Continuous Control of Cart-Pole System using Deep Deterministic Policy Gradient (DDPG)

Aman Chandak
Dept. of Mechanical and Aerospace Engineering
Arizona State University
achan160@asu.edu

***Abstract*—This report presents the implementation and evaluation of the Deep Deterministic Policy Gradient (DDPG) algorithm to solve the *cartpole:balance* task within the DeepMind Control Suite. DDPG, an actor-critic method tailored for continuous action spaces, was implemented and trained over multiple seeds (0, 1, 2). The agent successfully learned to balance the pole, achieving near-optimal rewards within 80 episodes. This report discusses the implementation details, the choice of hyperparameters, and analyzes the properties of the testing environment.**

***Index Terms*—Reinforcement Learning, DDPG, Continuous Control, Robotics, DeepMind Control Suite.**

I. Introduction

The Cart-pole problem is a canonical benchmark in control theory and reinforcement learning (RL). The objective is to balance an unactuated pole by applying forces to a cart moving along a track. DeepMind Control Suite provides the physics-based simulations for Reinforcement learning tasks for continuous control environments.

To balance a pole upright in a moving cart, I employed Deep Deterministic Policy Gradient (DDPG) [2], a model-free, off-policy actor-critic algorithm. DDPG combines the stability of Deep Q-Networks (DQN) with the ability to handle continuous action spaces using a deterministic policy gradient.

II. Methodology and Implementation

A. Algorithm Overview

DDPG maintains two neural networks: an Actor $\mu_\theta(s)$ which specifies the current policy, which is deterministic and only depends on the current state, and a Critic $Q_w(s, \mu_\theta(s))$ which approximates the Bellman equation. To ensure stability, DDPG employs two key mechanisms:

- **Replay Buffer:** Breaking temporal correlations by sampling random mini-batches of transitions $(s, a, r, s', done)$.
- **Target Networks:** Slow-tracking target networks, one for Actor $\mu_{\theta^-}(s')$ and another for critic $Q_{w^-}(s', \mu_{\theta^-}(s'))$, that are used to compute target values, updated via "soft updates" controlled by a parameter $\tau \ll 1$.

The critic, $Q_w(s, \mu_\theta(s))$, is trained to minimize the mean squared error loss:

$$L = \frac{1}{N} \sum (y_i - Q_w(s, \mu_\theta(s)))^2 \tag{1}$$

where $y_i = r_i + \gamma Q_{w^-}(s', \mu_{\theta^-}(s'))$. The actor is trained to maximize the estimated Q-value via policy gradient update:

$$\nabla J = \left[\nabla_a Q_w(s, \mu_\theta(s)). \nabla_\theta \mu_\theta(s) \right] \tag{2}$$

B. Architecture

The implementation uses the following architecture:

- **Input:** A flattened observation vector containing position and velocity of the cart and pole.
- **Actor/Target actor:** A Multi-Layer Perceptron (MLP) with two hidden layers (400, 300 units) utilizing ReLU activations. The output layer uses a $tanh$ activation scaled to the action limits.
- **Critic/Target critic:** Similar architecture (400, 300 units), but takes both state and action as inputs in the first layer.

C. Exploration Strategy

Exploration in continuous spaces is achieved by adding noise to the actor's policy. While the original paper suggests Ornstein-Uhlenbeck (OU) noise, I used Gaussian noise instead. For the final results, uncorrelated Gaussian noise $\mathcal{N}(0, 0.2)$ was selected for its simplicity and effective performance on the balance task.

D. Code Availability

The complete implementation, including training loops and evaluation scripts, is available at: Google Colab notebook.

III. Experimental Setup

A. Hyperparameters

The hyperparameters were tuned based on standard literature values and adjusted for the specific dynamics of the Cart-pole task. The final configuration is detailed in Table I.

B. Training Procedure

The policy was trained on three distinct random seeds (0, 1, 2) to verify reproducibility. Each seed was trained for 80 episodes because when tuning with the episodic length, it came out that the agent is able to learn effectively in 80 episodes, and the three seeds can be trained in under 25 minutes. The evaluation was performed separately on Seed 10, with exploration noise disabled, and then also tested with a different random seed 3 for checking the angle of the pole in another single episode of 1000 timesteps.

TABLE I
HYPERPARAMETER CONFIGURATION

Parameter	Value
Actor Learning Rate	1×10^{-4}
Critic Learning Rate	1×10^{-3}
Discount Factor (γ)	0.99
Batch Size	64
Episode length	80
Replay Buffer Size	100,000
Soft Update (τ)	0.005
Noise Sigma (σ)	0.2
Hidden Layers	(400, 300)

IV. RESULTS

A. Learning Curves

Figure 1 illustrates the average reward over episodes across the three seeds. The shaded region represents the standard deviation. The agent consistently learns to stabilize the pole within the first 40 episodes, reaching a near-maximum reward, which is 1000, as the reward can be 1 for each timestep, and one episode has 1000 timesteps.

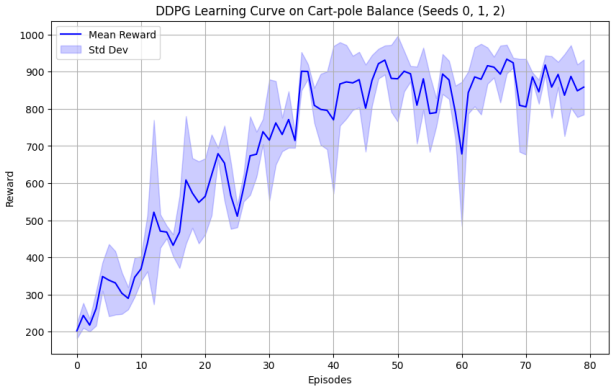

Fig. 1. Average Learning Curve over 3 seeds (Mean $\pm$ Std).

B. Evaluation

Upon evaluating the trained agent on Seed 10, the policy demonstrated robust stability, maintaining the pole upright for the the whole episode, Fig. 2.

The agent demonstrates near-perfect stability with a steady-state error of less than $2\%(\cos\theta > 0.9)$, Fig. 3. The consistent low-amplitude oscillation indicates a stable limit cycle where the policy effectively performs micro-corrections to maintain equilibrium.

V. DISCUSSION ON HYPERPARAMETERS

The performance of DDPG is highly sensitive to hyper-parameter choices. I observed the following effects during tuning:

1) Learning Rates: A disparity between Actor and Critic learning rates was crucial. Setting the Critic LR (10^{-3}) an order of magnitude higher than the Actor LR (10^{-4}) provided the most stable convergence. When rates were equal, the Actor often updated too quickly based on inaccurate Q-values, leading to policy divergence.

2) Noise Scale: The magnitude of exploration noise (σ) significantly impacted convergence speed. A high $\sigma(> 0.3)$ caused the agent to constantly destabilize the pole even after learning the correct policy. A low $\sigma(< 0.1)$ resulted in the agent getting stuck in local optima, failing to discover the swing-up motion required if the pole fell. $\sigma = 0.2$ offered the best balance.

3) Soft Update (τ): I have used a slow target update ($\tau = 0.005$). Increasing this value resulted in oscillating Q-values, as the target networks chased the noisy main networks too closely.

Fig. 2. Instantaneous reward [0,1] for each timestep.

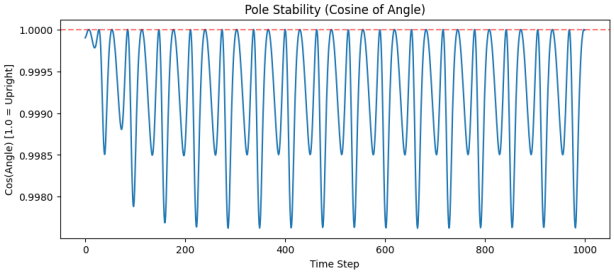

Fig. 3. Stability of pole with cos of angle from upright position.

REFERENCES

[1] S. Tunyasuvunakool et al., "dm_control: Software and tasks for continuous control," *Software Impacts*, vol. 6, 2020.

[2] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.