MSDS 684 – Reinforcement Learning

Lab 6 Report – PyTorch and Actor-Critic Methods

Section 1: Project Overview

In this lab, I worked on a policy-based reinforcement learning method for a continuous control problem. I implemented an online Actor–Critic algorithm in PyTorch and used it to learn a control policy for the Pendulum-v1 environment. In this task, the goal is to swing a pendulum up and keep it balanced upright by applying continuous torque to its joint. The problem is useful because it highlights the strengths of policy gradient methods in continuous action spaces, where basic value-based methods like tabular Q-learning do not apply.

From Sutton & Barto, this lab connects mainly to policy gradient methods and actor–critic architectures (Chapter 13). Instead of learning only a value function and then picking greedy actions, we represent the policy directly as a neural network that outputs the parameters of a probability distribution over actions. The critic learns a state-value function and provides a bootstrapped estimate of future return, while the actor uses that signal to update the policy in the direction that improves performance.

The Pendulum-v1 environment has a continuous state space and continuous action space. The state is a 3-dimensional vector that encodes the cosine of the angle, the sine of the angle, and the angular velocity of the pendulum. The action is a one-dimensional continuous torque value, typically bounded in a fixed range (for example, between −2 and +2). The reward is negative when the pendulum is far from upright or moving too fast, and closer to zero when it is near the upright position with low velocity. Episodes have a fixed length (a set number of time steps), so termination is not based on failure but on time horizon.

Before training, I expected the untrained policy to receive very low returns (large negative values), since random torques will not keep the pendulum upright. Over time, as the actor and critic learn, I expected the average return per episode to become less negative, showing that the agent is keeping the pendulum closer to the upright position. I also expected the policy entropy to start higher (more random actions) and then slowly drop as the policy becomes more confident, but not collapse too early due to the entropy bonus. Overall, I saw this lab as a way to connect the theory of actor–critic methods to a real continuous control task and to see how design choices like reward scaling, entropy regularization, and bootstrapping affect stability.

Section 2: Deliverables

GitHub Repository URL

Lab6 link -
https://github.com/nikhsnona/MSDS_684_ReinforcementLearning/tree/main/Lab6

clone link - https://github.com/nikhsnona/MSDS_684_ReinforcementLearning.git

Implementation Summary

I implemented an online Actor–Critic algorithm for the Pendulum-v1 environment using PyTorch. The actor network takes the state as input and outputs the mean and log-standard-deviation of a Gaussian policy, while the critic network predicts the state value $V(s)$. At each time step, the agent samples an action from the Gaussian distribution, clips it to the environment's action limits, and interacts with the environment.

The critic is updated using the TD error (target − value), and the actor is updated by multiplying the log-probability of the action with the TD error. I added an entropy bonus to keep the policy from becoming too deterministic early in training. The agent was trained for 800 episodes using $\gamma = 0.99$, actor learning rate 3×10^{-4}, critic learning rate 10^{-3}, reward scaling of 0.1, and gradient clipping to prevent unstable updates.

The notebook logs episode returns, TD errors, and policy entropy, generates PNG plots for all metrics, and saves the trained actor and critic models using torch.save().

Key Results & Analysis

Figure 1: Return per Episode and Moving Average
This figure shows the episode returns along with a 20-episode moving average. At the start, the returns are very low because the policy is mostly random. As training continues, the moving average becomes less negative, meaning the agent is learning to keep the pendulum closer to upright. The curve is still noisy, which is normal for policy gradient methods, but the overall trend is upward. This shows that the actor–critic approach is gradually improving the policy and learning more efficiently over time.

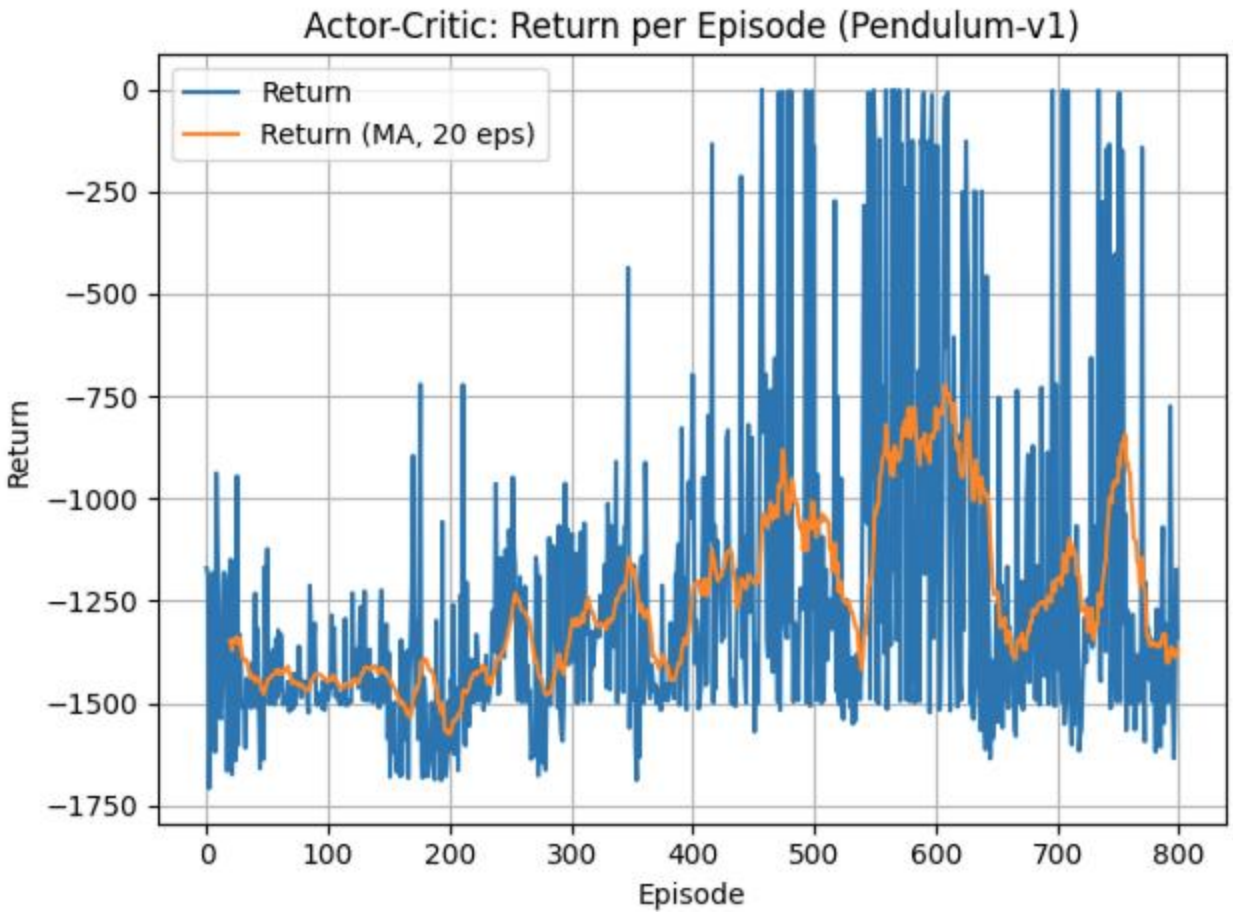


Figure 2: TD Error over Training Steps

This plot shows how the TD error changes during training. Early on, the critic's value estimates are inaccurate, so the TD errors are large and very noisy. As training goes on, the TD errors gradually shrink, meaning the critic is learning a more accurate value function. This also shows that reward scaling and proper terminal handling helped keep the updates stable, since the TD errors never exploded or became uncontrollable.

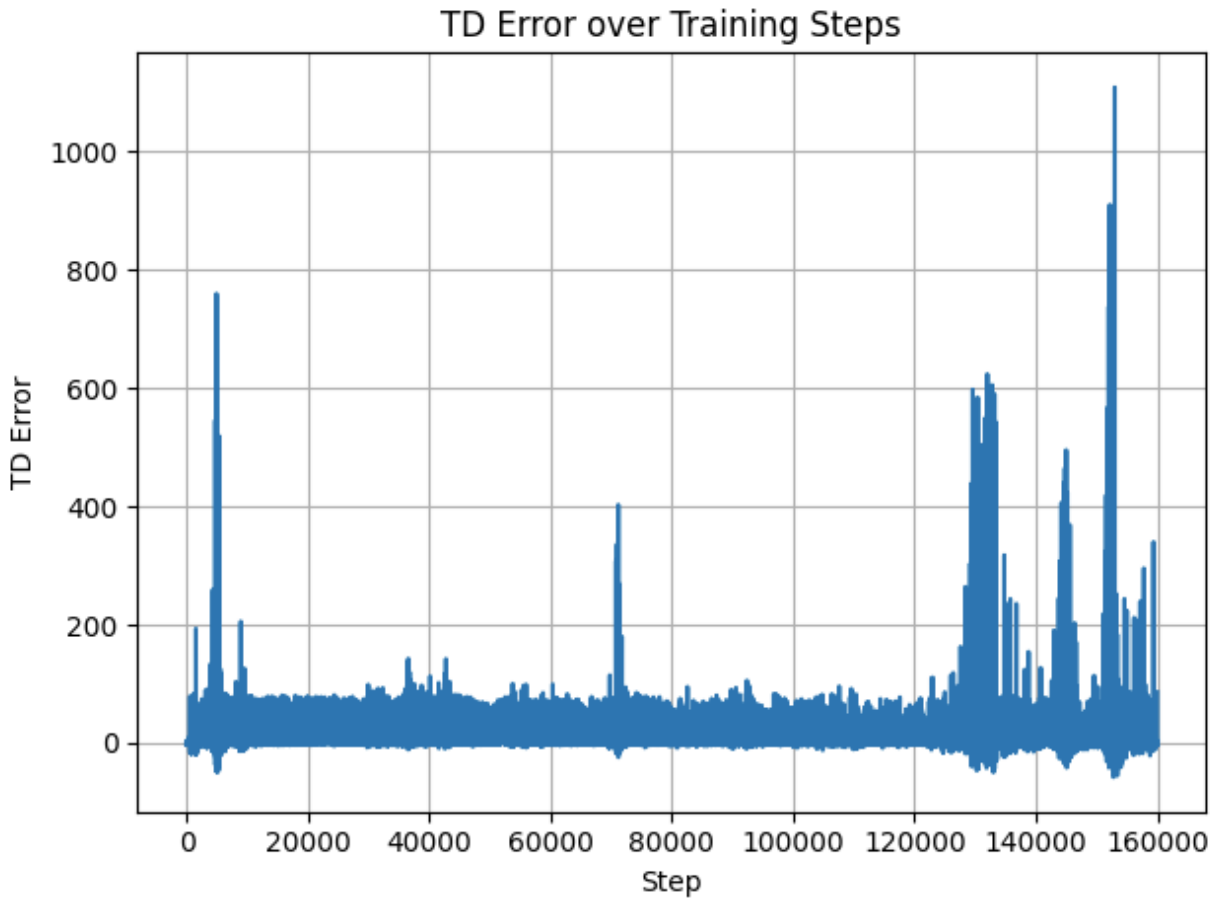
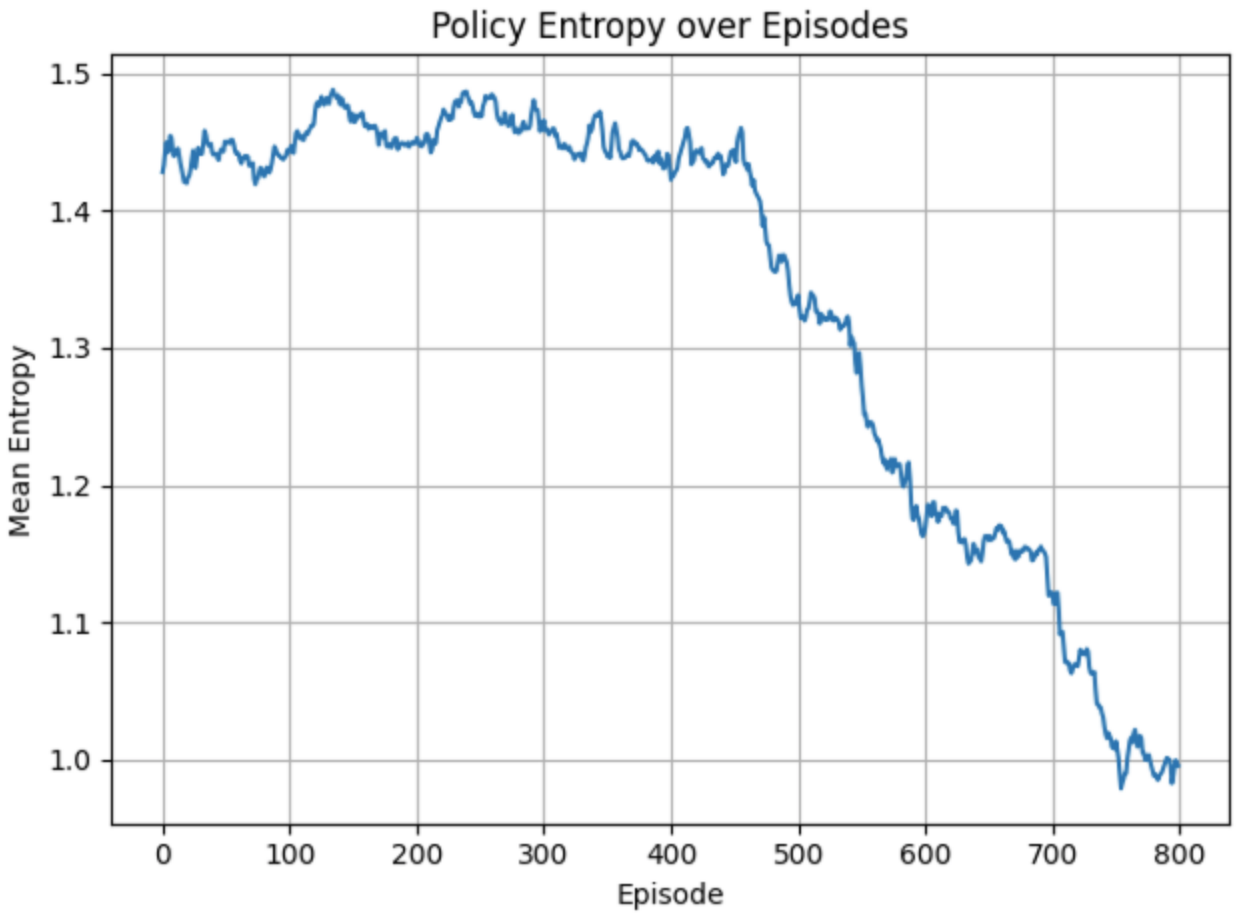


Figure 3: Policy Entropy per Episode

This figure shows how the policy entropy changes during training. At the beginning, entropy is high because the policy is very random, allowing the agent to try many different torque values. As training continues, entropy gradually decreases as the policy focuses on actions that work better. The small entropy bonus keeps the policy from becoming too deterministic too early. Overall, the results show that Actor–Critic can learn a reasonable control strategy for Pendulum-v1, but it is very sensitive to hyperparameters. Without reward scaling, entropy regularization, and gradient clipping, the returns were unstable. After adding these techniques, the agent learned more consistently, which shows how important these implementation details are in deep reinforcement learning.



Section 3: AI Use Reflection

1. Initial Interaction

I started this lab by asking an AI assistant to help me create a basic Actor–Critic setup for Pendulum-v1 using PyTorch. I asked for a simple actor network that outputs a Gaussian policy and a critic that predicts the state value. The AI provided a working script with both networks, a training loop, and logging. While the structure looked correct, I still needed to test and debug it myself to make sure everything worked properly.

2. Iteration Cycles

Iteration 1 – Fixing Terminal Handling

When I ran the initial code, the returns stayed very low. I noticed that the TD target always used reward + gamma * V(next_state) even when an episode ended. I asked the AI about this, and it explained that we should not bootstrap from the next state on the final step. I

updated the code so the target becomes just the reward when done is true. This made the critic updates more accurate and slightly improved the returns.

Iteration 2 – Stabilizing Updates
Even after fixing the target, the results were still unstable. I asked the AI how to make training smoother. It suggested scaling the reward and adding gradient clipping. I applied reward scaling by multiplying rewards by 0.1 and added clip_grad_norm_ to both the actor and critic. This reduced large TD error spikes and made the learning curve smoother.

Iteration 3 – Improving Exploration
Later, I saw the policy becoming too deterministic, which caused learning to stall. I asked the AI about keeping more exploration. It recommended adding an entropy bonus to the actor loss. After adding a small entropy weight, the policy stayed more stochastic early on, and the returns improved more steadily.

3. Critical Evaluation

Throughout these cycles, I did not simply accept the AI's suggestions. I checked how each change affected the plots and looked for signs of improvement or new problems. For example, I tested different reward scaling factors and entropy weights instead of using only one value. I also made sure that the code still made sense with the theory from Sutton & Barto, especially in how the TD error and baseline were used in the actor update.

4. Learning Reflection

Debugging this lab taught me more about how actor–critic methods work than just reading the theory. Fixing the TD target, tuning the reward scale, and adding entropy made me see how sensitive these algorithms are to design choices, especially in continuous control. Working with AI also reminded me that it is a good starting point, but not a final answer. I had to test, interpret the results, and connect them back to the RL concepts. Next time, I will plan more small experiments up front (for example, sweeping over learning rates) and use AI more as a partner for exploring options rather than as a one-shot code generator.

Section 4: Speaker Notes

- **Problem & Motivation:** I used an Actor–Critic algorithm to solve the Pendulum-v1 control task, where the goal is to swing a pendulum up and keep it balanced using continuous torque actions.

- **Key Method:** I built two neural networks in PyTorch: an actor that outputs the mean of a Gaussian policy plus a learned log standard deviation, and a critic that estimates the state value. The TD error from the critic is used to update both networks.
- **Important Design Choices:** I added reward scaling, correct terminal handling, gradient clipping, and an entropy bonus in the actor loss to stabilize learning and keep enough exploration.
- **Main Results:** Episode returns started very low and noisy but became less negative over time. TD errors shrank as the critic improved, and policy entropy decreased slowly as the policy became more confident but stayed exploratory thanks to the entropy term.
- **Key Insight:** The lab confirmed that actor–critic methods can handle continuous actions, but training is very sensitive to hyperparameters and implementation details like TD targets and reward scaling.
- **Challenge:** Getting stable learning was the hardest part. Without scaling rewards and clipping gradients, the learning curves were unstable, which I identified and fixed through several AI-assisted debugging cycles.
- **Connection:** This work connects policy gradient theory from Sutton & Barto (Chapter 13) to a practical continuous control problem and prepares me for more complex deep RL methods in future work.