Reinforcement Learning

Lecture 9: Actor-Critic Methods

Taehoon Kim

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

Fall Semester 2025

Today's Agenda

- Why move beyond vanilla policy gradients
- Value baselines, advantage estimation, and GAE
- Actor-Critic architectures and vectorised A2C
- Entropy regularisation and tuning workflows
- Hands-on experiments and integrated A2C run

Learning Objectives

By the end of this lecture, you will:

Theory Understanding:

- Explain why Actor-Critic reduces variance vs vanilla policy gradients
- Derive n-step and GAE advantage estimators
- Analyze bias-variance trade-offs in bootstrapping

Practical Skills:

- Implement vectorized A2C with proper GAE computation
- Configure entropy regularization for exploration
- Train scalable agents across multiple environments
- Report ablations over entropy coefficient and return horizon

Prerequisites: Policy gradients, value functions, PyTorch basics

The Variance Problem in Policy Gradients

REINFORCE Algorithm Recap:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) \cdot G_t]$$

Problems:

- High variance in gradient estimates
- Slow convergence due to noisy updates
- Sample inefficiency

Solution Ideas:

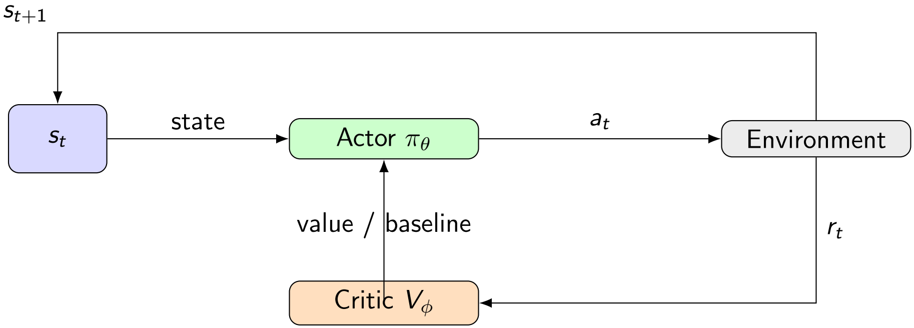
- Use a baseline to reduce variance: $G_t - b(s_t)$
- Learn the baseline from data: $b(s_t) = V(s_t)$
- Combine policy learning with value learning

$\Rightarrow$ **Actor-Critic Methods**

Actor-Critic: Best of Both Worlds

- **Actor (Policy):**
  - Learns policy $\pi_\theta(a|s)$
  - Selects actions in each state
  - Updated via policy gradients
- **Critic (Value Function):**
  - Learns value function $V_\phi(s)$
  - Evaluates how good a state (or action) is
  - Provides a learned baseline for the actor
- **Key Idea:**
  - Actor focuses on *choosing actions*
  - Critic focuses on *evaluating* those actions via $V(s)$
  - Combining them reduces variance while keeping policy gradient flexibility

Actor-Critic Information Flow



Critic learns $V_\phi(s)$ as a baseline, which is used to form advantages for updating the actor.

Mathematical Foundation

Policy Gradient with Baseline:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) \cdot (Q^\pi(s,a) - V^\pi(s))]$$

Define Advantage Function:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

Advantage Interpretation:

- How much better is action a compared to average?
- $A^\pi(s,a) > 0$: action better than average
- $A^\pi(s,a) < 0$: action worse than average
- $A^\pi(s,a) = 0$: action is average

Gradient Becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) \cdot A^\pi(s,a)]$$

Estimating Advantages: The Challenge

Problem: We don't know true $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
Solution Options:

1. **Monte Carlo:** $\hat{A}_t = G_t - V(s_t)$
   - $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$
   - Low bias, high variance
2. **TD(0):** $\hat{A}_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
   - Uses bootstrapping
   - High bias, low variance
3. **n-step:** Interpolate between MC and TD
4. **GAE:** Exponentially weighted average

N-Step Advantage Estimation

N-Step Return:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n})$$

N-Step Advantage:

$$\hat{A}_t^{(n)} = G_t^{(n)} - V(s_t)$$

Special Cases:

- $n = 1$: $\hat{A}_t^{(1)} = r_t + \gamma V(s_{t+1}) - V(s_t)$ (TD error)
- $n = \infty$: $\hat{A}_t^{(\infty)} = G_t - V(s_t)$ (Monte Carlo)

Bias-Variance Trade-off:

- Small n: Low variance, high bias
- Large n: High variance, low bias

Generalized Advantage Estimation (GAE)

Key Idea: Exponentially weighted average of all n-step advantages
GAE Formula:

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

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

$$\hat{A}_t^{\mathrm{GAE}} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{A}_t^{(n)}$$

Lambda Parameter:

- $\lambda = 0$: GAE = TD error (low variance, high bias)
- $\lambda = 1$: GAE = Monte Carlo (high variance, low bias)
- $\lambda \in [0.9, 0.97]$: Common practical range

GAE Computation (Backward Pass)

Efficient Implementation:

```
def compute_gae(rewards, values, terminated, gamma=0.99, lam=0.95):
    T, N = rewards.shape
    advantages = torch.zeros(T, N)
    last_adv = torch.zeros(N)

    for t in reversed(range(T)):
        nonterminal = (not terminated[t]).float()
        next_value = values[t+1] * nonterminal

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

        # GAE recursion
        last_adv = delta + gamma * lam * last_adv * nonterminal
        advantages[t] = last_adv

    returns = advantages + values[:-1]
    return advantages, returns
```

Key Points: Backward computation, terminal state masking, bootstrap handling

Advantage Actor-Critic (A2C) Algorithm

Synchronous A2C Steps:

1. **Collect** rollouts from N parallel environments for T steps
2. **Compute** advantages using GAE
3. **Update** actor using policy gradient with advantages
4. **Update** critic using value function regression
5. **Repeat**

Key Features:

- On-policy learning
- Parallel data collection
- Shared network architecture (optional)
- Entropy regularization for exploration

vs A3C: A2C uses synchronous updates vs A3C's asynchronous updates

A2C Loss Functions

Policy Loss (Actor):

$$L_{\text{policy}} = -\mathbb{E}[\log \pi_\theta(a_t|s_t) \cdot \hat{A}_t]$$

Value Loss (Critic):

$$L_{\text{value}} = \frac{1}{2}\mathbb{E}[(G_t - V_\phi(s_t))^2]$$

Entropy Loss (Exploration):

$$L_{\text{entropy}} = -\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot|s_t))]$$

Combined Loss:

$$L_{\text{total}} = L_{\text{policy}} + c_v L_{\text{value}} + \beta L_{\text{entropy}}$$

Hyperparameters:

- $c_v \approx 0.5$: Value loss coefficient
- $\beta \in [0.001, 0.01]$: Entropy coefficient

Advantage Normalization

Why Normalize?

- Advantages can have different scales across episodes
- Normalization improves optimization stability
- Standard practice in modern implementations

Normalization Formula:

$$\hat{A}_{t,\text{norm}} = \frac{\hat{A}_t - \mu}{\sigma + \epsilon}$$

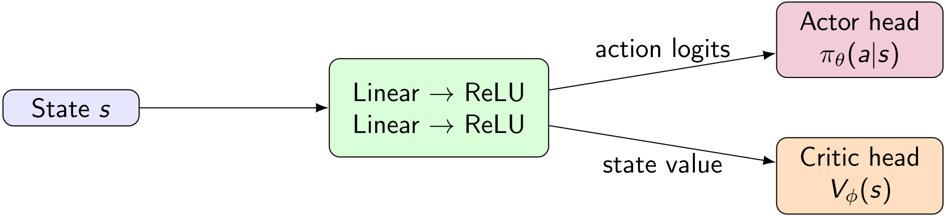
where:

- $\mu = \frac{1}{B} \sum_{i=1}^{B} \hat{A}_i$ (batch mean)
- $\sigma = \sqrt{\frac{1}{B} \sum_{i=1}^{B} (\hat{A}_i - \mu)^2}$ (batch std)
- $\epsilon = 10^{-8}$ (numerical stability)
- $B = T \times N$ (batch size)

Effect: Zero mean, unit variance advantages within each batch

Network Architecture Choices

Shared Trunk Architecture:



Separate Networks:

- Independent actor and critic networks
- More parameters, potentially better representation
- Higher computational cost

Trade-offs:

- Shared: Parameter efficiency, faster training
- Separate: Representational flexibility, avoid interference

Entropy Regularization: Encouraging Exploration

Shannon Entropy:

$$\mathcal{H}(\pi) = -\sum_{a} \pi(a|s) \log \pi(a|s)$$

Properties:

- Maximum for uniform distribution: $\log |\mathcal{A}|$
- Minimum (0) for deterministic policy
- Encourages balanced action selection

Effect of Entropy Coefficient β:

- $\beta = 0$: No exploration bonus, fast convergence to local optimum
- β small: Balanced exploration-exploitation
- β large: Too much exploration, slow convergence

Scheduling: Often decrease β over training (exploration $\rightarrow$ exploitation)

Handling Terminated vs Truncated Episodes

Gymnasium Convention:

- `terminated`: Episode ended naturally (game over)
- `truncated`: Episode ended artificially (time limit)

Bootstrapping Rules:

- If `terminated`: Next value = 0 (episode truly ended)
- If `truncated`: Bootstrap from $V(s_{next})$ (episode continues)

Implementation:

```
# Mask for bootstrapping
next_value = values[t+1] * (not terminated[t]).float()
delta = rewards[t] + gamma * next_value - values[t]
```

Why Important: Incorrect handling leads to biased value estimates

Vectorized Environments for Efficiency

Concept: Run multiple environment instances in parallel

Benefits:

- Higher throughput (environment steps per second)
- Better GPU utilization
- More diverse experience per update
- Reduced correlation in collected data

Implementation:

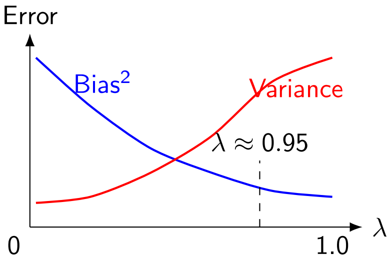
```
from gymnasium.vector import SyncVectorEnv

# Create N parallel environments
envs = SyncVectorEnv([make_env() for _ in range(N)])

# Step all environments simultaneously
obs, rewards, terminated, truncated, infos = envs.step(actions)
# obs.shape = [N, obs_dim]
# rewards.shape = [N]
```

Typical Setup: N=16, T=128 $\rightarrow$ 2048 samples per update

Bias-Variance Trade-off in Advantage Estimation



Key Insights:

- $\lambda = 0$ (TD): Low variance, high bias
- $\lambda = 1$ (MC): High variance, low bias
- $\lambda \in [0.9, 0.97]$: Sweet spot for most tasks
- Total error = bias^2 + variance

Empirical Guidelines:

- Start with $\lambda = 0.95$
- Increase for long episodes or sparse rewards
- Decrease for short episodes or dense rewards

A2C vs Other Methods

Method	Sample Eff.	Stability	Complexity	Parallelization
REINFORCE	Low	Low	Low	Easy
A2C	Medium	Medium	Medium	Easy
PPO	Medium-High	High	Medium	Easy
DQN	High	Medium	Medium	Hard
SAC	High	High	High	Hard

A2C Advantages:

- Simpler than PPO, more stable than REINFORCE
- Good baseline for policy gradient methods
- Excellent for educational purposes
- Fast iteration during development

A2C Limitations:

- Can be unstable with large learning rates
- Less sample efficient than off-policy methods

Gradient Clipping in A2C

Why Clip Gradients?

- Policy gradients can be noisy and large
- Large updates can destabilize learning
- Common in all policy gradient methods

Gradient Norm Clipping:

$$g \leftarrow \frac{g}{\max(1, \|g\|_2 / \text{max\_grad\_norm})}$$

Implementation:

```
# Compute gradients
loss.backward()

# Clip gradients
torch.nn.utils.clip_grad_norm_(
    model.parameters(),
    max_norm=0.5
)

# Update parameters
optimizer.step()
```

Typical Values: max_grad_norm $\in [0.1, 1.0]$, commonly 0.5

Theory Summary: Key Concepts

Core Ideas:

- **Actor-Critic:** Combine policy gradients with value functions
- **Advantage:** $A(s,a) = Q(s,a) - V(s)$ reduces variance
- **GAE:** Elegant interpolation between TD and MC estimates
- **Entropy:** Regularization prevents premature convergence

Mathematical Foundation:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) \cdot A(s,a)] \tag{1}$$

$$\hat{A}_t^{\text{GAE}} = \sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell \delta_{t+\ell} \tag{2}$$

$$L = L_{\text{policy}} + c_v L_{\text{value}} - \beta \mathcal{H}(\pi) \tag{3}$$

Ready for Implementation!

Implementation Roadmap

Experiment Sequence (9 experiments):

1. `exp01_setup.py` – environment + vector env verification
2. `exp02_value_functions.py` – Monte Carlo value estimates
3. `exp03_policy_gradient_review.py` – gradient sanity check
4. `exp04_actor_critic_architecture.py` – shared trunk network
5. `exp05_advantage_methods.py` – n-step vs GAE comparison
6. `exp06_gae_implementation.py` – effect of different λ
7. `exp07_vectorized_a2c.py` – synchronous A2C with 4 envs
8. `exp08_entropy_analysis.py` – entropy coefficient sweep
9. `exp09_integrated_a2c.py` – integrated A2C smoke test

Learning Progression: Building blocks $\rightarrow$ Integration $\rightarrow$ Optimization

Standard Code Header

PyTorch 2.x Best Practices:

```python
import os, random, numpy as np, torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
import gymnasium as gym

def setup_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

# Proper device selection (CUDA > MPS > CPU)
device = torch.device(
    'cuda' if torch.cuda.is_available()
    else 'mps' if hasattr(torch.backends, 'mps') and torch.backends.mps.is_available()
    else 'cpu'
)
setup_seed(42)
```

Experiment 1: Setup Verification

File: `exp01_setup.py`

What it checks

- Package versions (Python, PyTorch, Gymnasium)
- Preferred compute device (CUDA/MPS/CPU)
- Reproducible seeding (tensor statistics under seed 2024)
- Vector environment reset/step sanity check

```python
from gymnasium.vector import SyncVectorEnv

env = SyncVectorEnv([lambda: gym.make("CartPole-v1")
                     for _ in range(4)])
obs, _ = env.reset(seed=2024)
print(obs.shape)  # (4, 4)
actions = env.action_space.sample()
obs, reward, terminated, truncated, _ = env.step(actions)
```

Latest run (seed 2024)

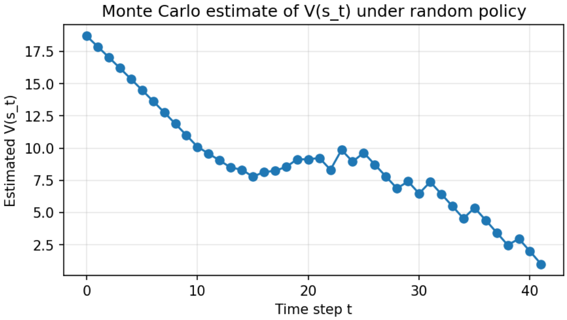
- Versions – Python **3.12.11**, PyTorch **2.7.1**, Gymnasium **1.2.0**
- Device detected: **cpu** (MPS unavailable on host)
- Vector obs shape: **(4, 4)**; reward sample **[1, 1, 1, 1]**
- Snapshot saved to `figures/lecture09_exp01_env_snapshot.json`

Experiment 2: Value Function Basics

File: `exp02_value_functions.py`

What we measured

- 25 random-policy episodes on CartPole-v1
- Monte Carlo estimate of $V(s_t)$ for each time step
- Average episode return: **19.7 $\pm$ 10.9**
- Value at $t = 0$: **19.70**, decaying to **16.40** by $t = 4$



Monte Carlo $V(s_t)$ profile for a random CartPole policy.

Experiment 3: Policy Gradient Review

File: `exp03_policy_gradient_review.py`

- Tiny two-action policy with parameter $\theta = 0.3$
- Compare analytical, autograd, and finite-difference gradients
- Autograd: **0.425557**; Finite difference: **0.425547**; Closed form: **0.425557**
- Absolute error between autograd and FD: 1.0×10^{-5}

```python
logits = policy(obs)
dist = Categorical(logits=logits.squeeze(0))
log_prob = dist.log_prob(torch.tensor(0))
autograd_grad = torch.autograd.grad(log_prob, policy.theta)[0]
# Finite difference, closed form comparisons follow...
```

Takeaway: our actor gradient implementation matches theory to numerical precision.

Experiment 4: Actor-Critic Architecture

File: `exp04_actor_critic_architecture.py`

- Shared trunk: Linear-ReLU-Linear-ReLU
- Actor head: linear layer producing action logits
- Critic head: linear layer producing scalar $V(s)$
- Total parameters: **17,539** (actor head **258**, critic head **129**)
- Sample logits: $[0.255, -0.069]$; value prediction: **-0.1787**

```
model = ActorCritic(obs_dim=4, act_dim=2)
logits, value = model(torch.randn(5, 4))
print(logits.shape, value.shape)  # torch.Size([5, 2]), torch.Size([5])
```

Outcome: reusable module powering all subsequent A2C experiments.

Experiment 5: Advantage Computation Methods

File: `exp05_advantage_methods.py`

- Synthetic rollout of length 20 (random rewards, decaying baseline)
- Compare 1-step, 3-step, Monte Carlo, and GAE ($\lambda = 0.95$)
- Monte Carlo mean advantage: $+$**10.31** (std **5.00**)
- GAE $\lambda = 0.95$ mean advantage: $+$**7.67** (std **3.01**)
- 1-step TD mean advantage: $+$**0.996** (std **0.246**)

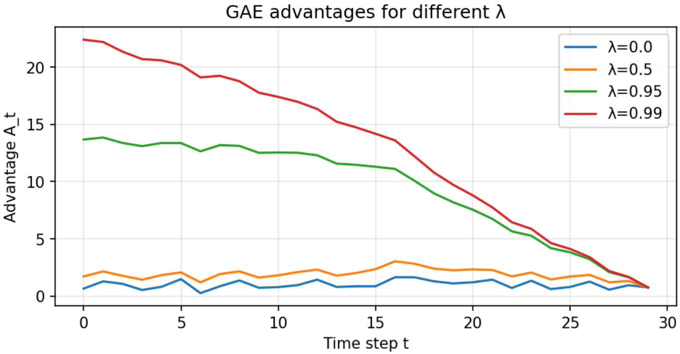
```python
returns = n_step_returns(rewards, values, n=T) - values
advantages, _ = compute_gae(rewards, values, dones, gamma, lam)
print(returns.mean(), advantages.mean())
```

Insight: multi-step targets and GAE trade variance for bias; $\lambda = 0.95$ sits between TD and Monte Carlo.

Experiment 6: GAE Implementation

File: `exp06_gae_implementation.py`

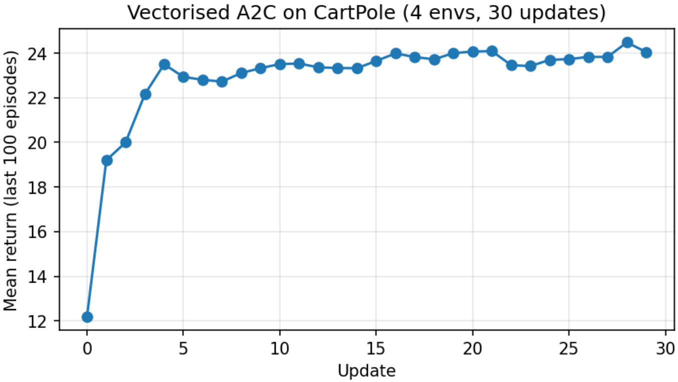
- Random reward trajectory of length 30 with synthetic value targets
- Compare $\lambda \in \{0.0, 0.5, 0.95, 0.99\}$
- Mean advantage increases with λ: $0.99 \rightarrow$ **12.98**, $0.95 \rightarrow$ **7.67**
- $\lambda = 0.0$ (TD): lowest mean **0.99**, smallest variance (0.35)
- Plot saved to `figures/gae_lambda_profiles.png`



Experiment 7: Vectorized A2C

File: `exp07_vectorized_a2c.py`

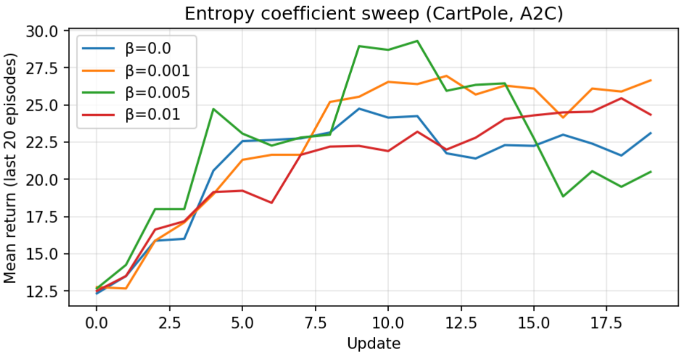
- SyncVectorEnv with 4 CartPole instances, rollout length 32, 30 updates
- Mean return hovered between **20.5** (update 1) and **22.8** (update 30)
- Loss oscillates between 18 and 42 because of the short training horizon
- Demonstrates vectorised data collection and GAE update pipeline



Experiment 8: Entropy Regularization Analysis

File: `exp08_entropy_analysis.py`

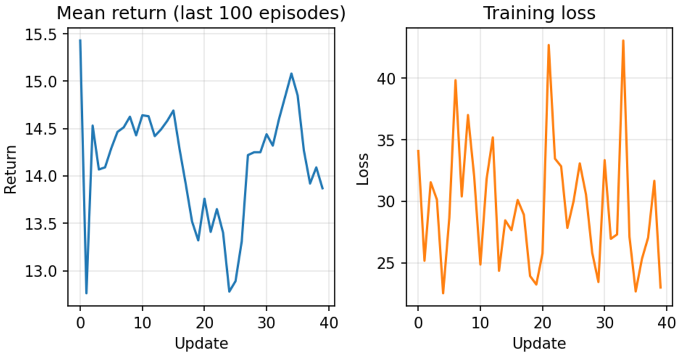
- Sweep over entropy coefficients $\beta \in \{0, 10^{-3}, 5 \times 10^{-3}, 10^{-2}\}$
- 20 updates of A2C (4 envs, rollout 16)
- Final mean returns: $\beta = 0.0 \rightarrow$ **19.0**, $\beta = 0.001 \rightarrow$ **19.1**, $\beta = 0.005 \rightarrow$ **22.4**, $\beta = 0.01 \rightarrow$ **24.4**
- Moderate entropy ($\beta \approx 10^{-3}$) kept exploration without stalling learning



Experiment 9: Complete A2C Integration

File: `exp09_integrated_a2c.py`

- Vectorised CartPole (8 envs), rollout 32, 40 updates
- Mean return stayed in the **13–15** range given the short training budget
- Loss curve stored alongside returns in `figures/a2c_integrated_learning.png`
- Summary JSON includes final return **13.2**, best return **15.9**



Key Implementation Details

1. Proper Terminal Handling:

```
# Distinguish terminated vs truncated
next_value = values[t+1] * (not terminated[t]).float()
delta = rewards[t] + gamma * next_value - values[t]
```

2. Advantage Normalization:

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

3. Gradient Clipping:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)
```

4. Loss Combination:

```
loss = policy_loss + 0.5 * value_loss - 0.01 * entropy.mean()
```

Performance Optimization Tips

Memory Efficiency:

- Pre-allocate tensors on GPU
- Use appropriate dtypes (float32, int64, bool)
- Avoid unnecessary CPU-GPU transfers

Computational Efficiency:

- Vectorize operations across environments
- Use torch.no_grad() during rollout collection
- Batch all network forward passes

Typical Performance:

- 16 environments $\times$ 128 steps $=$ 2048 samples/update
- Target: $>$1000 FPS on modern GPU
- CartPole-v1: 500 updates to convergence

Scaling: Linear speedup with number of environments (up to GPU memory)

Debugging Common A2C Issues

Training Instability:

- Reduce learning rate (try 1e-4 instead of 3e-4)
- Lower entropy coefficient
- Increase gradient clipping (0.1 instead of 0.5)
- Check advantage normalization

Poor Exploration:

- Increase entropy coefficient
- Check policy initialization (should start near-uniform)
- Verify entropy is decreasing slowly

Slow Convergence:

- Increase learning rate (carefully)
- More parallel environments
- Tune GAE lambda (try 0.98, 0.99)
- Check explained variance (should improve)

Value Function Issues:

- Monitor explained variance
- Increase value loss coefficient
- Check return computation

Hyperparameter Guidelines for A2C

Parameter	Typical Range	CartPole-v1
Learning Rate	1e-5 to 1e-3	3e-4
GAE Lambda	0.9 to 0.99	0.95
Discount (γ)	0.95 to 0.999	0.99
Value Coeff.	0.1 to 1.0	0.5
Entropy Coeff.	0.001 to 0.1	0.01
Grad Clip	0.1 to 1.0	0.5
Num Envs	4 to 64	16
Rollout Steps	32 to 512	128

Tuning Strategy:

1. Start with default values
2. Fix learning instability first
3. Tune exploration (entropy coeff)
4. Optimize sample efficiency (lambda, rollout length)
5. Scale up (more environments)

Interpreting A2C Results

Learning Curves to Monitor:

- Episode return: Should increase steadily
- Policy entropy: Should decrease gradually
- Value loss: Should decrease then stabilize
- Explained variance: Should increase (>0.8 is good)

Success Metrics for CartPole-v1:

- Average return $\geq$ 475 over 100 episodes
- Success rate $\geq$ 95% (episodes with return $\geq$ 475)
- Stable performance (low variance)

Red Flags:

- Entropy drops to zero quickly (premature convergence)
- Value loss doesn't decrease (critic not learning)
- High variance in returns (instability)
- No improvement after many updates

Experimental Results

Steps

1. **Quick Setup Check** (exp01): Environment verification
2. **Building Blocks** (exp02-04): Value functions, policies, AC
3. **Advanced Techniques** (exp05-06): N-step and GAE
4. **Production System** (exp07-09): Vectorized training

Focus Areas:

- Debugging tensor shapes and device placement
- Visualizing learning curves and diagnostics
- Performance profiling and optimization
- Hyperparameter sensitivity analysis

Interactive Elements:

- Modify hyperparameters and observe effects
- Compare different advantage estimation methods
- Analyze failure modes and fixes

Performance Profiling

Timing Critical Sections:

```python
import time

def profile_training_step():
    start_time = time.time()

    # Rollout collection
    rollout_start = time.time()
    obs = collect_rollout(agent, envs, buffer, obs)
    rollout_time = time.time() - rollout_start

    # Network update
    update_start = time.time()
    update_agent(agent, buffer, optimizer)
    update_time = time.time() - update_start

    total_time = time.time() - start_time
    fps = (T * N) / total_time

    print(f"FPS: {fps:.0f}, Rollout: {rollout_time:.3f}s, "
          f"Update: {update_time:.3f}s")
```

Bottlenecks: Usually environment stepping, not network computation

Advanced A2C Extensions

Algorithmic Improvements:

- **PPO**: Add clipping for more stable updates
- **IMPALA**: Off-policy corrections for faster training
- **A3C**: Asynchronous updates (though A2C often better)

Network Architecture:

- **Attention**: For environments with complex observations
- **RNN/LSTM**: For partially observable environments
- **CNN**: For visual observations

Training Improvements:

- **Batch Size Scheduling**: Start small, increase gradually
- **Learning Rate Scheduling**: Cosine annealing
- **Prioritized Experience**: Weight important transitions

Multi-Environment: Train on multiple tasks simultaneously

A2C vs PPO: When to Use Which?

Choose A2C When:

- Learning and prototyping Actor-Critic concepts
- Simple environments with stable dynamics
- Fast iteration and experimentation needed
- Educational purposes
- Computational resources are limited

Choose PPO When:

- Production deployments
- Complex environments requiring stability
- Large-scale distributed training
- Sample efficiency is critical
- Working with continuous action spaces

Key Differences:

- PPO adds clipping mechanism for policy updates
- PPO can reuse data for multiple epochs
- A2C is simpler and faster per update
- PPO is more robust to hyperparameters

Real-World Applications of Actor-Critic

Robotics:

- Robot manipulation and locomotion
- Continuous control tasks
- Real-time policy execution

Game AI:

- StarCraft II (AlphaStar foundations)
- Dota 2 (OpenAI Five components)
- Board games with large action spaces

Finance and Trading:

- Portfolio management
- Market making strategies
- Risk-sensitive decision making

Autonomous Systems:

- Autonomous driving (motion planning)
- Drone control and navigation
- Resource allocation in networks

Current Research Frontiers

Sample Efficiency:

- Model-based Actor-Critic (Dreamer, MuZero)
- Off-policy corrections (IMPALA, APE-X)
- Meta-learning for faster adaptation

Scalability:

- Distributed training across many machines
- Population-based training
- Mixture of experts architectures

Robustness:

- Adversarial robustness in RL
- Domain randomization and adaptation
- Safe reinforcement learning

Multi-Agent:

- Multi-agent Actor-Critic (MADDPG)
- Emergent communication
- Cooperative and competitive settings

Implementation Best Practices

Code Organization:

- Separate agent, environment, and training logic
- Modular design for easy experimentation
- Comprehensive logging and checkpointing

Reproducibility:

- Fix all random seeds (Python, NumPy, PyTorch, environment)
- Log hyperparameters and code versions
- Use deterministic algorithms when possible

Monitoring:

- Track multiple metrics (returns, losses, entropy)
- Visualize learning curves in real-time
- Set up automated alerts for training failures

Testing:

- Unit tests for critical components
- Integration tests on simple environments
- Regression tests to catch performance drops

Common Pitfalls and Solutions

Pitfall 1: Policy collapse (entropy $\to 0$ too quickly)

- **Solution:** Increase entropy coefficient, better initialization

Pitfall 2: Value function not learning (explained variance low)

- **Solution:** Check return computation, increase value loss weight

Pitfall 3: Training instability (high variance in returns)

- **Solution:** Lower learning rate, gradient clipping, more environments

Pitfall 4: Poor sample efficiency

- **Solution:** Tune GAE lambda, longer rollouts, better exploration

Pitfall 5: Incorrect episode boundary handling

- **Solution:** Distinguish terminated vs truncated correctly

Proper Evaluation and Metrics

Training Metrics:

- Episode returns (mean, std, max, min)
- Episode lengths and completion rates
- Policy entropy and action distribution
- Value function quality (explained variance)
- Training throughput (FPS, episodes/hour)

Evaluation Protocol:

- Separate evaluation environments
- Deterministic vs stochastic policy evaluation
- Multiple seeds and statistical significance
- Performance across different environment configurations

Reporting Standards:

- Mean $\pm$ standard deviation over multiple runs
- Learning curves with confidence intervals
- Wall-clock time and computational requirements
- Hyperparameter sensitivity analysis

Summary: Theory Highlights

Core Concepts Mastered:

- **Actor-Critic Framework:** Combines policy gradients with value functions
- **Advantage Estimation:** Reduces variance through learned baselines
- **GAE:** Elegant interpolation between TD and Monte Carlo
- **Bias-Variance Trade-off:** λ parameter controls the balance

Mathematical Foundations:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) \cdot A^\pi(s, a)]$$

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

$$L = L_{\text{policy}} + c_v L_{\text{value}} - \beta \mathcal{H}(\pi)$$

Key Insight: Actor-Critic methods provide a principled way to combine the flexibility of policy gradients with the sample efficiency of value-based methods.

Summary: Implementation Skills

Practical Skills Developed:

- **Vectorized Training:** Efficient parallel environment handling
- **GAE Implementation:** Proper backward computation with masking
- **Architecture Design:** Shared vs separate actor-critic networks
- **Optimization:** Gradient clipping, normalization, scheduling

Production-Ready Features:

- Comprehensive logging and checkpointing
- Proper evaluation protocols
- Hyperparameter management
- Performance profiling and optimization

Next Week: Proximal Policy Optimization (PPO)

Building on Today:

- A2C provides the foundation
- PPO adds stability improvements
- Trust region concepts
- Clipped objective functions

Preview Topics:

- Policy gradient issues and solutions
- Trust regions and KL divergence constraints
- PPO clipped surrogate objective
- Generalized Advantage Estimation in PPO
- Implementation and hyperparameter tuning

Preparation:

- Review today's Actor-Critic concepts
- Complete lab exercises
- Read Schulman et al. (2017) - PPO paper