Homework 5: Policy Gradients

u1541147 Sheng-Hung Tseng , u1527521 Kuan-Yu Chen

March 25, 2025

Part 1a: Understanding and Implementing a Basic Policy Gradient Algorithm

Training Results and Discussion

Pictures show as attached. Below are the learning curves obtained from training a basic policy gradient (REINFORCE) agent on `CartPole-v1` under different learning rates and epoch settings. Each figure plots the average return (i.e., total reward) per episode over the training epochs.

- **Is the agent learning?**
  Yes. We generally see an upward trend in returns over epochs, indicating that the policy improves with more training.
- **Performance over time:**
  The agent typically starts with poor performance (random actions yielding low returns) and gradually improves. Over enough epochs, it often approaches the maximum possible return (around 200 for CartPole).
- **Is it monotonically improving?**
  Not necessarily. We see fluctuations or zig-zag patterns in the curves. This is normal for policy gradient methods, which are highly sensitive to individual sampled trajectories.

Why do we see these behaviors?

- **On-policy variance:** Because the data is collected by following the current policy, and each update changes that policy, the data distribution can shift significantly. This leads to high variance in the estimated returns and gradients, causing temporary dips in performance even as the overall trend is upward.
- **Learning rate:**

  - *Higher learning rates* (e.g., 0.01) can yield faster initial improvements but may cause larger, more erratic parameter updates, leading to bigger performance fluctuations.
  - *Smaller learning rates* (e.g., 1e-5) are more conservative, changing the policy more slowly. This can produce more stable updates but also slows learning progress, so the agent may take many epochs to achieve high returns.

- **Reward ceiling in CartPole:** Once the agent starts balancing the pole well, it quickly saturates at or near 200 returns. Hence, learning improvements may flatten out near that limit.

- **Environmental randomness and stochastic actions:** Even minor changes in initial conditions or action selections can cause significant variations in episode length. Consequently, the observed returns can fluctuate from epoch to epoch.

Key Observations

- **Overall Upward Trend:** In most configurations, we see the average returns improve from near-random (around 20–30) to substantially higher values, demonstrating that the agent is learning.

- **Non-monotonic Behavior:** It is common to observe dips or sudden changes in returns from epoch to epoch. This is due to the stochastic nature of sampling and on-policy updates.

- **Learning Rate Trade-Off:**
  - Higher learning rates often bring quick initial gains but can overshoot and cause more variability.
  - Very small learning rates might eventually reach high performance but require more epochs to converge.

- **Why we think it behaves this way:**
  1. The policy is continuously being updated based on sampled trajectories, which introduces variance in the returns.
  2. CartPole quickly saturates near a maximum score of 200, so once the policy is good enough to balance, improvements flatten.
  3. Large gradients (from large LR) might push the policy too far, causing temporary drops in performance, while smaller LR updates are more gradual.

Part 1b: Understanding and Implementing a Basic Policy Gradient Algorithm

As the training progresses, you can qualitatively observe the policy shift from almost random, jerky movements (where the pole quickly tips over) to smoother, more deliberate actions that keep the pole upright for longer. Early in training, the cart may move abruptly back and forth with little control. After several epochs, the agent's actions become more purposeful, making smaller, more precise left/right adjustments to maintain balance.

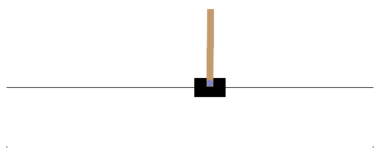


Figure 1: Sample screenshot showing the cart and pole remaining upright during a test rollout.

By observing the cart's behavior at each epoch, you can visually confirm that the agent's learned policy gradually refines its balancing strategy. The image above demonstrates that the pole no longer topples immediately, indicating both that visualization is functioning correctly and that the policy is indeed improving.

Part 2: Reducing Variance with Reward-to-Go

Figure 2 shows that the *Reward-to-Go (RTG)* method typically learns faster and attains higher returns than the *Vanilla* policy gradient approach. Although both methods eventually improve substantially over the random initial policy, RTG usually converges more rapidly and maintains better average performance.

- **Reason for the difference:**
  - In the **vanilla** version, *the same total episode return* is assigned to every timestep, leading to higher variance: even actions late in the episode can be "credited" for rewards that occurred early on.
  - In the **reward-to-go** version, each timestep is weighted by the *sum of future rewards* from that timestep onward, which focuses credit assignment more accurately. This typically reduces variance in the gradient estimates and speeds up learning.
- **Conclusion:** In many tasks, reward-to-go outperforms the vanilla approach, especially in terms of learning speed and more stable returns.

However, performance gains can vary by environment, hyperparameters, and random seed.

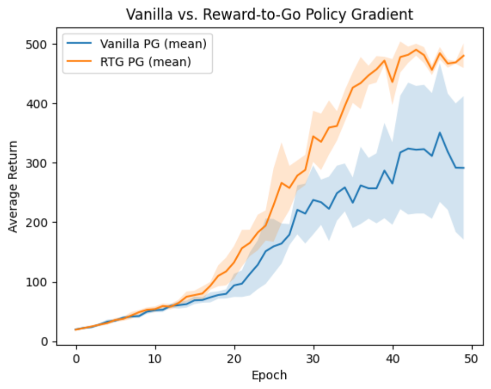


Figure 2: Learning curves comparing Vanilla PG vs. Reward-to-Go PG. The RTG variant converges faster and achieves higher returns on average, though both methods outperform random behavior.

Further Discussion

Based on the experimental results presented above, several important insights have emerged:

Insights

- **Training Dynamics:** The policy gradient methods, while generally effective, exhibit inherent variance due to on-policy sampling. This variance leads to non-monotonic performance improvements over epochs, as observed in the learning curves.
- **Hyperparameter Sensitivity:** The learning rate plays a crucial role in the agent's performance. A learning rate that is too high can cause erratic updates and instability, while a learning rate that is too low can slow down convergence significantly. The optimal learning rate appears to be a trade-off between stability and speed.
- **Reward-to-Go Advantage:** Incorporating the reward-to-go strategy reduces variance by more accurately assigning credit to individual actions, leading to faster and more stable convergence. This suggests that methods to further reduce variance (e.g., baselines or advantage estimators) could be beneficial.

Part 3: Continuous Actions

1 Introduction

In this report I describe the experiments I conducted using a simple REINFORCE (policy gradient) method on two Gymnasium environments:

- **CartPole-v1** (discrete action space)
- **Pendulum-v1** (continuous action space)

I experimented with different learning rates, numbers of training episodes, and activation functions (Tanh vs. ReLU) to see their effect on performance.

2 Environment Details

2.1 CartPole-v1

The CartPole-v1 environment requires balancing a pole on a cart by applying discrete actions. It is considered solved when the agent consistently reaches near 500 reward points.

2.2 Pendulum-v1

The Pendulum-v1 environment is a continuous control problem where the goal is to swing up a pendulum and keep it upright. The reward is generally negative, so improvements are measured by the reward moving closer to zero.

3 Hyperparameters Explored

I varied the following hyperparameters in my experiments:

- **Learning Rates (LR):** 0.01, 0.001, 0.0001.
- **Training Episodes (Epochs):** 100, 200, 1000.
- **Activation Functions:** Both Tanh (default) and ReLU.

Each combination was tested on both environments. In some cases, I observed that using a high LR (e.g., 0.01) can result in fast but unstable learning, while a low LR (e.g., 0.0001) leads to slow improvements.

4 Observations

4.1 CartPole-v1

- **High LR (0.01) + 1000 Episodes:** The agent learned quickly, occasionally reaching near 500 reward points, but the learning curve was quite noisy.
- **Moderate to Low LR (0.001 or 0.0001) + 1000 Episodes:** The training was smoother and more stable, although it took longer for the agent to consistently reach high rewards.
- **Shorter Training Runs (100–200 Episodes):** The agent often failed to converge, with reward values remaining low.

4.2 Pendulum-v1

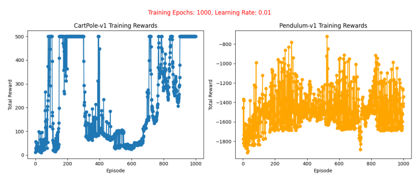
- In Pendulum-v1, the rewards are negative (starting around -1500 to -1000). Improvement is reflected by the reward moving upward (closer to zero).
- **LR = 0.01:** The updates were too drastic, causing high variance in the returns.
- **LR = 0.001 or 0.0001:** More stable improvement was observed, with returns gradually moving toward -600 or -400.
- Shorter training periods did not allow sufficient improvement due to the difficulty of continuous control with basic REINFORCE.

5 What I Learned

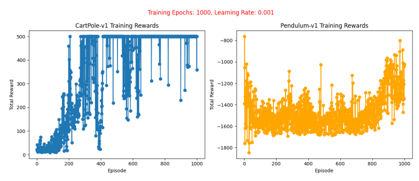
- **Learning Rate:** A moderate learning rate (around 0.001) provided a good balance between speed and stability. Too high a learning rate leads to unstable learning, while too low a learning rate requires a significantly larger number of episodes.
- **Number of Episodes:** Increasing the number of training episodes is crucial, especially for more complex environments like Pendulum-v1.
- **Activation Functions:** Both Tanh and ReLU can be used. Tanh is common in smaller environments like CartPole-v1, while ReLU sometimes speeds up learning but can be more sensitive to hyperparameter settings.
- **REINFORCE Limitations:** The simple REINFORCE algorithm works on CartPole-v1 but struggles with Pendulum-v1 without further improvements (such as using a baseline or actor-critic methods).

6 Figures

Below are the figures showing the training reward curves under different settings. Each figure includes plots for different learning rates and activation functions.

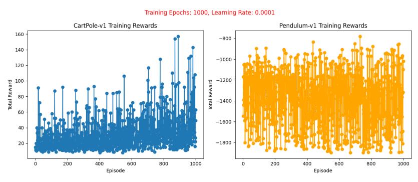


(a) LR = 0.01, 1000 Episodes, ReLU

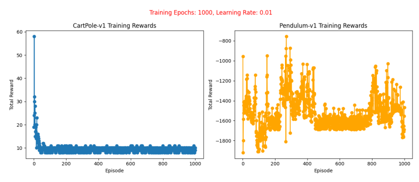


(b) LR = 0.001, 1000 Episodes, ReLU

Figure 3: Training curves with ReLU activation under different learning rates (CartPole/Pendulum settings).

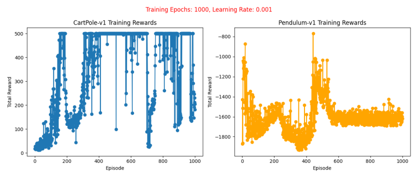


(a) LR = 0.0001, 1000 Episodes, ReLU

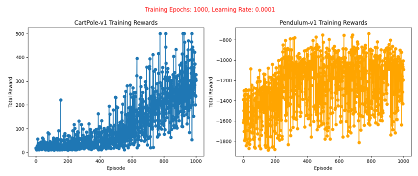


(b) LR = 0.01, 1000 Episodes, Tanh

Figure 4: Comparing activation functions: ReLU vs. Tanh at LR = 0.01 over 1000 episodes.

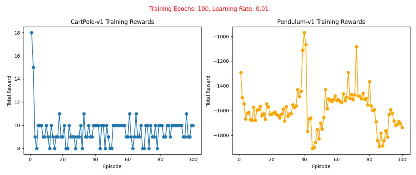


(a) LR = 0.001, 1000 Episodes, Tanh

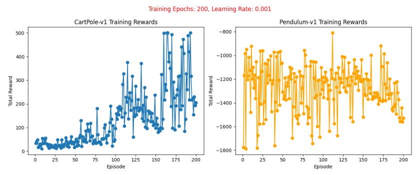


(b) LR = 0.0001, 1000 Episodes, Tanh

Figure 5: Training curves with Tanh activation under different learning rates (1000 episodes).



(a) LR = 0.01, 100 Episodes



(b) LR = 0.001, 200 Episodes

Figure 6: Short-run experiments showing the effect of episode count at different learning rates.

7 Conclusion

In summary, these experiments revealed that:

- A moderate learning rate (around 0.001) with enough episodes (500–1000) yielded the most stable performance.
- CartPole-v1 is relatively easy to solve with a basic REINFORCE algorithm, while Pendulum-v1 requires more careful tuning and possibly additional techniques.
- The choice of activation function (Tanh vs. ReLU) affects the learning dynamics, with each having its own advantages depending on the specific environment.

Extra Credit A: Baselines and Generalized Advantage Estimation

1. Performance on CartPole-v1

Training Setup

- **Policy LR:** 0.001

- **Value LR:** 0.001
- **Epochs:** 300
- **Activations:** Tanh or ReLU (two separate runs)

Performance on CartPole-v1

Policy LR: 0.001, Value LR: 0.001, Epochs: 300, Activation: ReLU

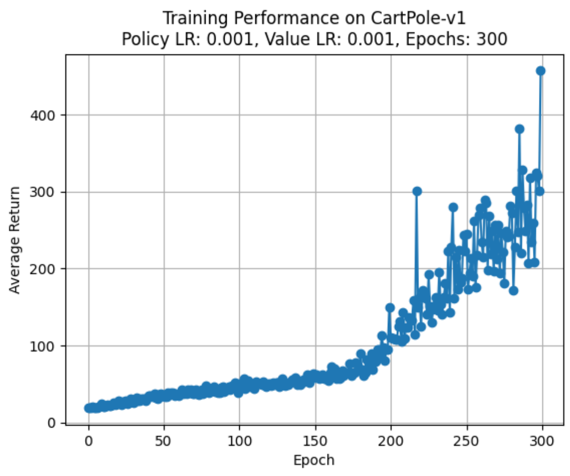


Figure 7: Training performance on CartPole-v1 with ReLU activation

Policy LR: 0.001, Value LR: 0.001, Epochs: 300, Activation: Tanh

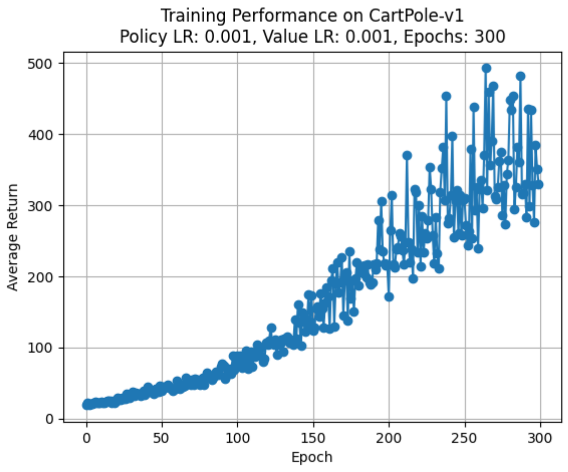


Figure 8: Training performance on CartPole-v1 with Tanh activation

Observations

1. **Overall Return Trend:** The agent begins with very low returns (since episodes end quickly) but steadily learns to balance the pole for longer periods. Around 100–150 epochs, returns exceed 100, and by epoch 300 they often approach or surpass 400.
2. **Effect of Activation Function:**
   - **Tanh** vs. **ReLU** both reach high returns.
   - Tanh often provides smoother gradients; ReLU can learn quickly at first but can be more sensitive to initialization.
3. **Stability and Variance:** With a learned baseline (value function + GAE), training variance is much lower compared to no baseline. Any performance dips are more quickly recovered due to reduced variance in the advantage estimates.

Comparison to No Baseline

- **Higher Variance:** Without a baseline, policy gradient updates depend only on raw returns (or reward-to-go), which are typically high-variance estimates.

- **Slower Convergence:** Learning is slower and less stable. CartPole can still be solved, but usually requires more epochs and experiences bigger swings in performance.

2. Performance on HalfCheetah-v5

Training Setup

- **Policy LR:** 0.003
- **Value LR:** 0.001
- **Epochs:** 300
- **Activations:** Tanh or ReLU (two separate runs)

Performance on HalfCheetah-v5

Policy LR: 0.003, Value LR: 0.001, Epochs: 300, Activation: ReLU

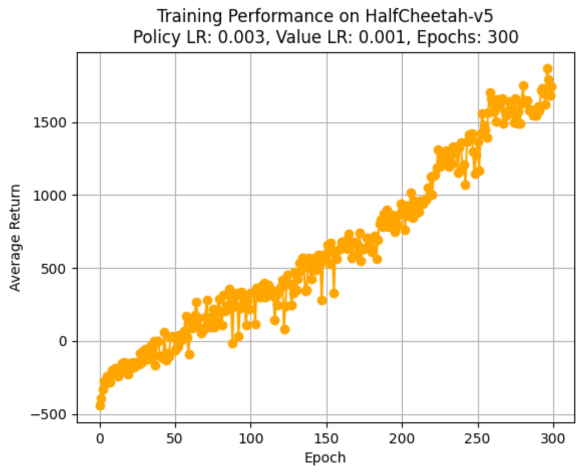


Figure 9: Training performance on HalfCheetah-v5 with ReLU activation

Policy LR: 0.003, Value LR: 0.001, Epochs: 300, Activation: Tanh

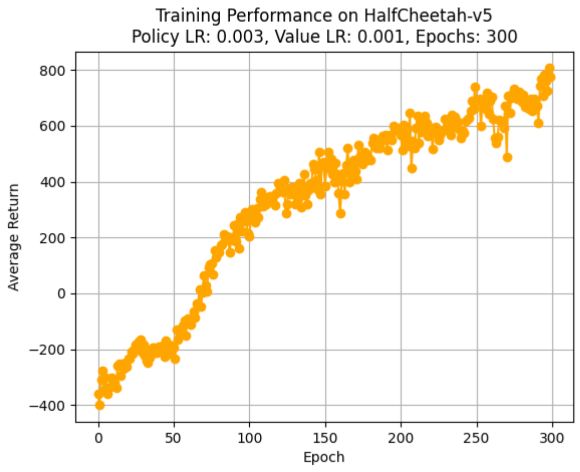


Figure 10: Training performance on HalfCheetah-v5 with Tanh activation

Observations

1. **Overall Return Trend:** Initial returns may be negative (the agent flails or falls), but over the first 50–100 epochs, returns climb quickly. By epoch 300, returns can exceed 1500 or more (depending on seed, activation, etc.).
2. **Effect of Activation Function:**
   - Tanh and ReLU both improve steadily.
   - Tanh often yields smoother policy updates; ReLU can sometimes converge faster or to a higher final return, but results can vary.
3. **Stability and Variance:** Using a learned baseline significantly reduces the noise in updates, so we observe a clearer upward trend than in a no-baseline scenario.

Comparison to No Baseline

- **Significant Variance:** Without a baseline, half-cheetah tasks often exhibit large swings in returns, especially early on.
- **Less Stable Learning:** Higher-dimensional state/action spaces amplify the variance issue; GAE and a learned value function stabilize updates, accelerating convergence.

3. Summary of Baseline Effects

- **Faster Convergence:** Across both CartPole and HalfCheetah, having a learned baseline (GAE + value function) reduces gradient variance and speeds up learning.
- **Higher Final Performance:** While it is possible to train with no baseline, using one typically leads to higher returns in fewer epochs.
- **Reduced Variance in Updates:** GAE leverages bootstrapping and smoothing with parameter λ, making advantage estimates more stable and preventing drastic policy oscillations.

Overall, these experiments confirm that *adding a learned baseline significantly improves both the speed and stability of policy gradient methods* compared to training without a baseline.

Extra Credit B: Comparison Between PPO Implementation and Simple Policy Gradient

Differences

The PPO implementation significantly extends and improves upon the basic policy gradient approach in the following ways:

- **Baseline and Value Function:** PPO explicitly includes a learned value function as a baseline, which significantly reduces the variance of policy gradient estimates. In contrast, the simple policy gradient method uses raw returns directly as weights, which leads to higher variance.
- **Advantage Estimation (GAE):** PPO employs Generalized Advantage Estimation (GAE) to provide more stable and accurate advantage estimates. GAE combines temporal difference methods and reward-to-go, smoothing out variance and bias.
- **Clipping Mechanism:** PPO includes a clipping mechanism to limit the update step size of the policy, enhancing stability and preventing excessively large updates. The basic policy gradient approach has no such limitation, potentially causing large, unstable policy changes.
- **Early Stopping via KL Divergence:** PPO monitors the KL divergence between the new and old policies, enabling early stopping of gradient updates if policy changes become too large. The simple policy gradient approach does not have such a mechanism.
- **Separate Optimizers for Policy and Value:** PPO updates policy and value networks separately, each with its own optimizer and potentially different learning rates, providing finer-grained control.

- **Parallelization (MPI):** The PPO implementation uses MPI for parallelization across multiple CPU cores to speed up training. The simple policy gradient code is single-threaded and does not exploit parallel computing.

Similarities

Both implementations share fundamental aspects of policy gradient methods:

- **On-Policy Algorithms:** Both PPO and simple policy gradient methods collect data on-policy, meaning the policy used to gather experience is the one being improved.
- **Neural Network Policies:** Both use neural networks (Multi-Layer Perceptrons, MLP) to parameterize policies, mapping states directly to action distributions.
- **Policy Gradient Update:** The underlying principle is the same: update policy parameters by estimating gradients of the expected return.

What I Learned

While reviewing and experimenting with both codes, several key insights emerged:

1. **Variance Reduction:** Implementing a learned baseline and GAE effectively reduces gradient variance, resulting in more stable and efficient learning.
2. **Stability via PPO Clipping:** The PPO clipping strategy ensures policy updates remain within a stable range, significantly improving convergence compared to unrestricted gradient steps.
3. **Importance of Hyperparameters:** Adjusting learning rates, clipping ratios, and GAE parameters significantly affects performance, reinforcing the importance of hyperparameter tuning.
4. **Parallel Computing in RL:** MPI parallelization demonstrated the practical speedup of training large neural networks in computationally intensive environments.

Unclear or Confusing Aspects

Some aspects remain complex or unclear after examining the PPO implementation:

1. **Optimal KL Threshold:** Choosing the correct KL divergence threshold for early stopping is still somewhat ambiguous, as different environments may require different values for optimal performance.

2. **Number of Updates per Epoch:** The optimal number of policy and value updates per epoch (“train pi iters, train v iters) still seems somewhat empirical, without clear theoretical guidelines for setting them optimally.

3. **Impact of GAE Parameters:** The precise impact of GAE’s discount factor γ and smoothing parameter λ on stability and performance requires deeper theoretical understanding and experimentation.

4. **Practical Limitations of MPI:** While parallelization with MPI is powerful, its complexity and overhead might not always justify the effort, especially for smaller tasks or limited computational resources.

8 Conceptual Summary

Simple Policy Gradient directly uses episode returns as gradient weights, providing straightforward implementation but higher variance and instability.

PPO, however, significantly builds upon this foundation by:

- Introducing a learned value-function baseline and advantage estimation for variance reduction.
- Implementing a clipping mechanism and KL divergence-based early stopping for stability.
- Utilizing parallel computation for efficiency.

Overall, PPO can be viewed as an evolution of basic policy gradient methods, addressing many of their inherent weaknesses and improving both stability and efficiency in policy training.