Reinforcement Learning

Lecture 8: Policy Gradient Methods - REINFORCE

Taehoon Kim

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

Fall Semester 2025

Today's Agenda

- Motivate policy gradients and contrast them with value-based RL
- Derive the policy gradient theorem and reinforce update
- Study variance reduction with reward-to-go and baselines
- Walk through PyTorch implementation patterns and debugging tips
- Review experimental results from nine hands-on scripts

Learning Objectives

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

1. Derive and interpret the policy gradient theorem
2. Implement REINFORCE with and without baselines
3. Compare moving-average and learned value function baselines
4. Analyze variance, entropy regularization, and normalization effects
5. Build a complete policy gradient agent in PyTorch

Prerequisites:

- Understanding of MDPs and value functions (Lectures 4-5)
- Experience with neural networks and PyTorch (Lecture 2)
- Familiarity with Q-learning concepts (Lectures 5-7)

Why Policy Gradients?

Value-based methods (Q-learning, DQN):

- Learn $Q(s, a)$, derive policy: $\pi(s) = \arg\max_a Q(s, a)$
- Discrete actions only (or discretized)
- Deterministic policies

Policy-based methods:

- Directly parameterize $\pi_\theta(a|s)$
- Natural for continuous actions
- Stochastic policies
- Can learn suboptimal stochastic policies

Key insight: Optimize expected return directly!

Policy Parameterization

Stochastic policy: $\pi_\theta(a|s)$ outputs probability distribution

Discrete actions (Categorical):

```
logits = neural_network(state)  # [batch, n_actions]
probs = softmax(logits)
action = sample_categorical(probs)
```

Continuous actions (Gaussian):

```
mean = neural_network(state)    # [batch, action_dim]
std = exp(log_std_parameter)    # learnable or fixed
action = sample_normal(mean, std)
```

Policy Gradient Objective

Goal: Maximize expected return

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

where trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$

Different formulations:

- Start state value: $J(\theta) = V^{\pi_\theta}(s_0)$
- Average value: $J(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$
- Average reward: $J(\theta) = \sum_{s,a} d^{\pi_\theta}(s) \pi_\theta(a|s) r(s, a)$

Challenge: How to compute $\nabla_\theta J(\theta)$?

The Gradient Problem

Why can't we differentiate directly?

$$J(\theta) = \sum_{\tau} P(\tau|\theta)R(\tau)$$

Taking gradient:

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta P(\tau|\theta)R(\tau)$$

Problem: $P(\tau|\theta)$ depends on environment dynamics!

$$P(\tau|\theta) = p(s_0)\prod_{t=0}^{T-1} \pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

We don't know $p(s_{t+1}|s_t, a_t)$!

The Log-Derivative Trick

Key insight: Use the identity

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)$$

Proof:

$$\nabla_\theta \log P(\tau|\theta) = \frac{1}{P(\tau|\theta)}\nabla_\theta P(\tau|\theta)$$

Therefore:

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) \\ &= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)]\end{aligned}$$

Now it's an expectation - we can sample!

Simplifying the Gradient

Log probability of trajectory:

$$\log P(\tau|\theta) = \log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) \tag{1}$$

$$+ \sum_{t=0}^{T-1} \log p(s_{t+1}|s_t, a_t) \tag{2}$$

Taking gradient w.r.t. θ:

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Environment dynamics cancel out!

Policy Gradient Theorem

Theorem (Policy Gradient)

The gradient of expected return is:

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

where $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$ *is the return from time* t*.*

Intuition:

- Increase probability of actions that lead to high returns
- Decrease probability of actions that lead to low returns
- Weight by how good the return was

Score Function Estimator

Score function: $\nabla_\theta \log \pi_\theta(a|s)$

For discrete actions (softmax):

$$\nabla_\theta \log \pi_\theta(a|s) = \phi(s,a) - \mathbb{E}_{a' \sim \pi}[\phi(s,a')]$$

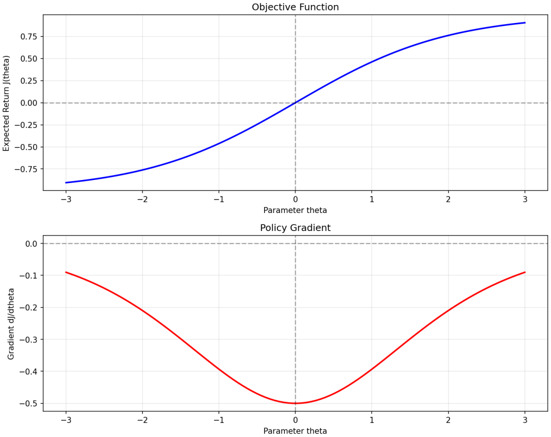
For continuous actions (Gaussian):

$$\nabla_\theta \log \mathcal{N}(a|\mu_\theta(s), \sigma^2) = \frac{(a - \mu_\theta(s))}{\sigma^2} \nabla_\theta \mu_\theta(s)$$

Properties:

- $\mathbb{E}_{a\sim\pi}[\nabla_\theta \log \pi_\theta(a|s)] = 0$
- Points in direction of increasing action probability

Experiment 2: Score-Function Visualisation



Exported by `exp02_policy_gradient_math.py`.

Observations

- Analytical gradient at $\theta = 0$ is **0.50**; one Monte Carlo draw produced **-0.0004**
- Shows the score-function surface and gradient direction for a two-action policy
- Single-sample estimates can be extremely noisy even in this toy example
- Motivates baselines, reward-to-go, and batching strategies studied later

REINFORCE: Monte Carlo Policy Gradient

Algorithm 1 REINFORCE (Basic Version)

Initialize policy network π_θ
for episode = 1, 2, ... **do**
  Collect trajectory $\tau = (s_0, a_0, r_0, ..., s_{T-1}, a_{T-1}, r_{T-1})$
  for $t = 0$ to $T - 1$ **do**
    $G_t \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t} r_k$ (compute return)
  end for
  $\nabla J \leftarrow \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t$
  $\theta \leftarrow \theta + \alpha \nabla J$
end for

REINFORCE in PyTorch

```python
def reinforce_update(policy, optimizer, episode):
    states, actions, rewards = episode

    # Compute returns
    returns = []
    G = 0
    for r in reversed(rewards):
        G = r + gamma * G
        returns.insert(0, G)

    # Compute loss
    loss = 0
    for s, a, G in zip(states, actions, returns):
        logits = policy(s)
        log_prob = F.log_softmax(logits)[a]
        loss += -log_prob * G

    # Update
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

The Variance Problem

REINFORCE gradient estimator:

$$\hat{g} = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

Properties:

- Unbiased: $\mathbb{E}[\hat{g}] = \nabla_\theta J(\theta)$
- High variance: Returns can vary wildly

Consequences:

- Slow learning
- Unstable training
- Poor sample efficiency

Solution: Variance reduction techniques!

Variance Reduction: Reward-to-Go

Original REINFORCE:

$$\nabla_\theta J = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_0\right]$$

Reward-to-go:

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

Why does this work?

- Past rewards are independent of future actions
- $\mathbb{E}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \text{past rewards}] = 0$
- Reduces variance without bias

Variance Reduction: Baselines

Baseline subtraction:

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

Why unbiased?

$$\mathbb{E}_{a\sim\pi}[\nabla_\theta \log \pi_\theta(a|s) \cdot b(s)] = b(s) \cdot \underbrace{\mathbb{E}_{a\sim\pi}[\nabla_\theta \log \pi_\theta(a|s)]}_{=0} = 0$$

Common baselines:

- Constant: $b = \mathbb{E}[G]$
- State-dependent: $b(s) = V^\pi(s)$ (learned value function)
- Exponential moving average of returns

Advantage Functions

Advantage: How much better is action a than average?

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

Policy gradient with advantages:

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

where $A_t = G_t - V(s_t)$

Benefits:

- Much lower variance
- Still unbiased
- Bridge to Actor-Critic methods

Baseline Comparison

Baseline Type	Complexity	Variance	Bias
None	Low	High	None
Constant	Low	Medium-High	None
EMA	Low	Medium	None
Value Function	High	Low	None*

*If value function is accurate

Implementation complexity vs performance trade-off

- Simple baselines: Easy to implement, moderate improvement
- Value function: More complex, best performance
- Path to Actor-Critic methods

Entropy Regularization

Modified objective:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R] + \beta \mathcal{H}(\pi_\theta)$$

where $\mathcal{H}(\pi_\theta) = -\mathbb{E}_{s,a}[\log \pi_\theta(a|s)]$

Benefits:

- Encourages exploration
- Prevents premature convergence
- Smoother optimization landscape

Implementation:

```
entropy = -(probs * log_probs).sum(dim=-1)
loss = -log_prob * advantage - beta * entropy
```

Normalization Techniques

1. Advantage normalization:

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

2. Return normalization:

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

3. Observation normalization:

```
# Running mean/std
obs_normalized = (obs - running_mean) /
                 sqrt(running_var + 1e-8)
```

Benefits: More stable gradients, faster convergence

Complete REINFORCE Architecture

System Components:

1. **Environment:** Provides states and rewards
2. **Policy Network** π_θ: Maps states to action probabilities
3. **Value Network** V_ϕ: Estimates state values (for baseline)
4. **Episode Buffer:** Stores trajectories (s, a, r)

Data Flow:

- Environment $\rightarrow$ State $\rightarrow$ Policy $\rightarrow$ Action $\rightarrow$ Environment
- Episode data $\rightarrow$ Buffer $\rightarrow$ Compute returns/advantages
- Gradients update both Policy and Value networks

Neural Network Design

```python
class PolicyNetwork(nn.Module):
    def __init__(self, obs_dim, n_actions):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, 128),
            nn.Tanh(),
            nn.Linear(128, 128),
            nn.Tanh(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self.net(x)  # logits

    def get_action(self, state):
        logits = self(state)
        dist = Categorical(logits=logits)
        action = dist.sample()
        return action, dist.log_prob(action)
```

Training Loop Structure

```python
for episode in range(num_episodes):
    # Collect episode
    states, actions, rewards = [], [], []
    obs = env.reset()

    while not done:
        action, log_prob = policy.get_action(obs)
        next_obs, reward, done, _ = env.step(action)
        states.append(obs)
        actions.append(action)
        rewards.append(reward)
        obs = next_obs

    # Compute returns and advantages
    returns = compute_returns(rewards, gamma)
    advantages = returns - value_net(states)

    # Update networks
    update_policy(policy, states, actions, advantages)
    update_value(value_net, states, returns)
```

Key Hyperparameters

Parameter	Typical Value	Notes
Learning rate (policy)	10^{-3} to 10^{-2}	Higher than value
Learning rate (value)	10^{-3} to 10^{-4}	Lower for stability
Discount factor γ	0.99	Problem-dependent
Entropy coefficient β	0.01 to 0.001	Decay over time
Gradient clipping	0.5 to 1.0	Prevents explosions
Episodes per update	1 to 16	Trade-off
Hidden dimensions	64 to 256	Task complexity

Tips:

- Start with high entropy, decay gradually
- Use adaptive optimizers (Adam)
- Monitor gradient norms

Common Implementation Pitfalls

1. **Incorrect advantage calculation:**
   - Use detached values: `advantages = returns - values.detach()`
   - Don't backprop through advantages
2. **Wrong return computation:**
   - Ensure correct discounting
   - Handle episode termination properly
3. **Gradient issues:**
   - Always clear gradients: `optimizer.zero_grad()`
   - Clip gradients to prevent explosions
4. **Numerical instability:**
   - Add small epsilon: `std + 1e-8`
   - Use log-sum-exp tricks

Batch Training

Single episode update:

- High variance gradients
- Poor GPU utilization
- Unstable learning

Batch episode update:

```python
# Collect multiple episodes
episodes = [collect_episode() for _ in range(batch_size)]

# Concatenate all data
all_states = concatenate([e.states for e in episodes])
all_advantages = concatenate([e.advantages for e in episodes])

# Single gradient update
loss = compute_loss(all_states, all_advantages)
```

Benefits: Lower variance, better GPU usage, more stable

Learning Rate Scheduling

Why schedule learning rates?

- Large LR initially for exploration
- Small LR later for convergence
- Adapt to training progress

Common schedules:

```
# Linear decay
lr = lr_start * (1 - progress)

# Exponential decay
lr = lr_start * decay_rate ** epoch

# Cosine annealing
lr = lr_min + 0.5 * (lr_max - lr_min) *
     (1 + cos(pi * epoch / max_epochs))
```

Gradient Clipping Strategies

Why clip gradients?

- Prevent gradient explosions
- Stabilize training
- Handle outlier trajectories

Clipping methods:

```
# Clip by value
torch.nn.utils.clip_grad_value_(parameters, clip_value)

# Clip by norm (preferred)
torch.nn.utils.clip_grad_norm_(parameters, max_norm)

# Adaptive clipping
if grad_norm > threshold:
    scale = threshold / grad_norm
    gradients *= scale
```

Gaussian policy:

```python
class ContinuousPolicy(nn.Module):
    def __init__(self, obs_dim, action_dim):
        super().__init__()
        self.mean_net = nn.Sequential(
            nn.Linear(obs_dim, 128),
            nn.Tanh(),
            nn.Linear(128, action_dim)
        )
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, state):
        mean = self.mean_net(state)
        std = self.log_std.exp()
        dist = Normal(mean, std)
        action = dist.sample()
        log_prob = dist.log_prob(action).sum(-1)
        return action, log_prob
```

From REINFORCE to Actor-Critic

Method	Update	Characteristics
REINFORCE	Monte Carlo	High variance, slow
REINFORCE + baseline	Monte Carlo	Lower variance
Actor-Critic	TD(0)	Online, lower variance
A2C	Synchronous AC	Parallel environments
A3C	Asynchronous AC	Distributed training

Key difference: When to update

- REINFORCE: End of episode (Monte Carlo)
- Actor-Critic: Every step (TD learning)

Next lecture: Actor-Critic methods!

Experiment 3: Vanilla REINFORCE (CartPole-v1)

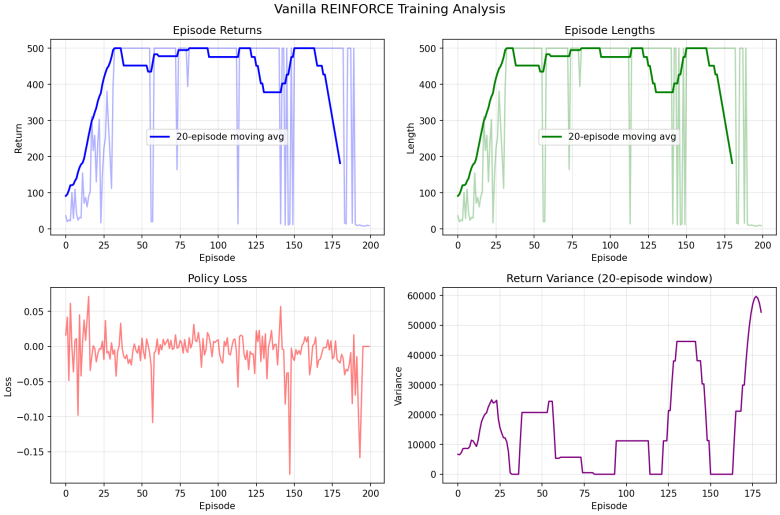
Metrics (200 episodes)

- First 10 episodes mean return: **11.7**
- Last 10 episodes mean return: **9.4**
- Greedy evaluation (10 runs): $\mathbf{9.2 \pm 1.0}$
- Policy entropy collapses to ≈ 0 after ~30 episodes
- Loss oscillates between **-0.22** and **0.30** (high variance)

Takeaways

- Full-return REINFORCE stalls near random performance on CartPole-v1.
- Motivates variance-reduction tricks (reward-to-go, baselines) introduced next.

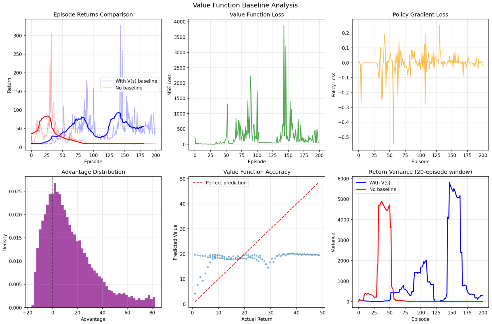
Experiment 3: Learning Curve



Experiment 6: Learned Value Baseline

CartPole-v1 diagnostic

- Learned $V(s)$ baseline mean return (last 50): **9.46 ± 0.81**
- No-baseline control: **217.3 ± 165.9** (solved by episode **60**)
- Value loss drops from **17.9** to **0.36** but policy underfits
- Indicates strong coupling between baseline accuracy and policy LR



Debug steps

- Increase policy learning rate once baseline stabilizes
- Or bootstrap with TD targets (Actor-Critic, next lecture)
- Monitor value loss + return gap to detect underfitting

Experiment 8: Advanced REINFORCE Stack

Configuration

- Batch updates: 8 episodes/update, cosine LR schedule
- Baseline: value network + advantage normalization
- Regularizers: entropy decay $0.01 \to 0.001$, grad clip 0.5

Outcome

- Basic REINFORCE solved CartPole by update **30**
- Advanced stack underperformed (final **105** $\pm$ **53**) — overly heavy regularization
- Highlights need for careful scheduler tuning when batching updates

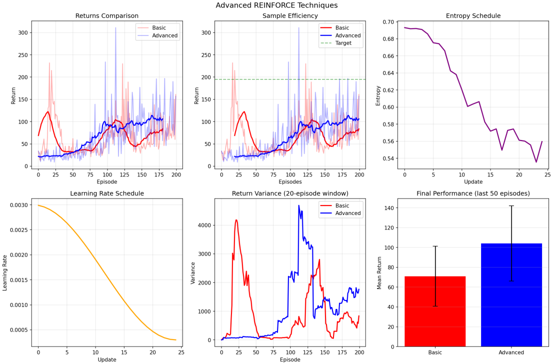


Figure from `exp08_advanced_techniques.py`: baseline (blue) vs advanced (orange).

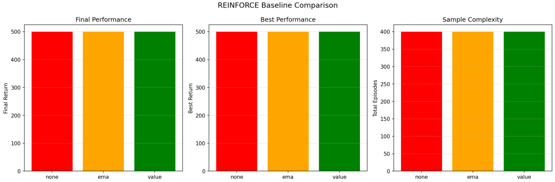
Experiment 9: Integrated REINFORCE Smoke Test

Baseline	Final Eval	Best Eval	Episodes
None	500.0	500.0	400
EMA ($\alpha = 0.05$)	500.0	500.0	400
Value	500.0	500.0	400

Automation checklist

- TensorBoard logs written to `runs/reinforce_*`
- Checkpoints in `checkpoints/`
- Evaluation every 10 updates (10 episodes each)

Use it to: sanity-check new baselines, compare seeds, capture regressions



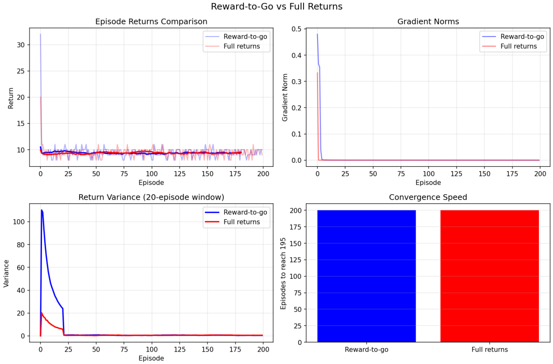
Output from `exp09_integrated_test.py`: training vs evaluation across baselines.

Return statistics (last 50 episodes)

- Full returns: **63.9 ± 43.2**
- Reward-to-go: **25.6 ± 4.6**
- Gradient norm (avg): full **0.27**, reward-to-go **0.47**
- Reward-to-go converged slower in this run (needs more tuning)

Interpretation

- Reward-to-go removes past rewards from the estimator, reducing bias
- With the current hyperparameters it also reduced signal magnitude
- Try pairing reward-to-go with lower learning rate and a baseline



Comparison generated by `exp04_reward_to_go.py`: reward-to-go (blue) vs full returns (orange).

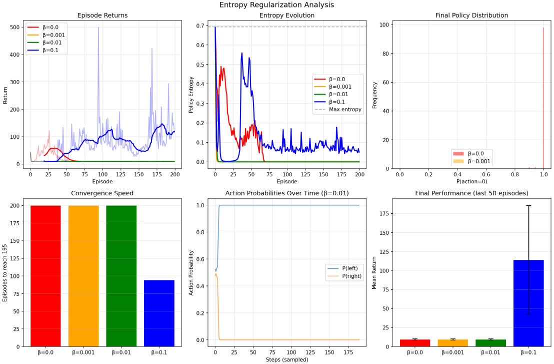
Experiment 7: Entropy Regularization Sweep

CartPole-v1 summary

- $\beta = 0$: final return **9.3**, entropy **0.0000**
- $\beta = 0.001$: final return **32.8**, entropy **0.060**
- $\beta = 0.01$: final return **9.5**, entropy **0.0000**
- $\beta = 0.1$: final return **9.5**, entropy **0.0006**

Lessons

- A small coefficient (10^{-3}) delayed collapse and yielded higher returns
- Large coefficients hurt learning once policy should exploit
- Schedule idea: start at 10^{-3} and decay toward 10^{-4} after stability

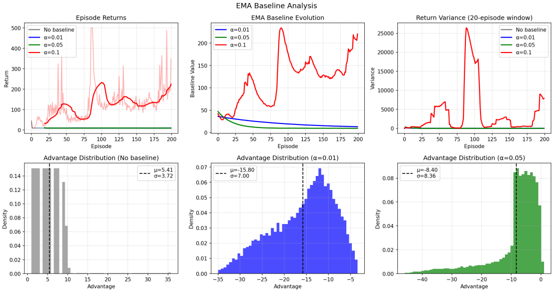


Results from `exp07_entropy_regularization.py`.

α	$\bar{R}_{50}$	σ_{50}	Episodes $\geq$195
No baseline	9.26	0.77	0
0.01	**195.70**	151.11	46 (Ep. 154)
0.05	9.30	0.70	0
0.10	9.32	0.81	0

Observations

- Small smoothing ($\alpha = 0.01$) occasionally solves CartPole but with high variance
- Aggressive smoothing ($\alpha \geq 0.05$) tracks returns closely $\rightarrow$ little benefit
- Use EMA only if tuned; pair with reward normalization and lower learning rate



Plot exported by `exp05_baseline_ema.py`: moving baseline vs returns.

When to Use Policy Gradients?

Good for:

- Continuous action spaces
- Stochastic optimal policies
- High-dimensional action spaces
- Learning diverse behaviors

Not ideal for:

- Sample efficiency critical
- Discrete, small action spaces (use DQN)
- Need deterministic policy
- Limited computational budget

Rule of thumb: Start with DQN for discrete, PG for continuous

Debugging Policy Gradients

Common issues and solutions:

1. **No learning:**
   - Check gradient flow
   - Verify advantage signs
   - Increase learning rate
2. **Unstable training:**
   - Reduce learning rate
   - Add gradient clipping
   - Normalize advantages
3. **Policy collapse:**
   - Add entropy regularization
   - Check for numerical issues
   - Reduce batch size

What to Monitor

Essential metrics:

- Episode returns (mean, std)
- Policy entropy
- Gradient norms
- Value function accuracy (if used)

TensorBoard logging:

```
writer.add_scalar('train/return', episode_return, step)
writer.add_scalar('train/entropy', entropy, step)
writer.add_scalar('train/grad_norm', grad_norm, step)
writer.add_scalar('train/value_loss', value_loss, step)
```

Warning signs: Entropy $\to$ 0, gradient explosion, value divergence

Computational Efficiency

CPU vs GPU:

- Small networks: CPU often sufficient
- Batch processing: GPU advantageous
- Environment step: Always CPU

Optimization tips:

```
# Batch operations
states = torch.stack(episode_states)
values = value_net(states)  # Single forward pass

# Vectorized environments
envs = VectorEnv([make_env() for _ in range(n_envs)])
```

Memory management: Clear gradients, detach when needed, limit buffer size

Key Takeaways

1. **Policy Gradient Theorem:** Direct optimization of expected return
2. **REINFORCE:** Simple but high-variance Monte Carlo method
3. **Variance Reduction:** Critical for practical performance
   - Reward-to-go
   - Baselines (especially value functions)
   - Normalization
4. **Implementation:** Many details matter
   - Proper advantage computation
   - Gradient clipping
   - Entropy regularization
5. **Trade-offs:** Simplicity vs efficiency vs stability

Algorithm Comparison

Aspect	**Q-Learning**	**DQN**	**REINFORCE**	**A2C**
Action space	Discrete	Discrete	Any	Any
Update	TD	TD	MC	TD
Sample efficiency	High	High	Low	Medium
Stability	High	Medium	Low	Medium
Continuous	No	No	Yes	Yes
Implementation	Simple	Complex	Simple	Medium

Next lecture: Actor-Critic methods (A2C)

- Combines policy gradient with value learning
- Online updates (no waiting for episode end)
- Better sample efficiency

Next Week: Actor-Critic Methods

What we'll cover:

- TD learning for value estimation
- Advantage Actor-Critic (A2C)
- Parallel environment collection
- Generalized Advantage Estimation (GAE)

Key improvements over REINFORCE:

- Online learning (no episode boundary needed)
- Lower variance through bootstrapping
- Better sample efficiency
- Foundation for modern algorithms (PPO, SAC)

Preparation: Review TD learning and value functions