# HW5: Policy Gradients

Atharv Kulkarni - u1322897

28 March 2025

## Part 1a: Basic Policy Gradient Training Results

In this experiment, we implemented and ran a basic policy gradient algorithm on the CartPole-v1 environment using Gymnasium. The goal was to observe how the agent learns over time using only full-trajectory returns. The algorithm was trained for 50 epochs, and we recorded the average return and episode length after each epoch.

From the training logs, we can see that the agent starts with a relatively low return of around 21 in epoch 0. However, as training progresses, the return steadily increases. By epoch 10, the average return has more than doubled, reaching around 56. As we move further, the agent continues to improve its performance, eventually achieving returns well above 200 in the final epochs. The highest recorded average return was approximately 294 in epoch 49.

Overall, the trend in performance shows a clear upward trajectory. Although there are some small fluctuations along the way, the general pattern is one of improvement. This suggests that the agent is effectively learning a better policy through policy gradient updates.

It's important to note that the learning is not strictly monotonic, there are slight dips in performance in some epochs, but the long-term trend is positive. This non-monotonic behavior is expected in reinforcement learning due to the high variance in gradient estimates and the stochastic nature of the environment and policy.

In summary, the agent is definitely learning. Its performance improves significantly over time, demonstrating the effectiveness of the basic policy gradient approach even in its simplest form.

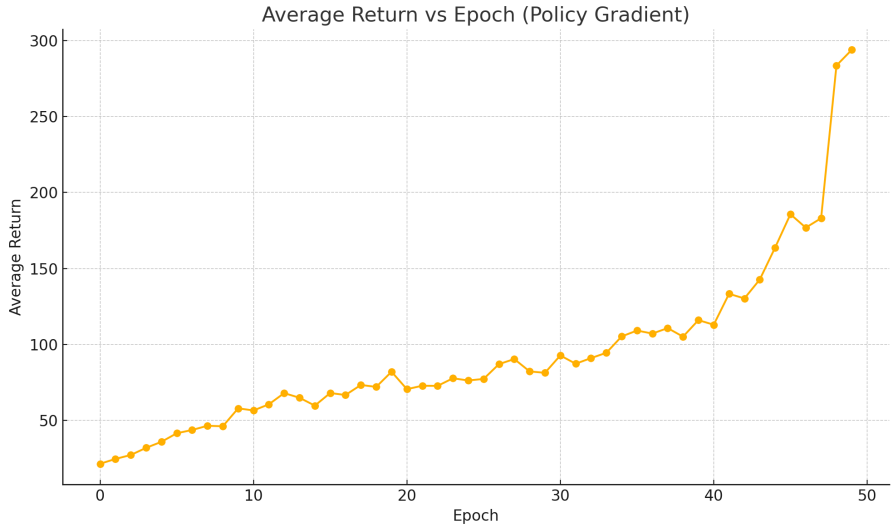



Figure 1: Average Return per Epoch for CartPole-v1 using Basic Policy Gradient

# Part 1b: Visualizing Policy Gradient Learning Behavior

To better understand how the agent's policy evolves during training, we modified our implementation to render one episode using the current policy after each epoch. We created a separate environment instance with `render_mode="human"` to visualize these test rollouts without affecting the training process.

Action selection during test rollouts was wrapped inside a `with torch.no_grad():` block to ensure that gradients were not computed or stored during evaluation. This separation ensures that inference remains efficient and does not interfere with training.

## Observations

In the early stages of training, the agent performed poorly, it often failed to balance the pole for more than a few seconds, and its actions appeared random or overly reactive. However, as training progressed, we observed noticeable improvements in stability and responsiveness. The agent began to make more deliberate and well-timed actions, effectively keeping the pole balanced for longer durations.

By the later epochs, the agent was able to consistently solve the environment, showing that the policy gradient method was successfully optimizing the agent's behavior. The transitions became smoother, and the agent's movements were more refined and less erratic, which indicated improved policy performance.

The following figure is a screenshot captured from one of the rendered test episodes in the later stages of training. Overall, this visualization provided valuable insight into the agent's learning process. Observing the policy improve over time helped verify that the implementation was working as intended and that learning was indeed occurring.

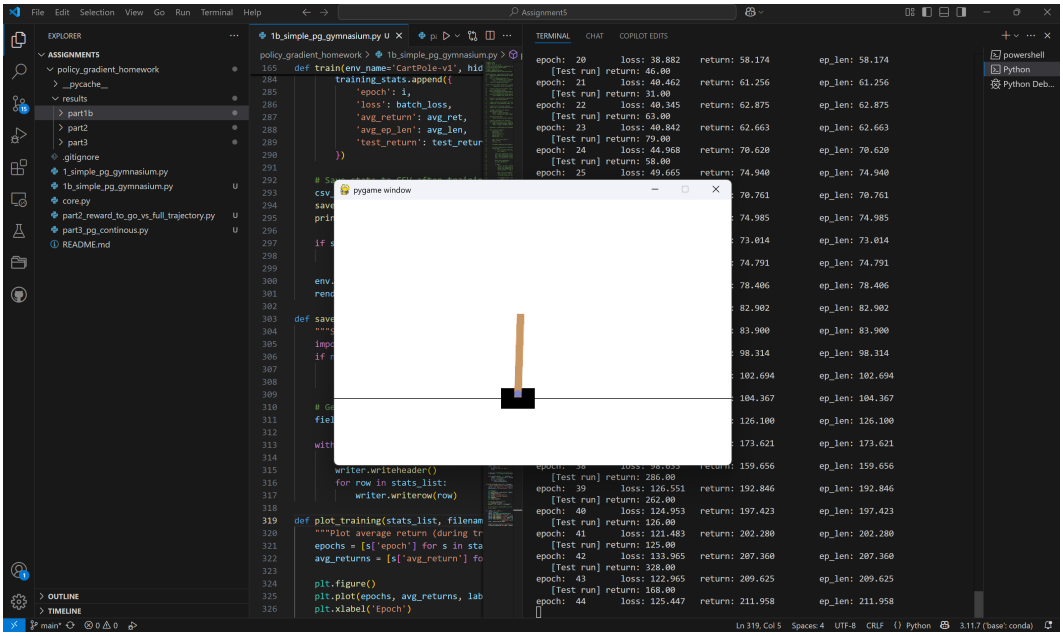



Figure 2: Rendered policy during a test rollout after training. The agent is successfully balancing the pole.

# Part 2: Reducing Variance with Reward-to-Go

To further improve our policy gradient implementation, we implemented the Reward-to-Go variant as described in the SpinningUp tutorial. In this version, instead of assigning the total return $R(\tau)$ to every action in an episode, we assign each action the sum of rewards it receives from its timestep onward. This change is intended to reduce the variance in policy gradient estimates by eliminating the influence of rewards received before each action.

We modified the policy gradient implementation by introducing a `reward_to_go()` function and updating the weights used in the loss function accordingly. The rest of the training logic remained the same.

## Experimental Comparison

To compare the effectiveness of the two approaches, we ran both the Full-Trajectory Return and Reward-to-Go versions of the policy gradient algorithm 3 times each and averaged their results. Each run was conducted on the CartPole-v1 environment for 50 epochs with a batch size of 5000. We tracked the average return at each epoch and plotted average return vs. total timesteps.

## Discussion

From the results shown in Figure 3, we can observe that the Reward-to-Go implementation consistently outperforms the Full-Trajectory Return method across nearly all timesteps. It not only converges faster, but also reaches a higher overall performance.

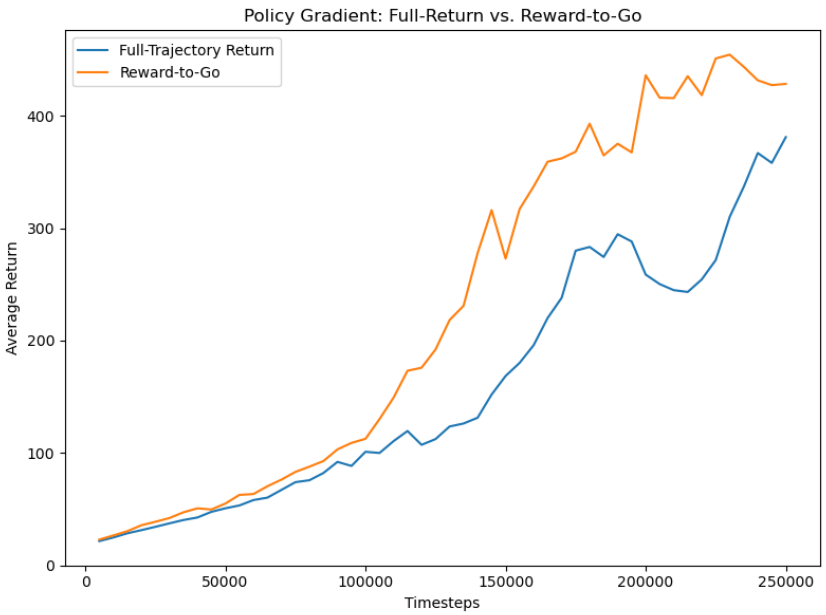



Figure 3: Comparison of Policy Gradient Variants: Full-Trajectory Return vs. Reward-to-Go

This makes sense because, Reward-to-Go reduces the variance in the gradient estimates by excluding unrelated rewards from earlier in the trajectory. As a result, the agent gets a more accurate signal for credit assignment, which leads to faster and more stable learning.

In contrast, the Full-Trajectory Return method assigns the same return to all actions in an episode, regardless of their timing or individual contribution. This can introduce unnecessary noise into the gradient estimation, especially in longer episodes where many actions have no relationship to earlier rewards.

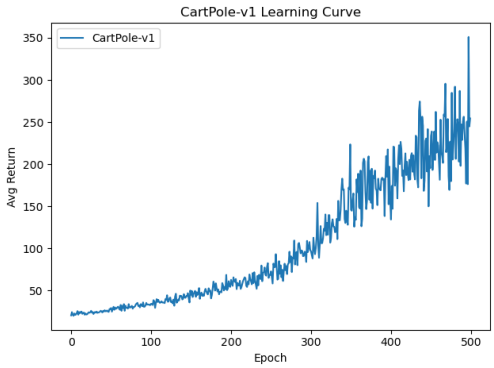
Therefore, based on our experiments, Reward-to-Go is the preferable approach and demonstrates a clear advantage in performance on the CartPole-v1 task.

# Part 3: Continuous Actions

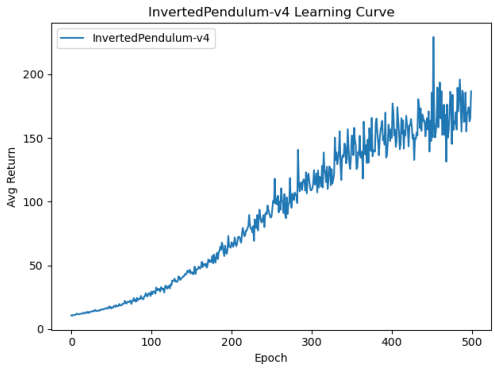
In this part, we extended our policy gradient implementation to support both discrete and continuous action spaces. We achieved this by utilizing the `MLPActorCritic` class provided in `core.py`, which abstracts away the differences between discrete and continuous policies. This allowed us to apply the same training code to a wide range of Gymnasium environments.

We tested our updated implementation on two tasks:

- **CartPole-v1** – a discrete control environment.
- **InvertedPendulum-v4** – a continuous control task using the MuJoCo simulator.




(a) CartPole-v1 (discrete action space)




(b) InvertedPendulum-v4 (continuous action space)

Figure 4: Learning curves for discrete and continuous action environments.

## Hyperparameter Tuning

During experimentation, we compared different hidden layer sizes. We tested both `[32, 32]` and `[64, 64]` architectures and found that using 64 units per layer provided more stable learning and higher returns in both environments.

We also experimented with different learning rates. While a higher learning rate such as $10^{-3}$ allowed for faster initial learning, it occasionally led to instability. A lower learning rate of $10^{-4}$ resulted in slower but more consistent and reliable training, and was selected for the final experiments.

Additionally, we observed that increasing the number of training epochs significantly improved performance. Specifically, increasing from 50 to 200 epochs led to better results, and extending to 500 epochs resulted in continued improvement. Therefore, all reported results in this section are based on training for 500 epochs.

## Results and Observations

The agent learned successfully in both environments, as shown in the learning curves above. For CartPole-v1, the return increased steadily with minimal variance. In the InvertedPendulum-v4 environment, which uses continuous actions, the learning curve followed a similar trend where the return increased steadily in a continuous control task.

## Learning's

This experiment demonstrated that policy gradient methods can be applied to both discrete and continuous action spaces with minimal changes to the underlying algorithm. Using the `MLPActorCritic` class allowed us to generalize our implementation, and tuning hyperparameters such as hidden layer size, learning rate, and number of epochs had a significant effect on performance.