:ppo_training_loop)

def collect_rollout(env, model, n_steps, device):
    transitions, episode_returns = [], []
    obs, _ = env.reset()
    obs_t = torch.FloatTensor(obs).unsqueeze(0).to(device)
    ep_return = 0.0

    for _ in range(n_steps):
        with torch.no_grad():
            dist, value = model(obs_t)
            action = dist.sample()
            log_prob = dist.log_prob(action).sum(-1)
        # ... step env, store transition, handle resets
        # (full code in scripts/labs/ppo_from_scratch.py)

    with torch.no_grad():
        _, next_value = model(obs_t)
    return transitions, next_value.squeeze(), episode_returns
```

The full implementation handles environment resets on episode boundaries, stores all tensors needed for the PPO update, and computes the bootstrap value for the final state. The outer training loop looks like:

```
for each iteration:
    transitions, next_value, ep_returns = collect_rollout(...)
    batch = transitions_to_batch(transitions, next_value)
    for epoch in range(n_epochs):
        shuffle indices
        for each minibatch:
```

```
            info = ppo_update(model, optimizer, minibatch)
    log metrics
```

You can see this in action by running the demo mode, which trains PPO from scratch on CartPole-v1 (~30 seconds on CPU):

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

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

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
18	37k	254	14.2	Solved (threshold: 195)

CartPole is a much easier task than FetchReach, but it demonstrates that the algorithm works end-to-end: the policy improves, the value loss eventually decreases, and the KL divergence stays bounded (typically below 0.05). This is the same algorithm SB3 uses -- the from-scratch version just makes every step explicit.

> **Checkpoint.** Run `bash docker/dev.sh python scripts/labs/ppo_from_scratch.py --verify` to exercise all components end-to-end. Expected output:
>
> ```
> ============================================================
> PPO From Scratch -- Verification
> ============================================================
> Verifying actor-critic network...
>   [PASS] Actor-critic network OK
> Verifying GAE computation...
>   [PASS] GAE computation OK
> Verifying PPO loss...
>   [PASS] PPO loss OK
> Verifying value loss...
>   [PASS] Value loss OK
> Verifying PPO update...
>   [PASS] PPO update OK
> ============================================================
> [ALL PASS] PPO implementation verified
> ============================================================
> ```
>
> This runs on CPU in under 2 minutes. If any check fails, the error message tells you which component and what went wrong.

3.9 Bridge: From-scratch to SB3

We have built PPO from scratch and verified it component by component. SB3 implements the same math in a production-grade library. Before we use SB3 for the real

training run, let's confirm that the two implementations agree on the same computation.

The bridging proof feeds the same random data (rewards, values, dones) through our `compute_gae` and through SB3's `RolloutBuffer.compute_returns_and_advantage`. Both use gamma=0.99, gae_lambda=0.95, and the same seed:

```
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
Max abs returns diff:   ~0
[PASS] Our GAE matches SB3 RolloutBuffer
```

The two implementations produce identical advantages and returns within floating-point precision (tolerance 1e-6). This means the same math drives both codebases.

What SB3 adds beyond our from-scratch code

Our implementation handles one environment at a time. SB3 adds engineering features that matter for real training:

- **Vectorized environments.** n_envs=8 parallel environments collect data simultaneously, increasing throughput by roughly 8x without algorithmic changes.
- **Learning rate scheduling.** SB3 can anneal the learning rate over training. We used a fixed 3e-4; SB3 defaults to the same but supports schedules.
- **Multi-epoch minibatch shuffling.** SB3 shuffles indices each epoch and handles batching efficiently. Our implementation does the same thing in a more explicit loop.
- **MultiInputPolicy.** SB3 handles dictionary observations automatically -- it builds separate encoders for each key (observation, achieved_goal, desired_goal) and concatenates them. Our from-scratch code assumed a flat observation vector.

Mapping SB3 TensorBoard metrics to our code

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

SB3 TensorBoard key	Our function	What it measur
`train/value_loss`	`compute_value_loss`	Critic prediction
`train/clip_fraction`	`compute_ppo_loss` -> `info["clip_fraction"]`	Fraction of upd
`train/approx_kl`	`compute_ppo_loss` -> `info["approx_kl"]`	Policy divergen
`train/entropy_loss`	`dist.entropy().mean()`	Policy randomn
`train/policy_gradient_loss`	`compute_ppo_loss` -> `info["policy_loss"]`	Clipped surroga
`rollout/ep_rew_mean`	(environment)	Mean episode r

This mapping is important. When you see `train/clip_fraction = 0.15` in TensorBoard, you know what that means -- 15% of the probability ratios exceeded the $[0.8, 1.2]$ clip range, and those updates were constrained. That number came from the same calculation as the `clip_fraction` in our `compute_ppo_loss`. You are not reading opaque metrics from a black box; you are reading quantities you have implemented and verified.

3.10 Run It: Training PPO on FetchReachDense-v4

> **A note on script naming.** The production script is called `ch02_ppo_dense_reach.py` because the repository's tutorial numbering differs from the Manning chapter numbering. In the tutorials, PPO on dense Reach is chapter 2; in this book, it is chapter 3. The script name and artifact paths (checkpoints, eval JSONs, TensorBoard directories) all use `ch02`. This is intentional -- renaming would break the tutorial infrastructure. When you see `ch02` in a command or file path, you are running the right script for this chapter.

```
------------------------------------------------------------
EXPERIMENT CARD: PPO on FetchReachDense-v4
------------------------------------------------------------
Algorithm:    PPO (clipped surrogate, on-policy)
Environment:  FetchReachDense-v4
Fast path:    500,000 steps, seed 0
Time:         ~5 min (GPU) / ~30 min (CPU)

Run command (fast path):
  bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all \
    --seed 0 --total-steps 500000

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

Expected artifacts:
  checkpoints/ppo_FetchReachDense-v4_seed0.zip
  checkpoints/ppo_FetchReachDense-v4_seed0.meta.json
  results/ch02_ppo_fetchreachdense-v4_seed0_eval.json
  runs/ppo/FetchReachDense-v4/seed0/    (TensorBoard logs)

Success criteria (fast path):
  success_rate >= 0.90
  mean_return > -10.0
  final_distance_mean < 0.02

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

Running the experiment

The one-command version:

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

This runs training, evaluation, and generates artifacts. It takes about 5-10 minutes on a GPU. For a quick sanity check that finishes in about 1 minute:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py train --total-steps 5000
```

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

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py eval \
  --ckpt checkpoints/ppo_FetchReachDense-v4_seed0.zip
```

Training milestones

Watch for these milestones during training:

Timesteps	Success Rate	What's happening
0-50k	5-10%	Random exploration, policy is not yet useful
50k-100k	30-50%	Policy starting to move toward goals
100k-200k	70-90%	Rapid improvement phase
200k-500k	95-100%	Fine-tuning, convergence

Our test run achieved 100% success rate after 500k steps, with an average goal distance of 4.6mm (the success threshold is 50mm), and throughput of approximately 1,300 steps/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 tow
`rollout/success_rate`	0 -> 1 over training	The primary success metric
`train/value_loss`	High initially, then decreasing	Critic is learning to predict returns
`train/approx_kl`	Small (< 0.03), occasional spikes OK	Measures how much the policy ch
`train/clip_fraction`	0.1-0.3	Some updates clipped, not all
`train/entropy_loss`	Slowly moves toward 0	Policy becoming more determinist

Remember, these are the same quantities you implemented in the Build It sections. `train/value_loss` is the output of your `compute_value_loss`. `train/clip_fraction` comes from your `compute_ppo_loss`. You know exactly what these numbers mean because you have computed them yourself.

Verifying results

After training completes, check the evaluation JSON:

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

Key fields to verify:

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

The passing criteria are: success rate above 90%, mean return above -10, and final distance below 0.02 meters. Our runs consistently exceed these thresholds. If yours do not, see What Can Go Wrong below.

What the trained policy does

The trained network maps the 16D concatenated observation (10D proprioceptive + 3D achieved goal + 3D desired goal) to 4D actions (dx, dy, dz, gripper). It has learned that to reach a goal, it should output velocities that point toward the goal position -- subtracting its current position from the desired position and scaling appropriately. The network discovered this purely from trial and error, using the dense reward signal as guidance.

What does this look like in practice? The robot arm starts at a default position. At each timestep, the policy sees the current gripper position (in `achieved_goal`) and the target position (in `desired_goal`), and outputs a 4D action that moves the gripper toward the target. Within about 10-15 steps (out of 50 per episode), the gripper reaches the target and holds position for the remaining steps. The gripper dimension (index 3) is largely irrelevant for Reach -- there is nothing to grasp -- so the policy typically outputs near-zero values for it.

The final distance of 4.6mm (0.0046 meters) means the policy overshoots the target by less than 5mm on average. The success threshold is 50mm (0.05 meters), so the policy is roughly 10x more precise than required. This margin gives us confidence that the solution is robust, not barely passing.

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 dictionary 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

This is the pipeline baseline. Every future chapter builds on this infrastructure. When something goes wrong with SAC in Chapter 4 or HER in Chapter 5, you can always come back here and verify that the foundation still works.

3.11 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.

`ep_rew_mean` flatlines near -20 for the entire run

Likely cause. The environment is misconfigured -- wrong observation or action shapes, or the policy network is not receiving goal information.

Diagnostic. Print the observation structure:

```
obs, _ = env.reset()
print({k: v.shape for k, v in obs.items()})
```

You should see `observation (10,)`, `achieved_goal (3,)`, `desired_goal (3,)`. Also verify that SB3 is using `MultiInputPolicy`, not `MlpPolicy` -- the latter cannot handle dictionary observations and will fail silently.

Success rate stays at 0% after 200k steps

Likely cause. You are using the wrong environment ID -- `FetchReach-v4` (sparse) instead of `FetchReachDense-v4` (dense). PPO cannot learn effectively from sparse rewards alone on this task.

Diagnostic. Print the reward from a random step. Dense rewards should be in the range [-1, 0] (negative distance). Sparse rewards are exactly 0 or -1. If you see only 0s and -1s, you have the sparse variant.

`value_loss` explodes (above 100) early in training

Likely cause. The reward scale is unexpected, or the GAE returns are not computed correctly.

Diagnostic. Check the reward range with a random policy. FetchReachDense rewards should be in [-1, 0]. If you see rewards on the order of -1000, something is miscon-

figured. Also check that GAE returns fall in a reasonable range (roughly [-50, 0] for FetchReachDense).

`approx_kl` consistently above 0.05

Likely cause. The learning rate is too high -- the policy is changing too fast per update.

Diagnostic. Reduce `learning_rate` from 3e-4 to 1e-4, or reduce `n_epochs` from 10 to 5. Either change limits how much the policy can move per update cycle.

`clip_fraction` near 1.0 every update

Likely cause. Updates are too aggressive. Nearly all actions are being clipped.

Diagnostic. Reduce the learning rate. If that does not help, reduce `clip_range` from 0.2 to 0.1. Also verify that advantages are normalized (subtracting mean, dividing by standard deviation) -- unnormalized advantages with large magnitudes can push ratios far from 1.0.

`clip_fraction` always 0.0

Likely cause. The policy is not learning. The learning rate may be too low, or the optimizer may not be attached to the model parameters.

Diagnostic. Check that `grad_norm` is nonzero in the training logs. If it is zero, gradients are not flowing -- verify that the loss tensor is connected to the model parameters (no `.detach()` in the wrong place).

`entropy_loss` immediately goes to 0

Likely cause. The policy collapsed to deterministic -- the `log_std` parameter went to negative infinity.

Diagnostic. Add an entropy coefficient: `ent_coef=0.01`. This penalizes overly deterministic policies and keeps the standard deviation from collapsing.

Training very slow (below 300 fps on GPU)

Likely cause. The GPU may not be in use, or `n_envs` is too low.

Diagnostic. Run `nvidia-smi` inside the container and check for a python process using GPU memory. If the GPU is being used but throughput is still 500-1300 fps, that is normal -- RL training on Fetch is CPU-bound on MuJoCo simulation, not GPU-bound on neural network operations. This is expected behavior, not a bug. With small networks (5k parameters) and batch sizes (256), GPU operations complete in microseconds while the CPU runs physics.

`--compare-sb3` shows mismatch above 1e-6

Likely cause. Episode boundary handling differs between our GAE and SB3's implementation.

Diagnostic. Check that the done mask is applied to both `next_value` and `last_gae` in the backward loop. SB3 uses an `episode_starts` convention that may align slightly differently with dones. If the mismatch is small (1e-5 to 1e-4), it is likely a floating-point precision issue and not a cause for concern.

Build It `--verify` fails on "Value loss should decrease"

Likely cause. The random seed produced an adversarial batch where 10 updates are not enough to improve predictions.

Diagnostic. Re-run -- the test uses random data and very occasionally hits an edge case. If the failure is persistent across multiple runs, check that `optimizer.step()` is being called and that model parameters are changing between iterations. You can verify this by printing the sum of parameters before and after: `sum(p.sum() for p in model.parameters())`.

3.12 Summary

This chapter derived PPO from first principles and built it from the ground up. Here is what you accomplished:

- **The objective.** We want to maximize expected discounted return $J(\theta)$. The policy gradient theorem tells us how to differentiate this, and the advantage function tells us which actions are better than average.
- **The instability problem.** Vanilla policy gradient is unstable because noisy advantage estimates can cause destructively large policy updates. PPO's clipped surrogate objective constrains the probability ratio ρ_t to $[1-\epsilon, 1+\epsilon]$, preventing catastrophic updates while still allowing improvement.
- **From-scratch implementation.** You built six components: the actor-critic network, GAE advantage computation, the clipped policy loss, the value loss, the combined update step, and the training loop. Each was verified with concrete checks before moving to the next.
- **Bridge to SB3.** The bridging proof confirmed that our GAE implementation and SB3's `RolloutBuffer` produce identical advantages. SB3 adds engineering features (vectorized environments, dict-observation handling, learning rate scheduling) but computes the same underlying math.
- **Pipeline validation.** PPO achieved 100% success rate on FetchReachDense-v4, establishing that the environment, network architecture, training loop, evaluation protocol, and GPU setup all work correctly.
- **On-policy limitation.** PPO discards all data after each update because it is on-policy. This is sample-inefficient -- millions of simulation steps are thrown away after a single use.

That last point is the gap that Chapter 4 addresses. SAC (Soft Actor-Critic) is off-policy: it stores transitions in a replay buffer and reuses them across many updates. This means every simulation step contributes to learning not once, but repeatedly. On FetchReachDense, you will see SAC converge faster than PPO using less total simulation. More importantly, the replay buffer is a prerequisite for Chapter 5's Hindsight Experience Replay (HER), which turns failed trajectories into learning signal by relabeling goals -- a technique that requires stored transitions to relabel.

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/ch02_ppo_dense_reach.py all \
      --seed $seed --total-steps 1000000
  done

Hardware:    Any modern GPU (tested on NVIDIA GB10; CPU works but ~6x slower)
Time:          ~8 min per seed (GPU), ~45 min per seed (CPU)
Seeds:         0, 1, 2

Artifacts produced:
  checkpoints/ppo_FetchReachDense-v4_seed{0,1,2}.zip
  checkpoints/ppo_FetchReachDense-v4_seed{0,1,2}.meta.json
  results/ch02_ppo_fetchreachdense-v4_seed{0,1,2}_eval.json
  runs/ppo/FetchReachDense-v4/seed{0,1,2}/

Results summary (what we got):
  success_rate: 1.00 +/- 0.00  (3 seeds x 100 episodes)
  return_mean:  -0.40 +/- 0.05
  final_distance_mean: 0.005 +/- 0.001

If your numbers differ by more than ~5%, 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 the experiment card:

```
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all \
  --seed 0 --total-steps 500000
```

Check the eval JSON: `success_rate` should be >= 0.90, `mean_return` > -10. Record your training time and steps per second -- you will compare these against SAC in Chapter 4.

2. (Tweak) GAE lambda ablation.

In the lab verification, modify `gae_lambda` and observe the effect on advantage magnitudes:

- `gae_lambda = 0.0` (one-step TD, high bias)
- `gae_lambda = 0.5` (midpoint)
- `gae_lambda = 1.0` (Monte Carlo, high variance)

Question: How do the advantage magnitudes change? Why does `lambda=0` produce smaller magnitude advantages?

Expected: `lambda=0` advantages are dominated by single TD residuals (one step of reward minus one step of value change). `lambda=1` advantages accumulate over the full trajectory, producing larger magnitudes and more variance between samples.

3. (Tweak) Clip range ablation.

Train PPO with different `clip_range` values (0.1, 0.2, 0.4) for 500k steps each. Compare:

- (a) Final success rate
- (b) Training stability (watch `approx_kl` in TensorBoard)
- (c) `clip_fraction` values

Expected: `clip_range=0.1` is more conservative -- the policy changes slowly per update, which may require more iterations but is stabler. `clip_range=0.4` allows larger changes per update, which risks instability but may converge faster on easy tasks. The default of 0.2 is a compromise that works reliably across many tasks.

4. (Extend) Add wall-clock time tracking.

Modify the eval report to include wall-clock training time and compute steps per second. Compare GPU versus CPU performance. Expected: GPU is roughly 2-5x faster, but the speedup is modest because the bottleneck is CPU-bound MuJoCo simulation, not GPU-bound neural network operations.

5. (Challenge) Train the from-scratch implementation on FetchReachDense.

Extend the `--demo` mode in `ppo_from_scratch.py` to work with `FetchReachDense-v4` instead of CartPole. You will need to handle dictionary observations (concatenate `observation` + `desired_goal` as the network input, giving a 13D vector) and continuous actions (remove the discrete threshold used for CartPole). Does it learn? How does it compare to SB3 in sample efficiency?

Expected: it should learn but more slowly than SB3 -- SB3 uses vectorized environments (8 parallel), optimized rollout storage, and proper observation preprocessing. With a single environment, your from-scratch implementation may reach 50-80% success rate in 500k steps. The performance gap is engineering, not algorithmic -- both implementations compute the same math.