Week 1 Briefing

Making robots learn by trial and error

Jack Cashman

July 5, 2024

Plan for presentation

1. Background Knowledge
   - Proximal Policy Optimisation (PPO)
   - Twin Delayed Deep Deterministic Policy Gradient (TD3)

2. Implementation
   - Scope
   - Results

Motivating PPO: Introduction to Policy Optimisation

Let $\pi_{\boldsymbol{\theta}}$ be a policy, where $\boldsymbol{\theta}$ is a real-valued parameter vector. Formally, $\pi_{\boldsymbol{\theta}}$ is a probability distribution over the actions, conditioned on the current state.

Furthermore, let $J(\pi_{\boldsymbol{\theta}})$ be a function that we wish to *maximise*. J should tell us how 'good' $\pi_{\boldsymbol{\theta}}$ is. (e.g. expected reward)

In VPG, we wish to make updates $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}_k}$ until convergence.

In practice, VPG can perform poorly. If α is too small, convergence is painfully slow. Moreover, even a single bad step can collapse the performance of $\pi_{\boldsymbol{\theta}}$

Motivating PPO: Introduction to Policy Optimisation

To address this, more complicated methods have been developed.

Trust Region Policy Optimisation (TRPO) includes a hard constraint in terms of the KL divergence. We then solve an approximate second order optimisation problem.

Proximal Policy Optimision (PPO) is a family of first-order methods. Instead of adding a hard constraint, PPO methods penalise 'large steps' in the objective function itself.

PPO is significantly simpler than TRPO, and empirically seems to perform at least as well. Of the PPO family, I'm focusing on the PPO-Clip algorithm.

Proximal Policy Optimisation (PPO)

PPO-Clip updates policies via:

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{s,a\sim\pi_{\boldsymbol{\theta}_k}} \left[L(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) \right]$$

Where we have that:

$$L(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) = \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta_k}}(a|s)} A^{\pi_{\boldsymbol{\theta}_k}}(s, a)), g(\varepsilon, A^{\pi_{\boldsymbol{\theta}_k}}(s, a)) \right\}$$

With:

$$g(\varepsilon, A) = \begin{cases} (1+\varepsilon)A & A \geq 0 \\ (1-\varepsilon)A & A < 0 \end{cases}$$

The PPO-clip Algorithm

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Motivating TD3: An introduction to DDPG

In the famous Q-learning, once we have a strong estimation of the Q-function, the optimal action is given by $a^*(s) = \max_{a \in \mathcal{A}} Q(s, a)$. If $\mathcal{A}$ is infinite, this max is highly non-trivial.

Instead of computing the max, we learn a function approximator $\mu(s)$ such that $\max_a Q(s, a) \approx Q(s, \mu(s))$. That is, $\mu(s)$ approximates the optimal action. We need another function approximator to estimate the Q-function.

Due to time constraints, I skip some details in this description, but this is essentially Deep Deterministic Policy Gradient (DDPG)

TD3 (Extending DDPG)

Empirically, DDPG often begins to dramatically overestimate Q-values, which leads to policy collapse. To fix, this TD3:

- learns two Q-functions instead of just one, and uses the smaller one in the update.
- updates the policy (and target networks) less frequently than the Q-function.
- add Gaussian noise to the action to regularise the function approximators.

TD3

Algorithm 1 Twin Delayed DDPG

1: Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: **repeat**
4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** j in range(however many updates) **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

13: Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

14: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

15: **if** j mod `policy_delay` $= 0$ **then**
16: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

17: Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

18: **end if**
19: **end for**
20: **end if**
21: **until** convergence

The target on line 13 follows from Bellman's equation, and the update on line 14 is simply the MSBE.

Implementation Scope

1. I've only been granted access to Bunya yesterday, so the following testing was conducted on my laptop.
2. Given this reduced compute power, I've opted to train agents using PPO and TD3 on 3 separate `mujoco` environments using `gymnasium`. Each agent was trained on 1 million time steps.
3. The environments I used were `HalfCheetah-v4`, `Hopper-v4`, and `Walker2d-v4`.
4. All implementation is done using `cleanrl` and logging via `tensorboard`.

Implementation Scope

1. I've only been granted access to Bunya yesterday, so the following testing was conducted on my laptop.
2. Given this reduced compute power, I've opted to train agents using PPO and TD3 on 3 separate `mujoco` environments using `gymnasium`. Each agent was trained on 1 million time steps.
3. The environments I used were `HalfCheetah-v4`, `Hopper-v4`, and `Walker2d-v4`.
4. All implementation is done using `cleanrl` and logging via `tensorboard`.

Implementation Scope

1. I've only been granted access to Bunya yesterday, so the following testing was conducted on my laptop.
2. Given this reduced compute power, I've opted to train agents using PPO and TD3 on 3 separate `mujoco` environments using `gymnasium`. Each agent was trained on 1 million time steps.
3. The environments I used were `HalfCheetah-v4`, `Hopper-v4`, and `Walker2d-v4`.
4. All implementation is done using `cleanrl` and logging via `tensorboard`.

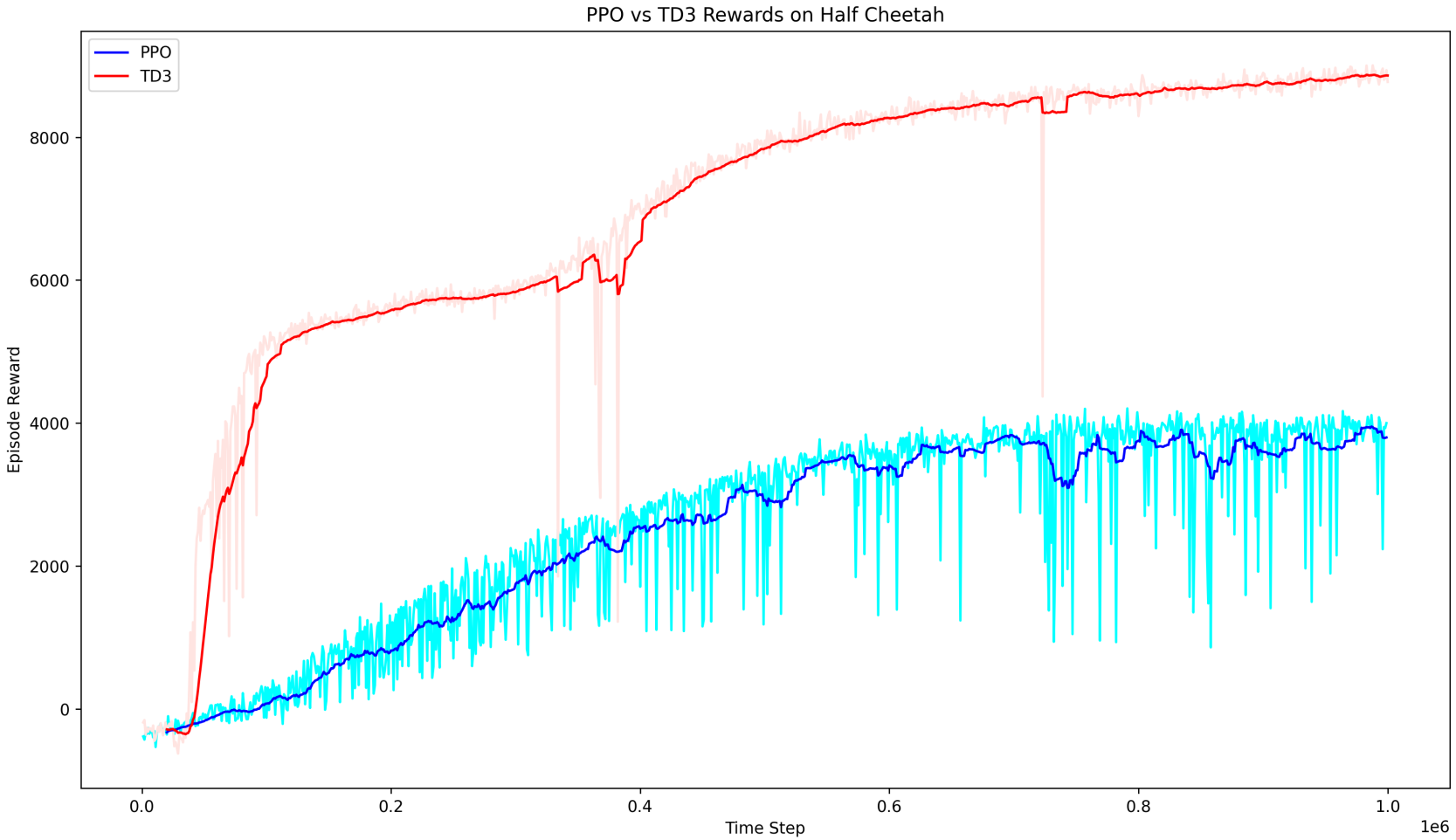
Implementation Scope

1. I've only been granted access to Bunya yesterday, so the following testing was conducted on my laptop.
2. Given this reduced compute power, I've opted to train agents using PPO and TD3 on 3 separate `mujoco` environments using `gymnasium`. Each agent was trained on 1 million time steps.
3. The environments I used were `HalfCheetah-v4`, `Hopper-v4`, and `Walker2d-v4`.
4. All implementation is done using `cleanrl` and logging via `tensorboard`.

Results - Half Cheetah

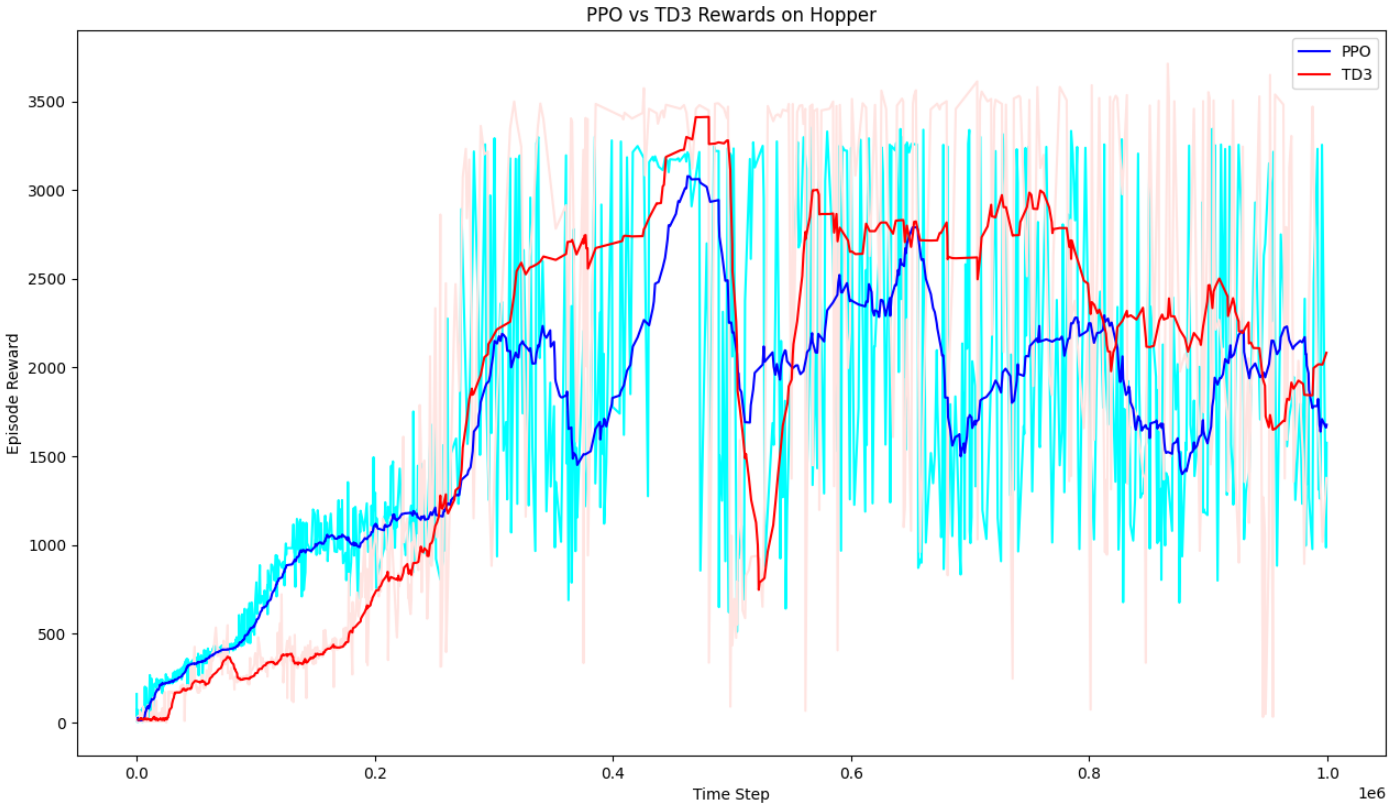
Compute Time: PPO 22.71 minutes, TD3 2.403 hours

Reward: PPO 3786, TD3 8855



Results - Hopper

Compute Time: PPO 19.64 minutes, TD3 2.341 hours
Reward: PPO 1784, TD3 1978



Results - Walker 2d

Compute Time: PPO 19.28 minutes, TD3 3.265 hours
Reward: PPO 3549, TD3 4552