Proximal Policy Optimization with Clipping and GAE for a Pendulum Environment

August 15, 2025

Abstract

We implement Proximal Policy Optimization (PPO) with the clipped surrogate objective and Generalized Advantage Estimation (GAE) for a continuous-state continuous-action Pendulum environment. This document presents the full and detailed mathematical formulation of the algorithm and a mapping to the Python implementation.

1 Environment

The Pendulum environment has the following properties:

- **State:** At each discrete timestep $t \in \{0, 1, \ldots, T-1\}$, the pendulum's state is represented as
$$s_t = (\cos\theta_t,\ \sin\theta_t,\ \dot{\theta}_t) \in [-1, 1]^2 \times [-8, 8],$$
where θ_t is the angular displacement of the pendulum from the upright position (in radians), and $\dot{\theta}_t$ is its angular velocity (in radians per second).
- **Action:** $a_t \in [-2, 2]$, representing the continuous torque applied to the pendulum at time t. Positive and negative values correspond to torques in opposite directions.
- **Reward:** The instantaneous reward at time t is
$$r_t = -\left(\theta_t^2 + 0.1\,\dot{\theta}_t^2 + 0.001\,a_t^2\right),$$
which penalizes deviation from the upright position ($\theta_t = 0$), high angular velocities, and large control efforts.
- **Episode termination:** Episodes are truncated after a fixed horizon T timesteps. There is no terminal state in this environment; the `done` flag is only triggered by this time-limit truncation.

2 Policy and Value Networks

In our Proximal Policy Optimization (PPO) implementation, the agent is composed of two separate neural networks: the **policy network** $\pi_\theta(a|s)$ and the **value network** $V_\phi(s)$. Each network has its own set of parameters, denoted by θ and ϕ respectively. The networks are trained simultaneously, but with different learning objectives.

2.1 Policy Network

The role of the policy network is to decide *which action to take* given the current state s_t. We denote the network by f_θ, where θ represents its parameters. The network maps a state s_t to the mean action $\mu_\theta(s_t)$ of a Gaussian distribution, defining the stochastic policy:

$$a_t \sim \pi_\theta(\cdot \mid s_t) = \mathcal{N}\big(\mu_\theta(s_t),\ \sigma^2 I\big),$$

where:

- $\mu_\theta(s_t) = 2\tanh\big(f_\theta(s_t)\big)$ ensures the action mean lies within $(-2, 2)$ to match the environment's torque bounds.
- σ is a fixed standard deviation ($\sigma = 0.5$ in our implementation), making $\Sigma = \sigma^2 I$ a constant, diagonal covariance matrix.

The network f_θ is implemented as a multi-layer perceptron (MLP) with two hidden layers. Its output is first bounded to $(-1, 1)$ by the tanh activation and then scaled by 2 to match the action range $[-2, 2]$. Finally, the action a_t is sampled from $\mathcal{N}(\mu_\theta(s_t), \sigma^2 I)$ and clipped to remain within the valid range.

2.2 Value Network

The value network estimates the *value* of a state, which is the expected future reward starting from that state when following the current policy:

$$V_\phi(s_t) \approx \mathbb{E}_{a_{t:\infty}\sim\pi_\theta}\left[\sum_{l=0}^{\infty} \gamma^l r_{t+l}\right]$$

This formula defines the *true* state value function, which is generally intractable to compute exactly. In practice, for each state s_t visited during training, the value network (a multi-layer perceptron) outputs a *single scalar* $V_\phi(s_t)$, which is the *parametrized* estimate of this value under the current parameters ϕ. This estimate is then used as a *baseline* in the policy gradient computation, reducing variance by measuring how much better or worse the observed return was compared to what was expected for that state.

3 Training Procedure

The PPO training loop alternates between two main stages: (1) *collecting trajectories* (rollouts) by running the current policy in the environment, and (2) *updating the policy and value networks* using these collected samples.

3.1 Data Collection

At the start of each iteration:

1. The environment is reset, producing the initial state s_0 via the call `s_0 = env.reset()`, which returns the starting observation of the pendulum in the form $(\cos\theta_0, \sin\theta_0, \dot{\theta}_0)$.
2. For each timestep $t = 0, \ldots, T-1$:
   - Sample the action by the Gaussian policy from the Policy Network(see Section **??**):

$$a_t \sim \pi_\theta(\cdot \mid s_t) = \mathcal{N}(\mu_t, \sigma^2 I)$$

- Record the state, chosen action, reward, and the log-probability under the *old* policy:

$$\ell_t^{\text{old}} = \log \pi_\theta(a_t \mid s_t)$$

  This ℓ_t^{old} is required in the PPO clipped objective later.

- Apply the action to the environment, obtaining the next state s_{t+1} and reward r_t. In the code, this is done using the Gym environment method:

$$s_{t+1}, r_t, \_, \_ = \texttt{env.step}(a_t)$$

  This call advances the simulation by *one step* (smallest time unit) using the applied torque a_t, returning the next state, the immediate reward, and additional info (unused here).

Repeating this step T times forms *one trajectory* (or episode) of fixed horizon T. Collecting N such trajectories before an update produces the PPO *rollout batch*.

At each timestep t of a trajectory, we store the following *parametrized tuple*:

$$\tau_t = (s_t,\ a_t,\ r_t,\ \log \pi_\theta(a_t \mid s_t),\ V_\phi(s_t))\,,$$

where:

- s_t is the state at step t,
- a_t is the action taken,
- r_t is the reward received,
- $\log \pi_\theta(a_t \mid s_t)$ is the log-probability under the *old* policy (before update), stored for computing the PPO probability ratio later,
- $V_\phi(s_t)$ is the current value function estimate for s_t.

  A *trajectory* of horizon T is then:

$$\mathcal{T} = \{\tau_0, \tau_1, \ldots, \tau_{T-1}\}\,,$$

and the full *rollout batch* collected over N trajectories in one iteration is:

$$\mathcal{D} = \bigcup_{i=1}^{N} \mathcal{T}_i.$$

3.2 Advantage Estimation (GAE)

Once we have full trajectories, we need to estimate how *good* each taken action was compared to the baseline value estimate (the current value of "next state" that the Value Network yields.) The *temporal-difference (TD) residual* at time t is:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

This measures the one-step improvement over the value estimate.

Instead of using just one step or the full return, PPO uses *Generalized Advantage Estimation (GAE)* to compute a trade-off between bias and variance:

$$\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}, \quad \hat{A}_T = 0$$

Here:

- $\lambda \in [0, 1]$ controls the weighting between short-horizon TD estimates and long-horizon returns.
- γ is the discount factor.
- The recursion runs *backwards* from the end of the trajectory.

To stabilize training, we normalize the advantages across the whole batch:

$$\tilde{A}_t = \frac{\hat{A}_t - \mu(\hat{A})}{\sigma(\hat{A}) + 10^{-8}}$$

where $\mu(\hat{A})$ and $\sigma(\hat{A})$ are the mean and standard deviation over the batch, and 10^{-8} prevents division by zero.

The *value target* for the critic is defined as:

$$R_t = V_\phi(s_t) + \hat{A}_t$$

This choice is consistent with using GAE and ensures that the value network is trained to match the observed returns.

3.3 Loss Functions

PPO optimizes a composite loss that balances three objectives.

1. Policy loss (clipped surrogate objective) We want to improve the policy in the direction of the estimated advantages but avoid large, destabilizing updates. In theory, the probability ratio between the new and old policies is:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}.$$

In the code, this ratio is computed in a numerically stable way by storing the *old* log-probability $\ell_t^{\text{old}} = \log \pi_{\theta_{\text{old}}}(a_t|s_t)$ at data collection time, computing the *new* log-probability $\ell'_t = \log \pi_\theta(a_t|s_t)$ during the update phase, and forming:

$$r_t(\theta) = \exp\left(\ell'_t - \ell_t^{\text{old}}\right).$$

Both expressions are mathematically equivalent, but the log-difference form avoids numerical underflow/overflow.

The PPO clipped objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\tilde{A}_t,\ \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\tilde{A}_t\right)\right]$$

This keeps updates within a trust region defined by ε; if $r_t(\theta)$ moves outside the interval $[1 - \varepsilon, 1 + \varepsilon]$, the objective is clipped, preventing excessive policy shifts.

2. Value loss (critic update) The critic is trained by minimizing the mean-squared error between predicted values and targets:

$$L^{\text{VF}}(\phi) = \frac{1}{2}\mathbb{E}_t\left[\left(V_\phi(s_t) - R_t\right)^2\right]$$

where the expectation over t means simply an average over all parametrized tuples in all trajectories in all rollouts. This ensures V_ϕ approximates the true expected return for each state under the current policy.

3. Entropy bonus (exploration incentive) To discourage premature convergence to a deterministic policy, an entropy term is added:

$$\mathcal{H} = \mathbb{E}_t \left[\mathcal{H}\big(\pi_\theta(\cdot|s_t)\big) \right]$$

Higher entropy means more randomness in action selection, which promotes exploration of the state-action space.

4. Total loss The final objective combines the above terms:

$$\mathcal{L}(\theta, \phi) = -L^{\text{CLIP}}(\theta) + c_1 L^{\text{VF}}(\phi) - c_2 \mathcal{H}$$

where c_1 and c_2 weight the contributions of the value loss and entropy bonus relative to the policy loss. The negative sign in front of L^{CLIP} reflects that we *maximize* the surrogate objective but *minimize* the total loss in gradient descent.

Notation/Mapping to Code.

Algorithm	Code Variable
Horizon T (steps/trajectory)	`num_timesteps`
Trajectories per iteration N	`num_trajectories`
Iterations K	`num_iterations`
Epochs E	`epochs`
Discount γ	`gamma`
GAE λ	`lambda_`
Clip ε	`eps`
Value coeff. c_1	`vf_coef`
Entropy coeff. c_2	`entropy_coef`
LR α	`learning_rate`

4 Algorithm

Algorithm 1 PPO with Clipping and GAE

Inputs: horizon T, trajectories per iter N, iterations K, epochs E, discount γ, GAE λ, clip ε, coeffs c_1, c_2, learning rate α.
Initialize policy θ, value ϕ.
for $k = 1$ to K **do** ▷ PPO iterations
  $\mathcal{D} \leftarrow \emptyset$
  for $i = 1$ to N **do** ▷ Collect N on-policy trajectories
    $s_0 \leftarrow \text{env.reset}()$
    for $t = 0$ to $T - 1$ **do**
      $\mu_t \leftarrow 2 \tanh\big(f_\theta(s_t)\big)$
      $a_t \sim \mathcal{N}(\mu_t, \sigma^2 I), \quad \tilde{a}_t \leftarrow \text{clip}(a_t, -2, 2)$
      $\ell_t^{\text{old}} \leftarrow \log \pi_\theta(\tilde{a}_t \mid s_t)$
      $s_{t+1}, r_t \leftarrow \text{env.step}(\tilde{a}_t)$
      Append $(s_t, \tilde{a}_t, r_t, \ell_t^{\text{old}})$ to $\mathcal{D}$
    end for
    Compute values, GAE, and targets per trajectory
    for each trajectory in $\mathcal{D}$ **do**
      $v_t \leftarrow V_\phi(s_t)$, set $v_T \leftarrow 0$; $\delta_t \leftarrow r_t + \gamma v_{t+1} - v_t$
      $\hat{A}_T \leftarrow 0$; **for** $t = T - 1 \to 0$: $\hat{A}_t \leftarrow \delta_t + \gamma\lambda \hat{A}_{t+1}$
      $R_t \leftarrow v_t + \hat{A}_t$
    end for
    Normalize advantages: $\tilde{A} \leftarrow \text{norm}(\hat{A})$
    for $e = 1$ to E **do** ▷ Full-batch PPO updates
      $\mu'_t \leftarrow 2 \tanh(f_\theta(s_t)), \quad \ell'_t \leftarrow \log \pi_\theta(\tilde{a}_t \mid s_t)$
      $r_t(\theta) \leftarrow \exp(\ell'_t - \ell_t^{\text{old}})$
      $L^{\text{CLIP}} \leftarrow \frac{1}{M} \sum_t \min\big(r_t \tilde{A}_t,\ \text{clip}(r_t, 1-\varepsilon, 1+\varepsilon)\tilde{A}_t\big)$
      $L^{\text{VF}} \leftarrow \frac{1}{M} \sum_t \|V_\phi(s_t) - R_t\|^2, \quad \mathcal{H} \leftarrow \frac{1}{M} \sum_t \mathcal{H}(\mathcal{N}(\mu'_t, \sigma^2 I))$
      $\mathcal{L}(\theta, \phi) \leftarrow -L^{\text{CLIP}} + c_1 L^{\text{VF}} - c_2 \mathcal{H}$
      Adam step with learning rate α: $\theta, \phi \leftarrow \text{AdamStep}(\theta, \phi, \nabla_{\theta,\phi} \mathcal{L}, \alpha)$
    end for
    Clear $\mathcal{D}$; log mean return this iteration.
  end for

5 Training Parameters

In the current code, I have used: K=250, T=200, N=10, E=100, γ=0.99, λ=1.0, ε=0.2, c_1=1.0, c_2=0.01, and Adam learning rate α=$3 \cdot 10^{-4}$.

6 Results

The code reports the **Total reward** (sum of rewards obtained in a single trajectory during the current iteration) and the **Average Score** (mean of total rewards over the last N trajectories collected in that iteration) for each iteration. Each figure is saved with parameterized filenames and a timestamp in the results folder.

7 Hyperparameters tuning

- In the results, we have varied the clipping range (which is 0.2 for now) as a parameter to the main code of PPO to see the variation. One can vary others to see how the results change.
- The result seems to be dependent on the beginning state and the first rewards. Sometimes it converges normally. Sometimes it takes longer to converge, but it finally converges. You can increase the iterations if you want to ensure convergence. It is set at 250 because it is the norm for the previous projects. Especially if the clipping range is small, we should wait longer to see convergence. You can also check the convergence by tracking the average score.

8 Environment and Dependencies

The code was developed and tested in a **Python 3.9** virtual environment, using the following package versions:

```
torch==2.0.1
numpy==1.23.5
gym==0.21.0
matplotlib==3.7.1
```

Additional Python packages required:

- `tqdm`
- `seaborn`

A full list of all packages can be found in the project folder with the name "requirements". All dependencies can be installed with:

```
pip install -r requirements.txt
```

References

[1] J. Schulman et al., *Proximal Policy Optimization Algorithms*, arXiv:1707.06347, 2017.

[2] J. Schulman et al., *High-Dimensional Continuous Control Using Generalized Advantage Estimation*, arXiv:1506.02438, 2016.