ATCI Coursework 1: RL Implementation
Proximal Policy Optimization (PPO) [1]

Bruno Sánchez Gómez

May 7, 2025

1 Introduction

Reinforcement Learning (RL) has emerged as a powerful paradigm for training agents to make sequential decisions in complex environments. Among the various RL algorithms, Proximal Policy Optimization (PPO) [1] stands out due to its robust performance, sample efficiency, and relative ease of implementation. PPO has become a popular choice for a wide range of control tasks, from simple game environments to complex robotic simulations.

This report details the implementation and evaluation of the PPO algorithm. The primary objective was to develop a functional PPO agent and assess its capabilities across a selection of benchmark environments from the Gymnasium library [2]. Specifically, we test our implementation on CartPole-v1, a classic discrete control problem, and on two more challenging continuous control tasks from the MuJoCo suite: HalfCheetah-v5 and Reacher-v5.

Our PPO implementation incorporates key algorithmic components such as an actor-critic architecture, Generalized Advantage Estimation (GAE) [3] for stable advantage calculation, and the characteristic clipped surrogate objective function of PPO to ensure stable policy updates. The agent's neural networks utilize Tanh activation functions, observation normalization, and specific weight initialization techniques inspired by established practices [4].

The implemented agent was successfully trained on the selected environments, achieving performance levels comparable to those reported in the original PPO paper [1]. To complement the quantitative results, an interactive Streamlit application was developed, allowing for the visualization of the trained agents' policies in their respective environments.

This report is structured as follows: Section 2 describes the PPO algorithm, the specifics of our implementation, and details of the environments used. Section 3 presents the experimental results, including learning curves and performance comparisons. Finally, Section 4 summarizes the findings and discusses potential avenues for future work.

2 Methodology

2.1 Algorithm

The core of this project is the implementation of the Proximal Policy Optimization (PPO) algorithm [1]. PPO is an on-policy, actor-critic reinforcement learning algorithm designed for its stability and sample efficiency. It optimizes a "clipped" surrogate objective function to prevent excessively large policy updates, which can lead to performance collapse. Our implementation incorporates several key features:

- **Actor-Critic Architecture:** The agent employs an actor-critic model, as defined in `src/models.py`. This architecture consists of two main components: an actor that learns the policy (mapping states to actions) and a critic that learns the value function (estimating the expected return from a given state). The implementation supports both separate networks for the actor and critic, as well as a shared feature layer followed by separate heads. The neural networks primarily use Tanh activation functions in their hidden layers. Observation normalization is applied to the input states using a running mean and standard deviation filter (`RunningMeanStd` in `src/models.py`) to stabilize training, especially in environments with varying state scales. The final layers of the actor and critic networks are initialized using a method similar to "normc" initialization, as implemented in the original PPO code [4].

- **Clipped Surrogate Objective:** PPO's hallmark is its clipped surrogate objective function. This objective limits the change in the policy at each update step by clipping the probability ratio between the new and old policies. The clipping range is determined by a hyperparameter ϵ (referred to as `PPO_EPSILON` in `src/config.py`). This mechanism helps to ensure more stable and reliable training by preventing destructive large updates. The specific implementation of this objective can be found in the `update` method within `src/ppo.py`.
- **Generalized Advantage Estimation (GAE):** To estimate the advantage function $A(s, a) = Q(s, a) - V(s)$, our implementation uses Generalized Advantage Estimation (GAE) [3]. GAE provides a an effective trade-off between bias and variance in the advantage estimates by using an exponentially-weighted average of multi-step TD errors. The calculation is performed in the `_calculate_gae` method of the `PPOMemory` class (see `src/memory.py`), using hyperparameters γ (`GAMMA`) for discounting future rewards and λ (`GAE_LAMBDA`) for controlling the GAE trade-off. The advantages can optionally be normalized (`NORMALIZE_ADVANTAGES` in `src/config.py`) before being used in the policy update.
- **Multiple Epochs and Minibatch Updates:** For each batch of collected experience (transitions from multiple actors over `PPO_STEPS`), the PPO algorithm performs multiple epochs of updates (`PPO_EPOCHS`) using randomly sampled minibatches (`MINI_BATCH_SIZE`). This allows for better sample utilization and more stable learning. This process is managed within the `update` method of the `PPOAlgorithm` class in `src/ppo.py`.
- **Entropy Bonus:** To encourage exploration and prevent premature convergence to suboptimal policies, an optional entropy bonus can be added to the loss function. This bonus is scaled by a hyperparameter $\beta_{entropy}$ (`ENTROPY_BETA` in `src/config.py`).
- **Optimization:** The actor and critic networks are optimized using the Adam optimizer [5] with a configurable learning rate (`LEARNING_RATE`).

2.2 Environments

All environments used in this project are sourced from the Gymnasium library [2], the successor to OpenAI Gym. For continuous control tasks, the `ClipAction` wrapper from Gymnasium is utilized to ensure that actions selected by the agent remain within the valid bounds defined by the environment's action space, as seen in `src/train.py`.

- **CartPole-v1:**
  - *Description:* This is a classic reinforcement learning problem where the goal is to balance a pole upright on a cart that moves along a frictionless track.
  - *State Space:* Continuous, 4-dimensional, representing the cart's position and velocity, and the pole's angle and angular velocity.
  - *Action Space:* Discrete, with 2 actions: push the cart to the left or to the right.

    - *Purpose:* Due to its simplicity and fast training times, CartPole-v1 serves as an initial benchmark to verify the correctness and basic functionality of the PPO implementation.

- **MuJoCo Environments (via Gymnasium-Robotics):** These environments utilize the MuJoCo physics engine [6] and represent more challenging continuous control tasks.
    - **HalfCheetah-v5:**
        - *Description:* This environment features a 2D simulated cheetah robot. The objective is to make the cheetah run forward as fast as possible.
        - *State Space:* Continuous, 17-dimensional, including joint positions, joint velocities, and other physical properties of the cheetah's body parts.
        - *Action Space:* Continuous, 6-dimensional, representing the torque applied to each of the cheetah's six actuated joints.
        - *Purpose:* This is a standard benchmark for continuous control algorithms. The results obtained on HalfCheetah-v5 are compared with those reported in the original PPO paper to gauge the performance of our implementation.
    - **Reacher-v5:**
        - *Description:* In this environment, a two-jointed robotic arm must reach a randomly positioned target in its workspace.
        - *State Space:* Continuous, 11-dimensional, typically including the cosine and sine of joint angles, joint angular velocities, and the Cartesian coordinates of the target.
        - *Action Space:* Continuous, 2-dimensional, representing the torques applied to the two joints of the arm.
        - *Purpose:* Similar to HalfCheetah, Reacher-v5 is another common benchmark for continuous control. It allows for further evaluation of the PPO implementation's capabilities on tasks requiring precise motor control, and results are also compared against the PPO paper.

3 Results

This section presents the performance of our PPO implementation across the selected environments. The training progress is illustrated with learning curves that show the average reward per step over each update cycle, and the final performance is compared with established benchmarks where applicable. All agents were trained for a specified number of timesteps, and their final performance was evaluated by averaging rewards over 100 test episodes generated with the learned policy, as detailed in the evaluation results CSV file (`results/evaluation_results.csv`).

To further illustrate the learned behaviors, the trained agents can be visualized in action within their respective environments. For each environment, three pre-generated video renders showcasing the agent's policy are available in the 'renders' subdirectory within the corresponding agent's results folder. Additionally, an interactive Streamlit application has been developed to facilitate this visualization. This application allows users to select an environment and generate new renders on-the-fly using our trained agents. The Streamlit visualizer can be accessed at: `https://ppo-visualizer.streamlit.app/`

Note: When using the app, please mind that the render generation process may take several seconds to complete, especially for the MuJoCo environments, since it is being generated in real-time in the dedicated Streamlit Cloud machine.

3.1 CartPole-v1

The CartPole-v1 environment serves as a fundamental benchmark for reinforcement learning algorithms. The objective is to balance a pole on a cart for as long as possible. Our PPO agent was trained for 20,000 timesteps.

The agent successfully learned to solve this environment, achieving the maximum possible average reward of **500.0** over 100 test episodes. The learning curve, depicted in Figure 1, shows a rapid increase in reward, quickly reaching and maintaining optimal performance.

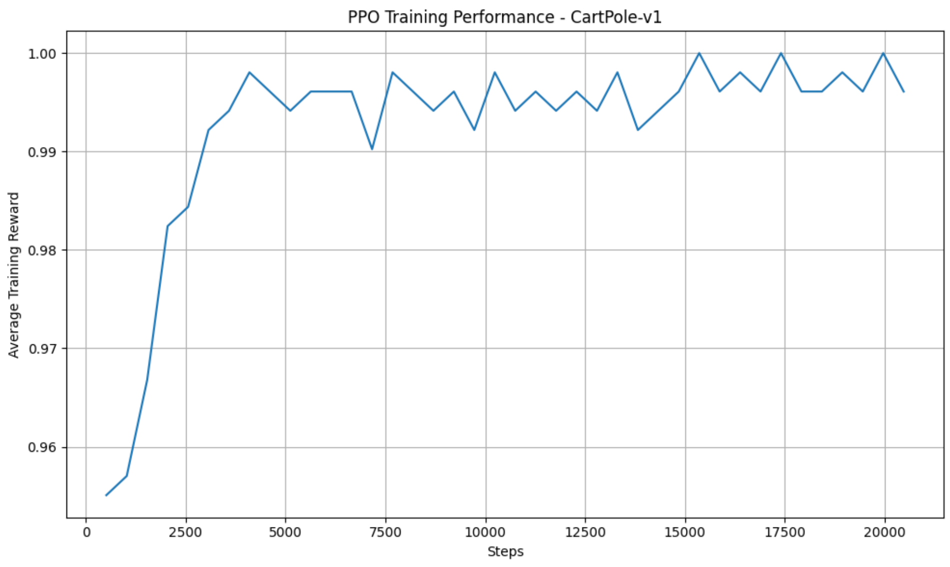

Figure 1: Training progress for PPO on CartPole-v1. The plot shows the average per-step reward (averaged over all steps taken during an update cycle) against training timesteps.

This result confirms the fundamental correctness and effectiveness of our PPO implementation on a classic discrete control task. The configuration for this agent, detailed in the results CSV file. This is the only experiment for which we utilized shared features between the actor and critic and normalized advantages.

3.2 HalfCheetah-v5

HalfCheetah-v5 is a more complex continuous control task from the MuJoCo suite, where a 2D cheetah robot aims to run forward as quickly as possible. The agent was trained for 1 million timesteps.

Our PPO implementation achieved a final average reward of approximately **1770.62** over 100 test episodes. The training progress is shown in Figure 2. The learning curve demonstrates steady improvement over the training duration, indicating successful learning in this high-dimensional continuous space, though with some variance typical of complex RL tasks.

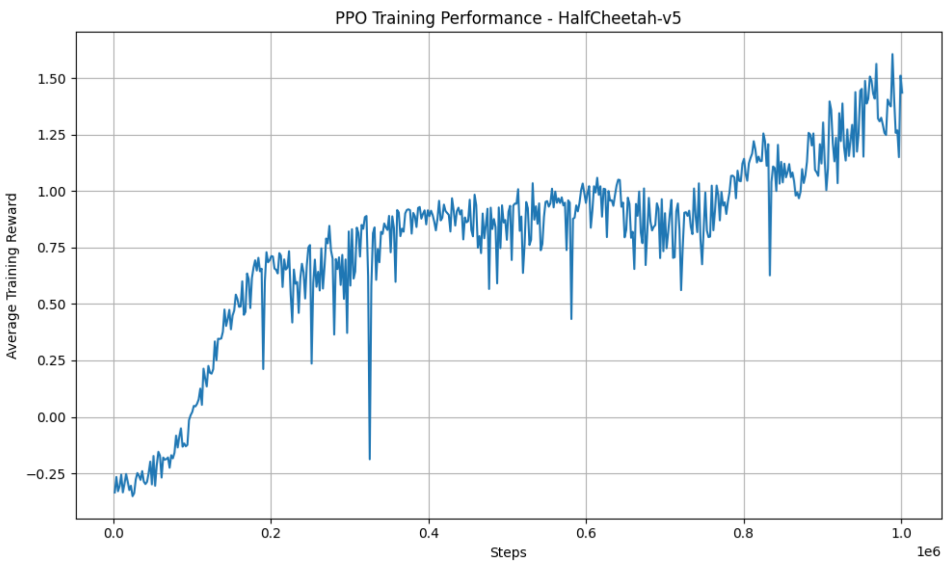

Figure 2: Training progress for PPO on HalfCheetah-v5.

Comparing our result to the original PPO paper [1], which reports scores for PPO variants on HalfCheetah typically ranging from 1500 to 2100 after 1 million timesteps, our achieved score of approximately **1770.62** is well within this expected performance range. The configuration used for this experiment was based on the settings reported in the original PPO paper for MuJoCo environments, aiming to make our results as comparable as possible. Specific hyperparameter details can be found in the results file. Notably, this run employed separate networks for the actor and critic and did not normalize advantages.

3.3 Reacher-v5

Reacher-v5 is another MuJoCo continuous control task where a two-jointed robotic arm must reach a randomly positioned target. The agent was trained for 1 million timesteps.

The PPO agent achieved a final average reward of approximately **-8.45** over 100 test episodes. The learning curve is presented in Figure 3. The plot shows the agent quickly learning to improve its performance and stabilizing at an optimal policy.

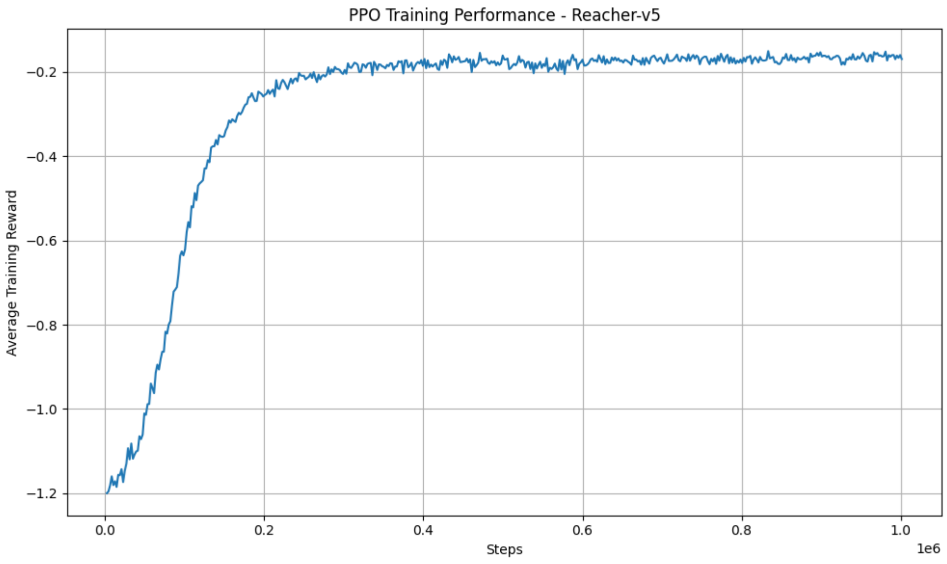

Figure 3: Training progress for PPO on Reacher-v5.

The original PPO paper [1] reports scores for Reacher typically around -5 to -10 after 1 million timesteps. Our achieved score of **-8.45** falls comfortably within this expected range, indicating a competent performance of our implementation on this task. Similarly to the HalfCheetah experiment, we used the same hyperparameters as in the original PPO paper for MuJoCo environments. The configuration details are available in the results file. This run also employed separate networks for the actor and critic and did not normalize advantages.

The results across these diverse environments demonstrate the capability of our PPO implementation to learn effective policies for both discrete and continuous control tasks, achieving performance comparable to established benchmarks.

4 Conclusion

In this project, we successfully implemented the Proximal Policy Optimization (PPO) algorithm and evaluated its performance on a diverse set of Reinforcement Learning environments: CartPole-v1, HalfCheetah-v5, and Reacher-v5. Our implementation incorporated key PPO features, including an actor-critic architecture with optional shared layers, Generalized Advantage Estimation (GAE), a clipped surrogate objective, multiple update epochs, and an entropy bonus for exploration.

The experimental results demonstrate that our PPO agent effectively learns policies for both discrete and continuous control tasks. In the CartPole-v1 environment, the agent quickly achieved optimal performance, mastering the task to a perfect reward of **500.0** within 20,000 timesteps. For the more complex MuJoCo environments, HalfCheetah-v5 and Reacher-v5, our agent achieved average rewards of approximately **1770.62** and **-8.45**, respectively, after 1 million timesteps. These results are consistent with the performance benchmarks reported in the original PPO paper [1], validating the correctness and effectiveness of our implementation.

Furthermore, we developed an interactive Streamlit application to visualize the behavior of the trained agents. This tool allows for on-the-fly rendering of agent performance, providing a qualitative understanding of the learned policies and enhancing the interpretability of the results.

Future work could explore several directions. Firstly, extending the range of environments to include more complex tasks or different domains (e.g., pixel-based observations) would further test the robustness and scalability of the implementation. Secondly, a more extensive hyperparameter optimization study could potentially lead to improved performance on the current set of tasks. Investigating the impact of different network architectures, such as using different activation functions or layer configurations, could also yield performance benefits. Finally, incorporating more advanced techniques, such as curiosity-driven exploration or recurrent policies for partially observable environments, could expand the capabilities of the agent.

In summary, this project provides a solid implementation of the PPO algorithm, demonstrates its successful application to standard benchmark tasks, and offers a foundation for further research and development in the field of reinforcement learning.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[2] Farama Foundation. Gymnasium. `https://github.com/Farama-Foundation/Gymnasium`, 2023.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael I Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[4] OpenAI. OpenAI Baselines: PPO1. `https://github.com/openai/baselines/tree/master/baselines/ppo1`, 2017. Original source code for the PPO1 algorithm from the OpenAI Baselines repository.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.