TD3 with Noisy Networks for Hopper-v4

Ishanya Sharma

Contents

1 Project Goal **2**

2 Core TD3 Implementation **2**

3 Noisy Networks Integration **3**

4 Experiments and Findings **3**

5 Precautions: **4**

1 Project Goal

The goal of this project was to implement the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm and enhance it using Noisy Networks to improve exploration strategies, specifically in the `Hopper-v4` environment from gym lib. The aim of this proj was to investigate whether learned noise parameters could offer a better alternative to traditional Gaussian noise for continuous control tasks.

2 Core TD3 Implementation

What Was Done

The TD3 algorithm was implemented using PyTorch and gym library. The major components included:

- **Actor Network:** A multi-layer perceptron (MLP) mapping state inputs to actions, scaled by `max_action` through a `tanh` activation.
- **Critic Network:** Two separate MLPs (Twin Critics) that take state-action pairs as input and output Q-values.
- **Target Networks:** Polyak-averaged (slow-updating) versions of the actor and critic networks, updated using a factor τ.
- **Replay Buffer:** Stores transitions $(s, a, r, s', \text{done})$ to allow decorrelated mini-batch training.
- **Optimizers:** Adam optimizers were used for training both actor and critic.

Key TD3 Logic

- **Critic Update:** The target Q-value is computed as:

  $$Q_{\text{target}} = r + \gamma \cdot \min(Q'_1, Q'_2)$$

  where Q'_1 and Q'_2 are the twin critic target outputs using the next state and action from the target actor.
- **Actor Update:** Done less frequently using delayed updates. The actor is optimized to maximize $Q_1(s, \pi(s))$.
- **Target Policy Smoothing:** Explicitly excluded as per assignment instructions.
- **Exploration Strategy:** Gaussian noise was added to the actor's action outputs during training.

3 Noisy Networks Integration

What Was Done

Standard exploration noise was replaced by parameterized noise within the actor network.

Concept

Instead of standard linear layers:

$$y = wx + b,$$

we used:

$$y = (\mu^w + \sigma^w \cdot \epsilon^w)x + (\mu^b + \sigma^b \cdot \epsilon^b),$$

where μ are learnable parameters, σ are learnable noise magnitudes, and ϵ are samples from a fixed distribution (usually Gaussian).

Implementation Details

- A custom `NoisyLinear` PyTorch layer was implemented.
- Replaced standard `nn.Linear` in the actor with `NoisyLinear`.
- Modified the TD3 agent to call `actor.sample_noise()` before acting.
- Removed standard Gaussian action noise by setting `EXPLORATION_NOISE = 0`.

Why This Was Done

Noisy Networks allow the model to learn how much exploration is needed in a state-dependent manner, potentially discovering more effective exploration strategies than static Gaussian noise.

4 Experiments and Findings

TD3 vs. TD3 + Noisy Networks

- **Learning Speed:** TD3+NoisyNets showed faster early learning due to more effective exploration.
- **Final Performance:** TD3+NoisyNets achieved better final reward averages, indicating better policy discovery.
- **Conclusion:** Learning exploration noise leads to more sample-efficient and robust learning compared to static noise injection.

Other Noise Distributions

- Tried fixed Beta and Gamma distributions for ϵ, but they did not outperform standard learned Gaussian noise.
- Key insight: Learning the noise magnitude σ is more important than the choice of base noise distribution.

5 Precautions:

- **Hyperparameters:** Parameters such as learning rate, τ, network architecture, and policy frequency (`POLICY_FREQ`) require careful tuning.
- **Adversarial Robustness (Conceptual):** While Noisy Nets may smooth outputs, they are not robust to adversarial attacks unless explicitly trained for robustness.