Finding Equilibrium of ToC Environment Using Independent Proximal Policy Optimization

Isaac Irwin

December 3, 2025

Contents

1 Overview

This project implements a multi-agent reinforcement learning model of the *Tragedy of the Commons* (TOC). A group of agents repeatedly harvest from a shared renewable resource. Each agent is individually rewarded for its own extraction, but over-exploitation collapses the resource and leads

to large penalties. The central question is whether independent learning agents can discover a sustainable equilibrium purely from reward feedback.

The implementation is structured around four main Python components:

- `CommonsEnv`: a custom environment encoding the common-pool resource dynamics and reward function.
- `ActorCritic`: a neural network mapping observations to both policy (action distribution) and value function.
- `PpoAgent`: an independent PPO (IPPO) agent using the actor–critic network and the PPO update rule.
- `Train.py`: the training script that coordinates multi-agent interaction with the environment and learning.

This document explains the mathematics behind each component, with particular focus on the PPO algorithm, the custom reward shaping, and the role of the hyperparameters currently used in the project.

2 Environment: `CommonsEnv`

2.1 State and Observation

Let $R_t \in [0, R_{\max}]$ denote the total amount of "resource" available at discrete time step t. The environment maintains:

$R_{\max} \in \mathbb{R}_{>0}$	maximum capacity
$R_t \in [0, R_{\max}]$	current resource level
$N \in \mathbb{N}$	number of agents
$t \in \{0, 1, \ldots, T_{\max}\}$	current step in the episode.

In the current setup, the observation for each agent is a scalar:

$$o_t = \frac{R_t}{R_{\max}} \in [0, 1].$$

All agents observe the same o_t; there is no explicit agent-specific state.

In code, this is implemented as

```
return np.array([self.resource / self.resource_max], dtype=np.float32)
```

so the observation dimension is `obs_dim = 1`.

2.2 Actions and Extraction

At each time step t, each agent $i \in \{1, \ldots, N\}$ chooses a scalar action $a_t^{(i)} \in [0, 1]$. This action is interpreted as a fraction of a per-agent extraction limit `max_extract`. The actual extracted amount $x_t^{(i)}$ is

$$x_t^{(i)} = a_t^{(i)} \cdot X_{\max},$$

where $X_{\max} =$ `max_extract`.

In the environment code:

```
actions = {i: float(np.clip(actions[i], 0, 1)) * self.max_extract
           for i in actions}
```

The total extraction at time t is

$$X_t = \sum_{i=1}^{N} x_t^{(i)}.$$

2.3 Resource Dynamics: Logistic Growth with Harvesting

The resource evolves according to a discrete-time logistic growth model with harvesting. Let $\alpha =$ `resource_regen_rate` denote the intrinsic growth rate. The update rule is

$$R_{t+1} \;=\; R_t + \alpha R_t \left(1 - \frac{R_t}{R_{\max}}\right) - X_t. \tag{1}$$

After this update, the resource is clipped to remain within physical bounds:

$$R_{t+1} \leftarrow \min\{\max\{R_{t+1}, 0\}, R_{\max}\}.$$

This ensures $R_{t+1} \in [0, R_{\max}]$.

2.4 Derivative-Based Reward Shaping

In addition to the base extraction reward, the environment computes the *change in resource* across one step:

$$\Delta R_t = R_{t+1} - R_t.$$

This is computed in code as

```
delta_R = self.resource - self.prev_resource
self.prev_resource = self.resource
```

with `self.prev_resource` storing R_t from the previous step.

If $\Delta R_t < 0$ (i.e., the resource is declining), a penalty is added to each agent's reward:

$$r_{t,\text{shaping}}^{(i)} = \beta\, \Delta R_t, \quad \text{for } \Delta R_t < 0,$$

where $\beta = 3.0$ in the current code:

```
if delta_R < 0:
    for i in rewards:
        rewards[i] += 3.0 * delta_R
```

Note that because $\Delta R_t < 0$, this term is negative and penalizes resource decline. The parameter β controls the strength of this *derivative-based shaping*. Larger β makes the agents more strongly avoid actions that lead to steep declines in the resource.

2.5 Episode Termination and Collapse Penalty

There are two ways an episode can terminate:

(i) **Collapse:** If the resource level falls below a critical threshold R_{crit}, the episode ends immediately. Currently, the code sets

$$R_{\text{crit}} = 1.0 \quad \text{and collapse occurs if } R_{t+1} \leq R_{\text{crit}}.$$

In code:

```
collapsed = self.resource <= 1.0
```

On collapse, all agents receive a large negative penalty:

$$r^{(i)}_{t,\text{collapse}} = -s_{\text{pen}} \cdot C,$$

where $C =$ `collapse_penalty` and s_{pen} is `penalty_scale`.
The environment sets the per-agent reward to this collapse penalty and returns with `done = True`.

(ii) **Horizon:** If the step counter t reaches `max_steps`, the episode ends due to horizon. In this case, a positive bonus is added to each agent's reward that scales with the remaining resource:

$$r^{(i)}_{T,\text{bonus}} = s_{\text{pen}} \cdot B \cdot \frac{R_T}{R_{\max}},$$

where $B =$ `scale_bonus`.

The collapse penalty and horizon bonus provide strong global shaping signals:

- Collapse is catastrophic and should be avoided.
- Arriving at the horizon with a high resource level is strongly rewarded.

2.6 Per-Time-Step Reward Function

Let us write the full per-step reward for agent i at time t (before episode termination logic) as

$$r^{(i)}_t = x^{(i)}_t + \mathbf{1}[\Delta R_t < 0] \cdot \beta \Delta R_t,$$

where $x^{(i)}_t$ is the extraction and ΔR_t is the resource change. At the final step, if the episode ended due to:

- **Collapse:**
$$r^{(i)}_t = -s_{\text{pen}} \cdot C.$$
- **Horizon:**
$$r^{(i)}_t \leftarrow r^{(i)}_t + s_{\text{pen}} \cdot B \cdot \frac{R_T}{R_{\max}}.$$

Here:

$$s_{\text{pen}} = \min\left(1, \frac{\text{episode\_index}}{\texttt{penalty\_ramp\_episodes}}\right)$$

ramps from 0 up to 1 during the early training episodes. This lets agents experience unpenalized over-exploitation early on and then slowly introduces the sustainability constraints.

2.7 Hyperparameters of the Environment

From `Train.py`, the environment is instantiated with:

$$\begin{aligned} R_{\max} &= 200, \\ \alpha &= 0.05, \\ X_{\max} &= 1.5, \\ T_{\max} &= 300, \\ C &= 90.0, \\ B &= 60, \\ N &= 5, \\ \texttt{penalty\_ramp\_episodes} &= 50. \end{aligned}$$

These values define a relatively harsh environment: agents can extract up to $NX_{\max} = 7.5$ units per step, while the maximal logistic growth near mid capacity is around $\alpha R_{\max}/4 = 2.5$. This makes over-exploitation easy and sustainability more subtle.

3 Actor–Critic Network: `ActorCritic`

The `ActorCritic` class is a neural network that maps observations to two outputs:

- A *policy head* that parameterizes a Gaussian distribution over actions.
- A *value head* that approximates the state-value function $V^\pi(s)$ under the current policy.

3.1 Network Architecture

Let the observation space be $\mathbb{R}^{d_o}$ (here $d_o = 1$). The network contains:

- A shared "body":

$$h_1 = \mathrm{ReLU}(W_1 o_t + b_1), \quad h_2 = \mathrm{ReLU}(W_2 h_1 + b_2),$$

  where $W_1 \in \mathbb{R}^{64\times d_o}$, $W_2 \in \mathbb{R}^{64\times 64}$.
- A *policy head*:

$$\mu_\theta(o_t) = W_\mu h_2 + b_\mu \in \mathbb{R},$$

  representing the mean of a 1-D Gaussian.
- A *value head*:

$$V_\theta(o_t) = W_v h_2 + b_v \in \mathbb{R},$$

  approximating $V^\pi(s_t)$.
- A learnable log standard deviation:

$$\log \sigma_\theta = \texttt{log\_std} \in \mathbb{R},$$

  shared across all states.

In code, the value head is implemented as:

```
self.v_head = nn.Linear(hidden, 1)
```

where `hidden = 64` is the dimension of h_2.

3.2 Stochastic Policy

For a given observation o_t, the actor–critic forward pass returns

$$(\mu_t, V_t, \log \sigma),$$

where $\mu_t = \mu_\theta(o_t)$, $V_t = V_\theta(o_t)$, and $\sigma = \exp(\log \sigma)$.

The policy is defined as a Gaussian in $\mathbb{R}$:

$$\tilde{a}_t \sim \mathcal{N}(\mu_t, \sigma^2).$$

This raw action $\tilde{a}_t$ is then squashed with a sigmoid to map into $[0, 1]$:

$$a_t = \sigma(\tilde{a}_t) = \frac{1}{1 + e^{-\tilde{a}_t}}.$$

This $a_t \in (0, 1)$ is precisely the *normalized* extraction fraction expected by the environment.

The log-probability used by PPO is computed in the *un-squashed* space:

$$\log \pi_\theta(\tilde{a}_t \mid o_t) = \log \mathcal{N}(\tilde{a}_t \mid \mu_t, \sigma^2).$$

This is a standard technique: we treat the Gaussian in raw action space as the policy for optimization, even though we pass the squashed action into the environment.

4 Independent PPO Agent: `PpoAgent`

4.1 Overview of PPO

Proximal Policy Optimization (PPO) maximizes a clipped policy gradient objective to provide stable updates. For a single agent, the objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\big)\Big],$$

where

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

is the importance sampling ratio and A_t is an estimate of the *advantage function*:

$$A_t \approx Q^\pi(s_t, a_t) - V^\pi(s_t).$$

The clipping parameter ϵ (here `clip_eps = 0.2`) constrains the size of policy updates by limiting $r_t(\theta)$ away from 1.

4.2 IPPO: One PPO per Agent

In this project, each agent has its own independent PPO instance (*IPPO*). There is no shared policy among agents; each `PpoAgent` has its own actor–critic network and optimizer. This allows agents to learn distinct strategies.

4.3 Trajectory Storage

During an episode, each agent i stores a trajectory:

$$\{(o_t^{(i)}, \tilde{a}_t^{(i)}, \log \pi_{\text{old}}(\tilde{a}_t^{(i)} \mid o_t^{(i)}), r_t^{(i)}, V_t^{(i)}, d_t)\}_{t=0}^{T-1},$$

where d_t is a done flag.

In `store()`:

```
self.obs_buf.append(obs)
self.act_raw_buf.append(raw_action)
self.logp_buf.append(logprob)
self.rew_buf.append(reward)
self.done_buf.append(done)
self.val_buf.append(value)
```

4.4 Return and Advantage Computation (GAE)

At the end of the episode, `finish_episode()` computes:

- Monte Carlo *returns*:

$$G_t = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k},$$

  where $\gamma = 0.99$.

- Generalized Advantage Estimation (GAE):

$$\delta_t = r_t + \gamma V_{t+1} - V_t,$$
$$\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1},$$

  with $\lambda = 0.95$.

In the code, this is implemented by iterating backwards through the episode:

```
for t in reversed(range(len(rew))):
    if done[t]:
        next_val = 0.0
        G = 0.0
        gae = 0.0

    delta = rew[t] + self.gamma * next_val - val[t]
    gae = delta + self.gamma * self.lam * gae
    G = rew[t] + self.gamma * G
    next_val = val[t]

    adv.insert(0, gae)
    returns.insert(0, G)
```

After this loop:

$$\hat{A}_t = \texttt{adv[t]}, \quad G_t = \texttt{returns[t]}.$$

Advantages are then normalized:

$$\hat{A}_t \leftarrow \frac{\hat{A}_t - \mu_A}{\sigma_A + 10^{-8}},$$

where μ_A and σ_A are the mean and standard deviation of the advantages within the episode. This centers and scales the gradient signal, improving stability.

4.5 PPO Update

The agent then runs `train_iters` optimization epochs over the episode data. For each epoch:

1. Recompute policy $\pi_\theta(\cdot \mid o_t)$ and value $V_\theta(o_t)$:

$$\mu_t, V_t^{\text{pred}}, \log \sigma = \texttt{self.ac(obs)}.$$

2. Construct the Gaussian distribution:

$$\tilde{a}_t \sim \mathcal{N}(\mu_t, \sigma^2).$$

3. Compute new log-probabilities:

$$\log \pi_\theta(\tilde{a}_t \mid o_t).$$

4. Compute the PPO ratio:

$$r_t(\theta) = \exp\left(\log \pi_\theta(\tilde{a}_t \mid o_t) - \log \pi_{\text{old}}(\tilde{a}_t \mid o_t)\right).$$

5. Compute the clipped surrogate loss:

$$L_{\text{actor}}(\theta) = -\mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_t\right)\right].$$

6. Compute the critic loss:

$$L_{\text{critic}}(\theta) = \mathbb{E}_t\left[(V_\theta(o_t) - G_t)^2\right].$$

7. Minimize the combined loss

$$L(\theta) = L_{\text{actor}}(\theta) + \frac{1}{2}L_{\text{critic}}(\theta).$$

In code:

```
unclipped = ratio * adv
clipped = torch.clamp(ratio, 1 - self.clip_eps,
                      1 + self.clip_eps) * adv
actor_loss = -torch.mean(torch.min(unclipped, clipped))
critic_loss = F.mse_loss(v_pred, returns)
loss = actor_loss + 0.5 * critic_loss
```

Gradients are clipped to norm 0.5:

```
nn.utils.clip_grad_norm_(self.ac.parameters(), 0.5)
```

which stabilizes training by preventing excessively large updates.

4.6 PPO Hyperparameters

From `Train.py`, each `PpoAgent` is created with:

$$\begin{aligned} \text{learning rate}: \ & \eta = 3 \times 10^{-4}, \\ \gamma: \ & 0.99, \\ \lambda: \ & 0.95, \\ \epsilon: \ & 0.2 \quad (\texttt{clip\_eps}), \\ \text{train\_iters}: \ & 10. \end{aligned}$$

- A smaller learning rate η encourages more stable but slower updates.
- $\gamma = 0.99$ emphasizes long-term return.
- $\lambda = 0.95$ in GAE is a standard setting balancing bias and variance.
- The clipping parameter $\epsilon = 0.2$ limits how much the new policy can deviate from the behavior policy in a single update.
- `train_iters = 10` allows multiple passes over the episode data, as in standard PPO.

5 Training Loop: `Train.py`

5.1 Environment and Agents Instantiation

The PPO training function constructs the environment and a set of independent agents:

$$\begin{aligned} N &= 5 \quad \text{agents}, \\ R_{\max} &= 200, \\ \alpha &= 0.05, \\ X_{\max} &= 1.5, \\ T_{\max} &= 300. \end{aligned}$$

Formally:

$$\texttt{env} = \text{CommonsEnv}(N, R_{\max}, \alpha, X_{\max}, T_{\max}, \dots),$$

and for each $i = 1, \dots, N$:

$$\texttt{agent}_i = \text{PpoAgent}(d_o = 1, \eta, \gamma, \lambda, \epsilon, \texttt{train\_iters}).$$

5.2 Episode Rollout

For each episode index $e = 0, \dots, E-1$ (with $E = 200$):

1. Reset the environment:
$$o_0^{(i)} \leftarrow \texttt{env.reset}(e),$$
which also updates `penalty_scale` depending on e.
2. For each time step t in the episode until `done`:
   (a) Each agent i chooses an action:
$$a_t^{(i)}, \tilde{a}_t^{(i)}, \log \pi_t^{(i)}, V_t^{(i)} = \texttt{agent}_i.\text{act}(o_t^{(i)}).$$
   (b) The environment takes the joint action:
$$o_{t+1}, \{r_t^{(i)}\}, \texttt{done} = \texttt{env.step}(\{a_t^{(i)}\}).$$

(c) Each agent stores its transition:

$$\texttt{agent}_i.\text{store}(o_t^{(i)}, \tilde{a}_t^{(i)}, \log \pi_t^{(i)}, r_t^{(i)}, V_t^{(i)}, \texttt{done}).$$

3. After the episode ends, each agent runs PPO:

$$\texttt{agent}_i.\text{finish\_episode}().$$

Total reward per episode and resource level at the end of each episode are logged and later plotted. These plots show:

- Early episodes: frequent collapse and very negative rewards.
- After penalties ramp in: agents gradually learn to limit extraction.
- Eventual stabilization: resource levels remain high (near 0.8–0.9 of $R_{\max}$) and rewards become less negative or mildly positive.

This progression reflects the agents discovering a sustainable extraction equilibrium under the complex reward shaping.

6 Summary

Mathematically, this project couples:

- A logistic common-pool resource model with harvesting,
- A derivative-based reward penalty for resource decline,
- Collapse and horizon shaping penalties and bonuses,
- Independent PPO agents with Gaussian policies and GAE,
- A multi-agent training loop in which agents learn purely from local rewards.

The resulting system demonstrates an emergent transition from over-exploiting the resource (collapse) to a near-equilibrium regime where agents extract aggressively but not so much as to trigger collapse. The detailed hyperparameter choices—especially `resource_regen_rate`, `max_extract`, `collapse_penalty`, `scale_bonus`, and the GAE/PPO parameters γ, λ, and ϵ strongly influence how quickly and how robustly this equilibrium is learned.