Reinforcement Learning

Lecture 10: Proximal Policy Optimization (PPO)

Taehoon Kim

Sogang University MIMIC Lab
https://mimic-lab.com

Fall Semester 2025

Today's Agenda

- Course recap & PPO motivation
- Trust region intuition for policy updates
- PPO objective design and clipping analysis
- Advantage estimation with GAE
- Implement the PPO training loop
- Extend PPO to continuous control tasks
- Tune hyperparameters systematically
- Debug and benchmark trained agents

Learning Objectives

By the end of this lecture, you will be able to:

- **Understand** the motivation and theory behind PPO
- **Implement** PPO with clipped surrogate objective
- **Apply** Generalized Advantage Estimation (GAE)
- **Extend** PPO to continuous control problems
- **Debug** common PPO training issues
- **Tune** hyperparameters systematically

Prerequisites:

- Policy gradient methods (Lecture 8-9)
- Actor-critic architectures
- PyTorch and Gymnasium

Why Proximal Policy Optimization?

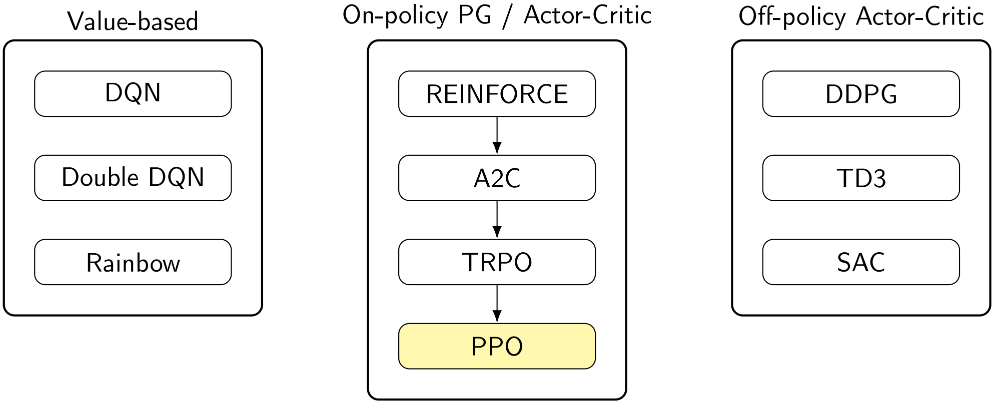
Problems with Vanilla Policy Gradients:

- High variance in gradient estimates
- Unstable learning with large policy updates
- Poor sample efficiency
- Sensitive to hyperparameters

PPO Advantages:

- Simple to implement and tune
- Stable training with clipped updates
- Good sample efficiency
- Works on both discrete & continuous control

PPO in the RL Landscape



PPO bridges the gap: Simple like policy gradients, stable like actor-critic

Policy Gradient Foundation

Vanilla Policy Gradient (REINFORCE):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \right]$$

With baseline (reduces variance):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t)) \right]$$

Advantage Actor-Critic:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t \right]$$

where $A_t = Q(s_t, a_t) - V(s_t)$ is the advantage function.

The Problem with Large Policy Updates

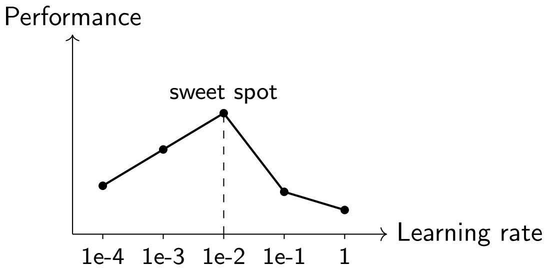
Policy Collapse Example:

```python
# Large learning rate
optimizer = Adam(lr=0.1)

# One large update
loss = -log_probs * advantages
loss.backward()
optimizer.step()

# Result: Policy changes too much
# Performance drops dramatically
```

Why does this happen?

- Policy distribution shifts drastically
- Data becomes off-policy
- Advantage estimates become invalid

Learning Rate vs Performance:



Sweet spot exists!

Trust Region Motivation

Key Insight: Limit how much the policy can change in each update

Trust Region Policy Optimization (TRPO):

$$\max_{\theta} \mathbb{E}_s \left[\mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} [\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A^{\pi_{\theta_{\text{old}}}}(s, a)] \right]$$

Subject to: $\mathbb{E}_s[D_{KL}(\pi_{\theta_{\text{old}}}(\cdot|s)||\pi_\theta(\cdot|s))] \leq \delta$

Problems with TRPO:

- Requires second-order optimization (conjugate gradient)
- Complex to implement correctly
- Computationally expensive
- Sensitive to hyperparameter δ

PPO Solution: Replace constraint with clipping!

PPO Clipped Surrogate Objective

Importance sampling ratio:

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

Clipped surrogate objective:

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

Key parameters:

- ϵ (clip range): typically 0.1-0.3
- $r_t(\theta) = 1$ when policies are identical
- Denominator uses fixed old policy $\pi_{\theta_{\text{old}}}$ (no gradients flow); only numerator π_θ is updated
- Clipping prevents $r_t(\theta)$ from going too far from 1

Intuition:

- If advantage is positive: limit how much probability can increase
- If advantage is negative: limit how much probability can decrease

Clipped Objective: Case Analysis

Case 1: Positive advantage ($A_t > 0$)

- Objective term: $L_t^{CLIP} = \min(r_t A_t,\ \text{clip}(r_t,\ 1-\epsilon,\ 1+\epsilon)A_t)$
- For $r_t \leq 1+\epsilon$: behaves like vanilla policy gradient $r_t A_t$
- For $r_t > 1+\epsilon$: objective becomes flat at $(1+\epsilon)A_t$
- **Effect:** Prevents overly aggressive increases in action probability

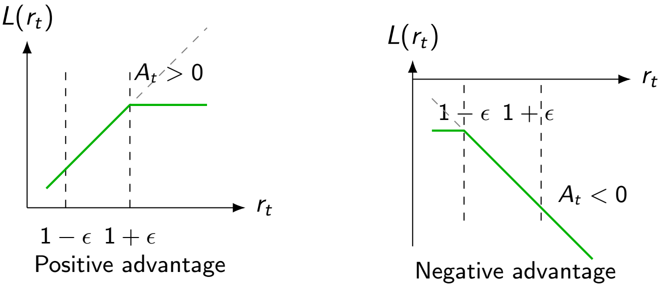
Case 2: Negative advantage ($A_t < 0$)

- For $r_t \geq 1-\epsilon$: behaves like $r_t A_t$ (reduces probability of bad actions)
- For $r_t < 1-\epsilon$: objective becomes flat at $(1-\epsilon)A_t$
- **Effect:** Avoids collapsing the policy by over-penalizing actions

Summary: PPO trusts the direction of A_t, but only within a local window around $r_t = 1$.

Understanding the Clipping Mechanism

Clipped Surrogate for Positive vs Negative Advantage



Green line = PPO objective $\min\left(r_t A_t,\ \text{clip}(r_t, 1-\epsilon, 1+\epsilon)A_t\right)$

Complete PPO Objective Function

Full PPO loss combines three terms:

$$\mathcal{L}(\theta, \phi) = -\mathbb{E}_t[L^{CLIP}(\theta)] + c_v \mathbb{E}_t[L^{VF}(\phi)] - c_e \mathbb{E}_t[L^{ENT}(\theta)]$$

1. Policy Loss (Clipped):

$$L^{CLIP}(\theta) = \min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)$$

2. Value Function Loss:

$$L^{VF}(\phi) = (V_\phi(s_t) - V_t^{\text{target}})^2$$

3. Entropy Loss (exploration):

$$L^{ENT}(\theta) = \mathbb{E}_t[\mathcal{H}(\pi_\theta(\cdot|s_t))]$$

Typical coefficients: $c_v = 0.5$, $c_e = 0.01$

Value Function Clipping (Optional)

Motivation: Prevent large value function updates
Clipped value loss:

$$L^{VF-CLIP}(\phi) = \max\left((V_\phi(s_t) - V_t^{\text{target}})^2, (V_{\phi_{\textbf{old}}}(s_t) + \text{clip}(V_\phi(s_t) - V_{\phi_{\textbf{old}}}(s_t), -\epsilon_v, \epsilon_v) - V_t^{\text{target}})^2\right)$$

Benefits:

- More stable critic learning
- Prevents value function from changing too rapidly
- Often improves sample efficiency

Drawbacks:

- Can slow convergence if clip range too small
- Additional hyperparameter to tune

Common practice: Use same ϵ for policy and value clipping

Generalized Advantage Estimation (GAE)

Problem: How to compute advantage $A_t = Q(s_t, a_t) - V(s_t)$?
GAE trades off bias and variance:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

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

Key parameter λ:

- $\lambda = 0$: $\hat{A}_t = \delta_t$ (high bias, low variance)
- $\lambda = 1$: $\hat{A}_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} - V(s_t)$ (low bias, high variance)
- $\lambda = 0.95$: Good balance (common choice)

Recursive computation:

$$\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$$

In practice, GAE is computed efficiently by starting at the final timestep T and moving backward using this recursion.

GAE Example: Three-Step Trajectory

Toy example: 3-step episode, $\gamma = 0.9$, $\lambda = 0.9$

- Rewards: $r_0 = 1,\ r_1 = 1,\ r_2 = 1$
- Value estimates: $V(s_0) = 0.5,\ V(s_1) = 0.5,\ V(s_2) = 0.5,\ V(s_3) = 0$

Step 1: TD errors

$$\delta_2 = r_2 + \gamma V(s_3) - V(s_2),\ \delta_1 = r_1 + \gamma V(s_2) - V(s_1),\ \delta_0 = r_0 + \gamma V(s_1) - V(s_0)$$

Step 2: Backward recursion

$$\hat{A}_2 = \delta_2, \quad \hat{A}_1 = \delta_1 + \gamma\lambda\hat{A}_2, \quad \hat{A}_0 = \delta_0 + \gamma\lambda\hat{A}_1$$

Takeaway: GAE mixes TD errors and multi-step returns in a simple backward pass.

GAE Implementation

```python
def compute_gae(rewards, values, dones, next_value, gamma=0.99, lam=0.95):
    """Compute Generalized Advantage Estimation"""
    advantages = []
    gae = 0

    # Work backwards through the episode
    for t in reversed(range(len(rewards))):
        if t == len(rewards) - 1:
            next_non_terminal = 1.0 - dones[t]
            next_value_t = next_value
        else:
            next_non_terminal = 1.0 - dones[t + 1]
            next_value_t = values[t + 1]

        # TD error
        delta = rewards[t] + gamma * next_value_t * next_non_terminal - values[t]

        # GAE update
        gae = delta + gamma * lam * next_non_terminal * gae
        advantages.insert(0, gae)

    return advantages
```

PPO Workflow: Data Collection

Key loop in `exp04_ppo_implementation.py`

1. Initialise the policy π_θ and value function V_ϕ.
2. For each iteration:
    - Roll out the current policy to gather trajectories.
    - Compute advantages with GAE and bootstrap targets V_t^{target}.
    - Normalise $\hat{A}_t$ so gradients have consistent scale.

PPO Workflow: Optimisation Loop

Inner updates from `exp04_ppo_implementation.py`

1. Run K epochs of minibatch SGD over the collected rollout batch.
2. Evaluate ratios $r_t = \pi_\theta(a_t|s_t)/\pi_{\theta_{\mathbf{old}}}(a_t|s_t)$.
3. Optimise the clipped surrogate L^{CLIP} plus value and entropy terms.
4. Clip gradients (max norm 0.5) and optionally stop early when KL grows.

Outcome: Stable policy improvement without large behaviour shifts.

Key Implementation Details

Critical for successful PPO training:

Data Collection:

- Vectorized environments
- Rollout length: 128-2048 steps
- Batch size: num_envs $\times$ num_steps

Advantage Processing:

- GAE with $\lambda = 0.95$
- Advantage normalization
- Bootstrap from value function

Training:

- Multiple epochs (4-10) per batch
- Minibatch SGD with shuffling
- Gradient clipping (max norm = 0.5)
- Early stopping on KL divergence

Hyperparameters:

- Learning rate: 2.5e-4
- Clip range: 0.2
- Entropy coefficient: 0.01

Important: These details matter greatly for performance!

Implementation Session: PPO from Scratch

What we'll implement together:

1. **Actor-Critic Network**
   - Shared feature extractor
   - Policy head (discrete actions)
   - Value head
2. **Rollout Buffer**
   - Store trajectories from vectorized envs
   - Compute GAE advantages
   - Prepare training batches
3. **PPO Training Loop**
   - Clipped surrogate objective
   - Multiple epochs per batch
   - Early stopping on KL divergence

Follow along: `exp04_ppo_implementation.py`

Actor-Critic Architecture

```python
class ActorCritic(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden_sizes=(64, 64)):
        super().__init__()

        # Shared feature extractor
        layers = []
        in_dim = obs_dim
        for hidden_dim in hidden_sizes:
            layers.extend([nn.Linear(in_dim, hidden_dim), nn.Tanh()])
            in_dim = hidden_dim
        self.feature_extractor = nn.Sequential(*layers)

        # Policy and value heads
        self.actor = nn.Linear(in_dim, act_dim)    # logits
        self.critic = nn.Linear(in_dim, 1)        # value

    def forward(self, x):
        features = self.feature_extractor(x)
        return self.actor(features), self.critic(features)
```

Actor-Critic: Sampling Utilities

```python
class ActorCritic(nn.Module):
    ...

    def get_action_and_value(self, x):
        logits, value = self.forward(x)
        dist = torch.distributions.Categorical(logits=logits)
        action = dist.sample()
        return action, dist.log_prob(action), value
```

Reference implementation: `exp04_ppo_implementation.py`

Rollout Collection (Setup)

```python
def collect_rollouts(agent, envs, num_steps):
    observations, actions, logprobs, rewards, dones, values = [], [], [], [], [], []

    next_obs, _ = envs.reset()
    next_done = torch.zeros(num_envs)
    # Loop over rollout steps shown on the next slide
```

Rollout Collection (Loop)

```python
    for step in range(num_steps):
        obs = next_obs

        # Get action from current policy
        with torch.no_grad():
            action, logprob, value = agent.get_action_and_value(obs)

        # Environment step
        next_obs, reward, terminated, truncated, infos = envs.step(action.numpy())
        done = np.logical_or(terminated, truncated)

        # Store step
        observations.append(obs)
        actions.append(action)
        logprobs.append(logprob)
        rewards.append(torch.tensor(reward))
        dones.append(torch.tensor(done, dtype=torch.float))
        values.append(value)

        next_obs = torch.tensor(next_obs, dtype=torch.float32)
        next_done = torch.tensor(done, dtype=torch.float32)

    return observations, actions, logprobs, rewards, dones, values, next_obs, next_done
```

Source: `exp04_ppo_implementation.py`

Old vs New Policy in Code

Implementing the ratio $r_t(\theta)$ in practice:

- During rollout, store:
  - Actions a_t
  - Old log-probabilities $\log \pi_{\theta_{\textbf{old}}}(a_t|s_t)$ (detached)
  - Value estimates $V_{\phi_{\textbf{old}}}(s_t)$
- During updates, recompute with the current policy:
  - New logits and $\log \pi_\theta(a_t|s_t)$
  - Ratios $r_t = \exp(\log \pi_\theta - \log \pi_{\theta_{\textbf{old}}})$
- Old values and log-probs are constants (no gradients).

Practical tip: Detach old log-probs in PyTorch when storing them in the rollout buffer.

PPO Update (Batch Preparation)

```python
def ppo_update(agent, optimizer, batch_data, clip_coef=0.2, epochs=4):
    obs, actions, old_logprobs, advantages, returns, old_values = batch_data

    # Normalize advantages
    advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

    for epoch in range(epochs):
        # Shuffle data
        indices = torch.randperm(len(obs))

        for start in range(0, len(obs), minibatch_size):
            end = start + minibatch_size
            mb_indices = indices[start:end]

            # Get current policy outputs
            logits, values = agent(obs[mb_indices])
            dist = torch.distributions.Categorical(logits=logits)
            new_logprobs = dist.log_prob(actions[mb_indices])
            entropy = dist.entropy().mean()
```

PPO Update (Losses & Optimisation)

```python
            # Compute ratios and losses
            ratio = torch.exp(new_logprobs - old_logprobs[mb_indices])

            # Clipped surrogate
            surr1 = ratio * advantages[mb_indices]
            surr2 = torch.clamp(ratio, 1-clip_coef, 1+clip_coef) * advantages[mb_indices]
            policy_loss = -torch.min(surr1, surr2).mean()

            # Value loss
            value_loss = F.mse_loss(values.squeeze(), returns[mb_indices])

            # Total loss
            loss = policy_loss + 0.5 * value_loss - 0.01 * entropy

            # Update
            optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(agent.parameters(), 0.5)
            optimizer.step()
```

Full script: `exp04_ppo_implementation.py`

Common PPO Issues and Debugging

Key metrics to monitor:

Policy Metrics:

- KL divergence (< 0.05)
- Clip fraction (0.1 - 0.3)
- Policy entropy (decreasing)
- Importance ratios (near 1.0)

Training Metrics:

- Gradient norms (0.1 - 10)
- Value function error
- Advantage statistics
- Episode returns

Common Problems:

- Too high learning rate $\rightarrow$ instability
- Too low clip range $\rightarrow$ slow learning
- No entropy $\rightarrow$ premature convergence
- Poor advantage estimation $\rightarrow$ high variance

Debug Script:

`exp07_debugging_techniques.py`

Rule: Always check KL divergence and clip fraction first! **Interpreting KL and clip fraction:**

- Large KL and high clip fraction $\rightarrow$ updates too aggressive; reduce learning rate or clip range
- Very small KL and near-zero clip fraction $\rightarrow$ updates too conservative; increase learning rate or clip range

Extending PPO to Continuous Control

Key differences for continuous actions:

Discrete Actions:

- Policy outputs: logits
- Distribution: Categorical
- Action sampling: argmax or sample
- Action space: $\{0, 1, 2, \ldots, n-1\}$

Example environments:

- CartPole-v1
- LunarLander-v2
- Atari games

Continuous Actions:

- Policy outputs: mean, std
- Distribution: Gaussian (Normal)
- Action sampling: $a \sim \mathcal{N}(\mu, \sigma)$
- Action space: $\mathbb{R}^d$ (bounded)

Example environments:

- Pendulum-v1
- BipedalWalker-v3
- MuJoCo robotics

Implementation: `exp08_continuous_control.py`

Gaussian Policy (Network Definition)

```python
class ContinuousActorCritic(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden_sizes=(64, 64)):
        super().__init__()

        # Shared features
        self.feature_extractor = build_mlp(obs_dim, hidden_sizes)

        # Policy head - outputs mean
        self.actor_mean = nn.Linear(hidden_sizes[-1], act_dim)

        # Log standard deviation (learnable parameter)
        self.actor_logstd = nn.Parameter(torch.zeros(act_dim))

        # Value head
        self.critic = nn.Linear(hidden_sizes[-1], 1)
```

Gaussian Policy (Sampling & Value)

```
class ContinuousActorCritic(nn.Module):
    ...

    def get_action_and_value(self, x, action=None):
        features = self.feature_extractor(x)

        action_mean = self.actor_mean(features)
        action_std = torch.exp(self.actor_logstd)

        # Create Gaussian distribution
        dist = torch.distributions.Normal(action_mean, action_std)

        if action is None:
            action = dist.sample()

        log_prob = dist.log_prob(action).sum(dim=-1)  # Sum over action dims
        entropy = dist.entropy().sum(dim=-1)
        value = self.critic(features).squeeze(-1)

        return action, log_prob, entropy, value
```

Script reference: `exp08_continuous_control.py`

Handling Action Bounds

Problem: Most continuous environments have bounded action spaces

Solution Options:

1. **Clipping (Simple):**
   - Sample from Gaussian, then clip: $a = \text{clip}(a_{\text{raw}}, a_{\text{min}}, a_{\text{max}})$
   - Pro: Easy to implement
   - Con: Breaks differentiability, can cause issues
2. **Tanh Squashing (Better):**
   - $a = \tanh(a_{\text{raw}}) \cdot \text{scale} + \text{bias}$
   - Include Jacobian correction in log probability
   - Pro: Smooth, differentiable
   - Con: Slightly more complex

   In research and production settings, log-probability Jacobian correction is essential, but for coursework-level implementations, using tanh squashing without correction still yields reasonable performance.
3. **Beta Distribution:**
   - Naturally bounded to $[0, 1]$, then rescale
   - Pro: Theoretically clean
   - Con: Less commonly used

Recommendation: Use tanh squashing for best results

PPO Hyperparameter Sensitivity Analysis

Critical hyperparameters ranked by sensitivity:

1. **Learning Rate** (most sensitive)
    - Range: 10^{-5} to 10^{-3}
    - Sweet spot: 2.5×10^{-4}
    - Effect: Too high $\rightarrow$ instability, too low $\rightarrow$ slow learning
2. **Clip Range (ϵ)**
    - Range: 0.1 to 0.3
    - Sweet spot: 0.2
    - Effect: Too low $\rightarrow$ conservative updates, too high $\rightarrow$ instability
3. **Batch Size (num_envs $\times$ num_steps)**
    - Range: 512 to 8192
    - Sweet spot: 1024-2048
    - Effect: Larger $\rightarrow$ more stable but less responsive

Systematic tuning: `exp06_hyperparameter_sensitivity.py`

Systematic Hyperparameter Tuning

Recommended tuning order:

1. **Start with defaults:**
   - Learning rate: 2.5e-4
   - Clip range: 0.2
   - GAE lambda: 0.95
   - Batch size: 2048
2. **Tune learning rate first:**
   - Try: [1e-4, 2.5e-4, 5e-4]
   - Look for stable learning curves
3. **Adjust batch size if needed:**
   - Larger for more stable environments
   - Smaller for faster iteration
4. **Fine-tune other parameters:**
   - Entropy coefficient (exploration)
   - Number of epochs per update
   - Value function coefficient

Key principle: Change one parameter at a time!

Advanced PPO Techniques

Performance optimizations:

Computational:

- Vectorized environments
- Mixed precision training (AMP)
- torch.compile() for faster execution
- Gradient accumulation

Algorithmic:

- Learning rate annealing
- Reward scaling/normalization
- Observation normalization
- Dual clip (PPO-M)

Engineering:

- Proper logging and monitoring
- Checkpointing and resumption
- Distributed training
- Reproducibility measures

Variants:

- PPO with Curiosity
- PPO with Hindsight Experience Replay
- Recurrent PPO (for partial observability)

Production considerations: Monitoring, robustness, deployment

PPO Performance Benchmarks

Expected performance on standard environments:

Environment	PPO Score	Random Score	Timesteps
CartPole-v1	400+	20	100K
LunarLander-v2	200+	-150	500K
BipedalWalker-v3	300+	-100	2M
Pendulum-v1	-200	-1500	200K
HalfCheetah-v2	2000+	-300	2M

Training tips for good performance:

- Use multiple random seeds (3-5) for reliable results
- Monitor clip fraction and KL divergence during training
- Ensure entropy decreases gradually (not too fast)
- Value function should learn to predict returns accurately

Comparison with other algorithms:

- PPO vs A2C: More stable, better sample efficiency
- PPO vs SAC: Simpler, works well for both discrete and continuous
- PPO vs DQN: Better for continuous control, policy-based exploration

PPO in the Real World: Games & Robotics

Game AI:

- OpenAI Five (Dota 2)
- DeepMind AlphaStar (StarCraft II)
- Hide-and-seek environments
- Minecraft and other sandbox games

Robotics:

- Manipulation and grasping policies
- Locomotion control for legged robots
- Drone navigation and trajectory following
- Humanoid balance and walking controllers

PPO in the Real World: NLP & Beyond

Language models and RLHF:

- Reinforcement Learning from Human Feedback (ChatGPT, GPT-4)
- Text summarisation and dialogue systems
- Code generation with preference optimisation

Other domains:

- Autonomous driving and fleet control
- Quantitative trading and resource allocation
- Scientific discovery and experiment design

Why PPO is popular:

- Robust across diverse tasks with minimal tuning
- Straightforward to implement and debug
- Balanced sample efficiency and stability
- Excellent open-source support and baselines

The clipped surrogate objective and GAE you learned today are core components underlying many RLHF-style training pipelines for large language models.

Implementation Best Practices: Architecture & Training

Code organization highlights:

- Choose between shared or separate actor/critic backbones.
- Use orthogonal weight initialisation for stability.
- Normalise layers or observations when gradients explode.
- Keep the training loop vectorised with shuffled minibatches.
- Stop early when KL divergence exceeds the trust-region budget.

Implementation Best Practices: Monitoring & Reliability

Monitor during training:

- Episode returns/lengths, policy entropy, gradient norms.
- Value prediction error and actual clip fraction.

Reproducibility checklist:

- Fix random seeds and deterministic flags.
- Log environment versions and hyperparameters.
- Track experiment configs alongside checkpoints.

Avoid these pitfalls:

- Skipping advantage normalisation or mishandling episode termini.
- Incorrect importance sampling ratios or too few parallel envs.

PPO vs Other RL Algorithms

Algorithm	Sample Eff.	Stability	Simplicity	Generality
PPO	Good	High	High	High
A2C/A3C	Moderate	Moderate	High	High
TRPO	Good	High	Low	High
SAC	Very Good	High	Moderate	Moderate
TD3	Very Good	High	Moderate	Low
DQN	Good	Moderate	Moderate	Low

When to choose PPO:

- Need both discrete and continuous control
- Want stable, reliable training
- Implementation simplicity is important
- Have sufficient computational resources for on-policy learning

When to consider alternatives:

- **SAC/TD3:** Need maximum sample efficiency for continuous control
- **DQN:** Discrete control with very limited compute
- **TRPO:** Need theoretical guarantees (research)

Final Integration and Testing

Let's build a complete PPO system:

File: `exp09_final_integration.py`

Features we'll implement:

1. Unified PPO class supporting discrete and continuous control
2. Comprehensive benchmarking suite
3. Hyperparameter sensitivity analysis
4. Model saving and loading
5. Performance visualization
6. Reproducibility testing

Environments we'll test:

- CartPole-v1 (discrete, simple)
- Pendulum-v1 (continuous, simple)
- LunarLander-v2 (discrete, complex)

Goal: Production-ready PPO implementation

Live Benchmarking Results

Running comprehensive benchmark...

Metrics we're tracking:

- Final episode return (mean $\pm$ std over multiple seeds)
- Training stability (coefficient of variation)
- Sample efficiency (timesteps to reach threshold)
- Wall-clock training time

Quality checks:

- Reproducibility across random seeds
- Monotonic improvement in early training
- Reasonable final performance vs literature
- Stable convergence (no catastrophic forgetting)

Expected results:

- CartPole: 450 ± 50 (out of 500 max)
- Pendulum: -200 ± 50 (higher is better)

Troubleshooting Common Issues

Let's debug together:

Scenario 1: Policy not learning (flat learning curve)

- Check: Learning rate too low, entropy too high, poor advantage estimation
- Debug: Monitor KL divergence, clip fraction, gradient norms

Scenario 2: Training unstable (high variance)

- Check: Learning rate too high, batch size too small, no advantage normalization
- Debug: Plot importance sampling ratios, value function accuracy

Scenario 3: Good training, poor evaluation

- Check: Overfitting, environment randomization, evaluation protocol
- Debug: Compare training vs evaluation environments

Debug script: `exp07_debugging_techniques.py`

PPO Extensions and Research Directions

Active research areas:

Algorithmic improvements:

- PPO with KL warmup
- Adaptive clipping schedules
- Multi-step returns
- Curiosity-driven exploration

Scalability:

- Distributed PPO
- GPU-accelerated environments
- Large batch training
- Mixture of experts policies

Applications:

- Multi-agent PPO (MAPPO)
- Hierarchical PPO
- Meta-learning with PPO
- Safe reinforcement learning

Theoretical analysis:

- Convergence guarantees
- Sample complexity bounds
- Policy improvement theory
- Connection to natural gradients

Key Takeaways from This Lecture

Theoretical insights:

- PPO elegantly solves trust region optimization with simple clipping
- GAE provides excellent bias-variance tradeoff for advantage estimation
- Importance sampling enables off-policy-like efficiency with on-policy data

Practical skills:

- How to implement production-ready PPO from scratch
- Debugging techniques using KL divergence and clip fraction
- Extension to continuous control with Gaussian policies
- Systematic hyperparameter tuning methodology

Implementation details matter:

- Advantage normalization is crucial for stability
- Multiple epochs per batch improves sample efficiency
- Proper environment vectorization enables fast training
- Early stopping prevents policy degradation

Next Week: Advanced Policy Methods

Lecture 11 Preview - Advanced Policy Optimization

Topics we'll cover:

- RLHF and DPO for language models
- Multi-agent PPO (MAPPO)
- Monte Carlo Tree Search (MCTS)
- AlphaZero and MuZero

Applications:

- ChatGPT training pipeline
- Game AI (chess, Go, poker)
- Strategic decision making
- Planning with learned models

Prerequisites for next week:

- Complete today's lab exercises
- Solid understanding of PPO
- Familiarity with transformer architectures (helpful)

Preparation:

- Review RLHF paper (Christiano et al.)
- Install transformers library
- Practice PPO debugging

Building towards: Complete RL practitioner skillset

Lecture 10 Summary

What we accomplished today:

1. **Theory:** Understood PPO's clipped surrogate objective and GAE
2. **Implementation:** Built PPO from scratch with proper components
3. **Extensions:** Extended to continuous control with Gaussian policies
4. **Debugging:** Learned systematic debugging and monitoring techniques
5. **Optimization:** Applied hyperparameter tuning and performance optimization
6. **Integration:** Created production-ready PPO with benchmarking

You now have:

- Deep understanding of modern policy optimization
- Practical implementation skills for real projects
- Debugging toolkit for reliable training
- Foundation for advanced RL methods

Ready to tackle complex RL challenges!