Implementation and Analysis of Dueling DDQN and PPO in Classic RL Environments

Sun Yuxiang
Student ID: 522020910073

June 2025

1 Introduction

Reinforcement Learning (RL) is a widely studied branch of machine learning that deals with how agents can learn to make sequences of decisions by interacting with an environment to maximize cumulative rewards. Generally, RL algorithms can be categorized into two main paradigms: value-based and policy-based methods.

Value-based methods, such as Deep Q-Networks (DQN) and its variants, aim to learn a value function that estimates the expected return for each state-action pair. The policy is derived implicitly by acting greedily with respect to this value function. This approach is typically simple to implement and efficient in discrete action spaces, but struggles with high-dimensional or continuous action spaces due to the need for discrete action selection.

In contrast, policy-based methods, including Policy Gradient algorithms and Actor-Critic variants like Proximal Policy Optimization (PPO), learn a parameterized policy directly. These methods are well-suited for problems with continuous or high-dimensional action spaces and can learn stochastic policies. However, they tend to suffer from high variance and typically converge to a local rather than global optimum.

In this project, we implement and evaluate both paradigms. For the value-based method, we adopt the Dueling Double DQN(Dueling DDQN) architecture and test it in the Atari `Pong-v5` environment with discrete actions. For the policy-based approach, we implement PPO and test it in three continuous control tasks: `Hopper-v4`, `Ant-v4`, and `HalfCheetah-v4`. Our goal is to enable the agent to have good performance in each environment and to analyze the strengths and limitations of both methods. Experimental results show that both algorithms achieve reasonable performance, demonstrating the effectiveness of modern RL algorithms in diverse settings.

2 Background

This section provides mathematical background for DQN, Dueling DDQN, Actor-Critic methods and PPO.

2.1 Dueling DDQN

DQN combine Q-learning with deep neural networks to approximate the action-value function $Q(s, a)$ for environments with large or continuous state spaces. The network is trained to minimize the squared temporal difference (TD) error:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2\right],$$

where D denotes the distribution of transitions drawn from a replay buffer, and θ^- represents the parameters of a fixed target network that is updated periodically to stabilize training.

However, using $\max_{a'} Q$ in the target introduces an overestimation bias due to the maximization over noisy value estimates. Double DQN (DDQN) addresses this by decoupling action selection and evaluation:

$$y_t = r + \gamma Q'(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-),$$

where the online network Q selects the action and the target network Q' evaluates it. This modification helps provide more accurate value estimates.

The Dueling DQN architecture further improves upon DQN by separately estimating the state-value function $V(s)$ and the advantage function $A(s,a)$, which are then combined to compute the final action-value:

$$Q(s,a;\theta) = V(f(s;\theta_{share}),\theta_V) + A(f(s;\theta_{share}),a;\theta_A) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(f(s;\theta_{share}),a';\theta_A),$$

where $f(\cdot)$ denotes the shared feature encoder and $\theta_{share}, \theta_V, \theta_A$ are the parameters of the shared encoder, value stream, and advantage stream, respectively. This architecture helps the agent learn which states are (or are not) valuable regardless of the action taken.

Combining the Double DQN update rule with the Dueling network leads to the Dueling DDQN algorithm, which optimizes the following objective:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\Big[(r + \gamma Q'(s', b^*(s'); \theta^-) - Q(s,a;\theta))^2\Big], \quad b^*(s') = \arg\max_{a'} Q(s',a';\theta).$$

These enhancements together provide better value estimation stability and improved learning efficiency, especially in environments with large state spaces or where the advantage function varies significantly across actions.

2.2 PPO

Policy gradient methods directly optimize a parameterized stochastic policy $\pi_\theta(a|s)$ to maximize the expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big].$$

Using the policy gradient theorem, the gradient can be estimated as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s) A^\pi(s,a)\Big], \quad A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

In practice, computing the advantage function $A^\pi(s,a)$ accurately is crucial. Generalized Advantage Estimation (GAE) is often used to trade off bias and variance by exponentially weighting multi-step TD errors.

Actor-Critic methods combine this policy gradient idea with value-function learning. The actor updates the policy π_θ, while the critic estimates the state-value function $V^\pi(s)$ to provide a low-variance advantage estimation. This hybrid framework enables more stable and sample-efficient training.

PPO further improves the stability and sample-efficiency by reusing collected samples more effectively. In standard policy gradient methods, once the policy π_θ is updated, the old data must be discarded. PPO observes that the true objective depends on trajectories sampled from the current policy, but in practice we have samples from the old policy $\pi_{\theta'}$. This mismatch can be corrected using importance sampling:

$$\mathbb{E}_{\pi_\theta}[\cdot] \approx \mathbb{E}_{\pi_{\theta'}}\Big[\frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)} \cdot\cdot\Big].$$

To avoid high variance from importance weights, Trust Region Policy Optimization (TRPO) constrains the policy update using KL divergence. PPO simplifies this by clipping the probability ratio, leading to the final surrogate objective:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}\Big[\min\big(r_t A_t,\ \text{clip}(r_t, 1-\epsilon, 1+\epsilon)A_t\big)\Big], \quad r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}.$$

This clipping prevents the new policy from deviating too far from the old policy, allowing multiple epochs of update while ensuring stability.

Additionally, an entropy bonus term $H[\pi_\theta]$ is added to encourage exploration:

$$L^{\text{PPO}} = L^{\text{CLIP}}(\theta) + \beta H[\pi_\theta],$$

where β controls the strength of the entropy regularization.

3 Experiment Setup

This section briefly introduces the environments used and summarizes key implementation details for our algorithms.

3.1 Environment Overview

In this project, we evaluate our algorithms on four standard reinforcement learning benchmarks:

- **Pong-v5**: A classic Atari game with a discrete action space of size 6. The agent observes a stack of frames (typically $210 \times 160 \times 3$ RGB images) and aims to maximize its score by bouncing the ball past the opponent's paddle.

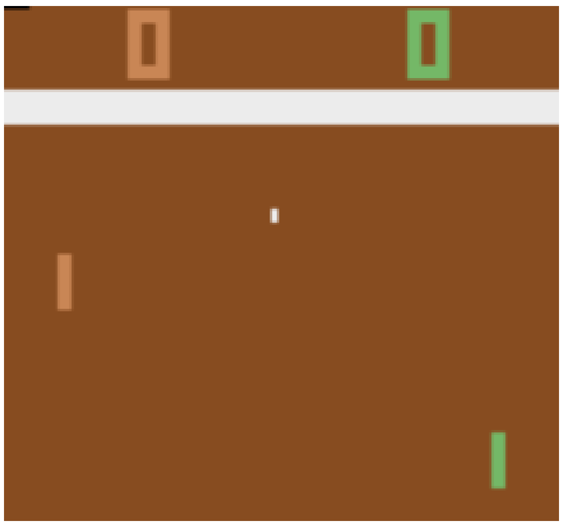


Figure 1: Pong-v5: A discrete-action Atari benchmark for testing value-based methods.

- **Hopper-v4**: A continuous control MuJoCo task where a 2D one-legged robot must learn to hop forward as fast as possible without falling over. The observation space is a 11-dimensional vector describing position and velocity, and the action space is 3-dimensional, representing torques applied at the joints.
- **HalfCheetah-v4**: Another MuJoCo benchmark where a planar two-legged agent (cheetah) must run forward efficiently. The observation space has 17 dimensions, including body position and velocity, and the action space is 6-dimensional for the hip and knee joints.
- **Ant-v4**: A more complex continuous control task featuring a 3D quadruped robot that must learn to walk and maintain balance. The observation space has 105 dimensions (positions, velocities, external forces), and the action space is 8-dimensional, representing torques at each joint.

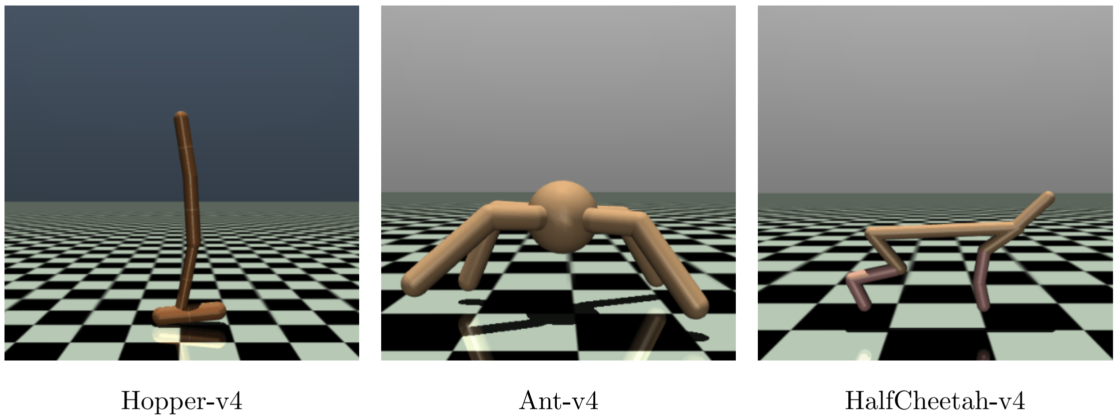


Figure 2: Continuous control MuJoCo tasks used for evaluating policy-based methods.

3.2 Implementation Details

Dueling DDQN. For the discrete Pong-v5 environment, we use Dueling DDQN. As Pong-v5 is relatively simple, studies show that Dueling DDQN alone can perform nearly as well as more complex methods like Rainbow. Before training, raw game frames are preprocessed by grayscaling, resizing, and stacking to construct the input states. The training follows this simplified logic:

Algorithm 1 Dueling DDQN

```
1: Initialize Q-network $Q$ with parameters $\theta$ and target network $Q'$ with $\theta^- \leftarrow \theta$.
2: Initialize replay buffer $D$.
3: for each step do
4:     Select action $a$ using $\epsilon$-greedy policy.
5:     Execute $a$, observe reward $r$ and next state $s'$.
6:     Store $(s, a, r, s', done)$ in $D$.
7:     if buffer size > mini_size then
8:         Sample mini-batch from $D$.
9:         Compute Double DQN target: $y = r + \gamma Q'(s', \arg\max_a Q(s', a; \theta); \theta^-)$.
10:        Minimize loss: $L(\theta) = (y - Q(s, a; \theta))^2$.
11:    end if
12:    Periodically update target network: $\theta^- \leftarrow \theta$.
13: end for
```

Network Architecture. The network consists of a shared convolutional encoder and two separate streams for value and advantage estimation, as illustrated in Figure 3. The shared feature extractor contains three convolutional layers with ReLU activations:

- Conv2D: out_channels = 32, kernel_size = 8, stride = 4
- Conv2D: out_channels = 64, kernel_size = 4, stride = 2
- Conv2D: out_channels = 64, kernel_size = 3, stride = 1

The output is then flattened and fed into two separate fully connected branches:

- **Value stream**: hidden layer with 512 units and ReLU activation, followed by a linear output producing the scalar $V(s)$.
- **Advantage stream**: same hidden structure but the output dimension matches the size of the action space $|\mathcal{A}|$.

This shared-convolution + split-stream design allows the network to learn both the state value and the relative advantage for each action, which are then combined as:

$$Q(s, a; \theta) = V(f(s; \theta_{share}), \theta_V) + A(f(s; \theta_{share}), a; \theta_A) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(f(s; \theta_{share}), a'; \theta_A).$$

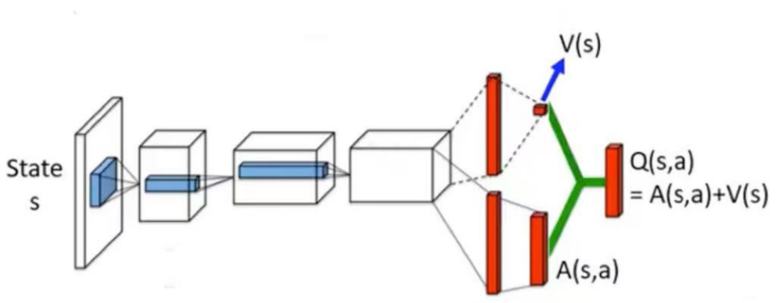


Figure 3: Dueling Double DQN network architecture with shared convolutional layers and separate value and advantage streams.

Hyperparameters. We set the learning rate to $1e^{-4}$, $\gamma = 0.99$, replay buffer size to 50,000, mini_size threshold to 10,000, mini-batch size to 64, target network updates every 1,000 steps, and total training frames to 2,000,000. The ϵ-greedy exploration decays from 0.9 to 0.01 over 50,000 steps.

PPO. For continuous control tasks Hopper-v4, Ant-v4, and HalfCheetah-v4, we implement PPO with Generalized Advantage Estimation (GAE) and running mean-variance normalization. The training loop is:

Algorithm 2 PPO

```
1: Initialize actor $\pi(\theta)$ and critic $V(\phi)$.
2: for each iteration do
3:     Normalize states with running mean and variance.
4:     Collect trajectories by sampling actions from $\pi(\theta)$.
5:     Compute returns and advantages using GAE.
6:     Use collected trajectories to update the networks:
7:     for each epoch do
8:         Compute probability ratio: $r(\theta) = \frac{\pi_\theta}{\pi_{\theta_{old}}}$.
9:         Compute clipped surrogate loss:
               $$L^{\text{CLIP}}(\theta) = \mathbb{E}\Big[\min\Big(r(\theta)A,\ \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A\Big)\Big].$$
10:        Update actor and critic parameters.
11:    end for
12:    Update running statistics.
13: end for
```

Hyperparameters. The table below summarizes the PPO hyperparameters for different MuJoCo tasks:

Parameter	**Hopper-v4**	**Ant-v4**	**HalfCheetah-v4**
Hidden Dimension	256	256	256
Actor/Critic LR	3e-4	3e-4	3e-4
Entropy Coefficient	1e-3	1e-3	1e-3
Gamma (γ)	0.99	0.99	0.99
GAE Lambda (λ)	0.95	0.95	0.95
Clip Range (ϵ)	0.2	0.2	0.2
Trajectory Length	1024	2048	2048
Batch Size	64	128	128
Epochs per Update	8	10	10
Max Grad Norm	0.5	0.5	0.5
Num Episodes	1500	2000	2000

Table 1: PPO hyperparameters for continuous control tasks.

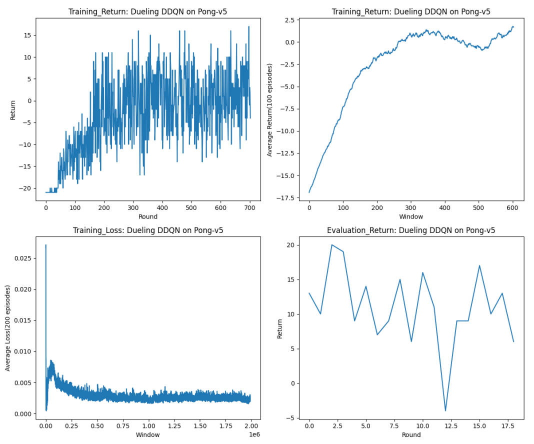
4 Experiment Results and Analysis

In this section, we present and analyze the training results of our Dueling DDQN and PPO implementations across the selected environments, highlighting performance trends, challenges, and key observations.

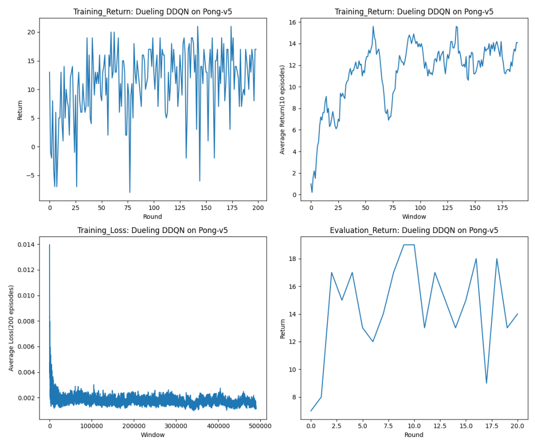
4.1 Dueling DDQN on Pong-v5

Results. In the first stage, we trained the Dueling DDQN agent on Pong-v5 for 2 million frames, during which the ϵ decayed from 0.9 to 0.1. Figure 4 shows that overall performance improved, but there was a noticeable performance drop between 1.2M and 1.8M frames. When evaluated, the agent could reach a maximum score of 20 and an average score around 11, but the variance was very large and even included negative scores.

To further improve stability and performance, we continued training for an additional 0.5 million frames, decaying ϵ from 0.1 to 0.01. The results show that the agent's performance fluctuated but overall trended upward. In the final evaluation, the best score slightly decreased to 19, but the average score increased to about 15, and the variance significantly decreased — though it still remained relatively high.



Training and evaluation curves after 2M frames.



Training and evaluation curves from 2M to 2.5M frames.

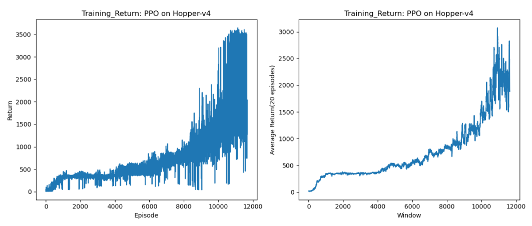
Figure 4: Dueling DDQN performance on Pong-v5 during initial and extended training phases.

Analysis. There are several reasons for the observed trends:

- **Performance drop during 1.2M–1.8M:** The decline may be due to the fixed ϵ value (0.1), which still allows occasional exploration to reach new or rarely visited states. If the agent unfortunately encounters trajectories that appear highly rewarding but are actually noisy, the bootstrapped Q-values can be overestimated. This overestimation propagates through updates and then gets corrected later, producing a dip in performance.
- **Large variance after 2M frames:** The high variance and occasional negative scores during evaluation suggest that the policy was still unstable and sensitive to state transitions in Pong, so we should train continually.
- **Fluctuations during additional 0.5M steps:** When continuing training with a lower ϵ, the agent can become more deterministic and thus prone to larger updates if the replay buffer still contains outlier transitions, which may cause short-term drops.
- **Why final variance remains large:** Even though the final policy is evaluated greedily, the Pong environment's pixel-based input and stochastic opponent behavior can still lead to varying trajectories. Small estimation biases may cause the agent to perform well in some episodes but fail early in others, making it difficult to eliminate score variance completely.

4.2 PPO on Hopper-v4

Results. Figure 5 (Left) shows the training return curve for PPO on Hopper-v4. Overall, the return steadily increases, with a clear boost in performance after around 8,000 episodes. This indicates that the agent gradually learns a stable hopping gait. The snapshot below illustrates the final learned behavior: the agent is able to maintain balance and hop forward efficiently.



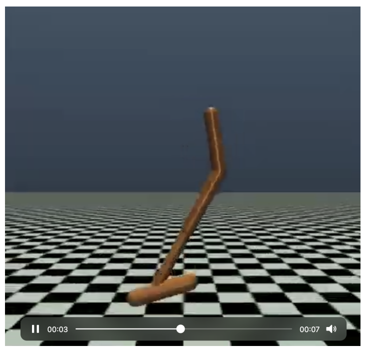


Figure 5: Training performance of PPO on Hopper-v4: (Left) episodic return curve; (Right) agent snapshot demonstrating learned hopping.

Analysis.

- **Hyperparameter choices:** We used a two-layer MLP with 256 hidden units per layer because initial experiments with a 128×128 architecture showed clear underfitting, while larger networks such as 256×256×256 tended to overfit and produced unstable training curves. The number of epochs was set to 8 since Hopper-v4 is relatively simple; increasing the epoch further can result in overly aggressive updates that hurt stability.
- **Performance boost after 8,000 episodes:** Around this point, the agent likely discovers more efficient gaits, resulting in a sharp performance rise.
- **Remaining fluctuations:** Locomotion tasks like Hopper-v4 have inherently unstable dynamics; slight input perturbations can cause the agent to lose balance temporarily.

4.3 PPO on Ant-v4

Results. Figure 6 (Left) shows the training return curve for PPO on Ant-v4. Unlike Hopper-v4, the Ant-v4 task is more complex and shows clear signs of unstable learning: while the average return rises significantly after around 4,000 episodes, it also exhibits large fluctuations and even occasional performance drops. The snapshot demonstrates the final policy: the agent is able to crawl forward efficiently.

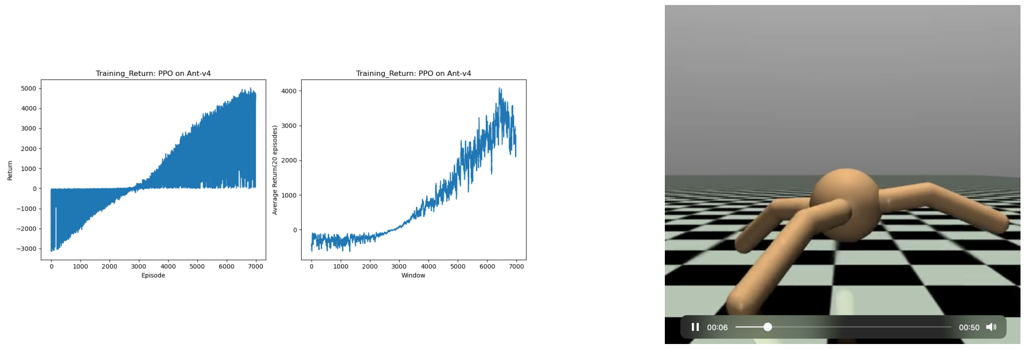


Figure 6: Training performance of PPO on Ant-v4: (Left) episodic return curve; (Right) agent snapshot demonstrating learned crawling.

Analysis.

-
- **Hyperparameter choices:** Considering the higher dimensionality and complex dynamics of Ant-v4, we initially used a three-layer MLP with 256 units per layer. However, further experiments showed that a two-layer 256×256 network achieved similar results without additional underfitting, so we adopted the simpler architecture . The training epoch was kept moderate to balance update stability and sample efficiency.
- **Performance boost after 4,000 episodes:** The agent likely discovers better locomotion strategies at this point, enabling forward crawling.
- **Large fluctuations:** Ant-v4's multiple legs and joints introduce coordination challenges and make the policy more sensitive to small input perturbations. This often leads to unstable gaits and occasional crashes, causing return spikes and drops.

4.4 PPO on HalfCheetah-v4

Results. Figure 7 (Left) shows the episodic return for PPO on HalfCheetah-v4. The overall curve demonstrates a clear upward trend with steadily increasing returns, although small dips can be observed throughout training. Interestingly, the final policy learns to flip its body over and crawl in an upside-down position, which allows stable forward motion but remains suboptimal compared to a natural gait, as shown in the snapshot on the right.

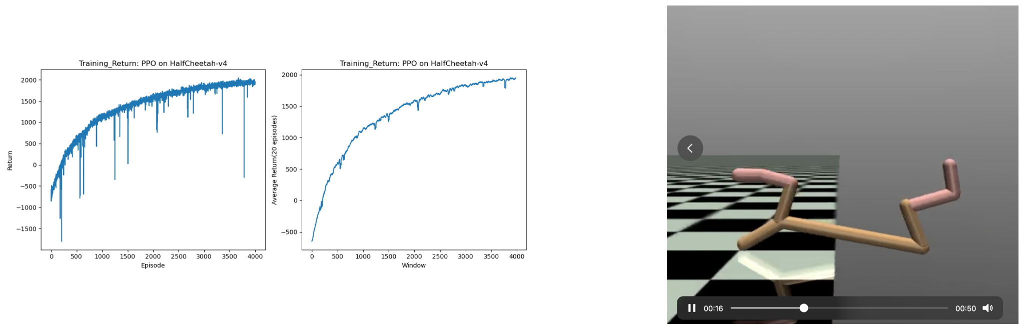


Figure 7: Training performance of PPO on HalfCheetah-v4: (Left) episodic return curve; (Right) agent snapshot showing learned upside-down crawling gait.

Analysis.

- **Hyperparameter choices:** For HalfCheetah-v4, we used a 256×256 MLP structure after observing that smaller networks failed to capture the environment's high-dimensional state-action mappings. A moderate epoch count was used to balance convergence speed and training stability.
- **Frequent small dips:** Minor dips throughout training are likely due to the agent occasionally testing alternative gaits that do not generalize well, causing short-term drops before the policy readjusts.
- **Final gait remains suboptimal:** The agent eventually settles on flipping over and crawling upside-down, which illustrates PPO's tendency to converge to a local optimum that maximizes reward in an unexpected way. While this achieves stable returns, it does not produce a natural running gait.

5 Conclusion

In this project, we implemented and evaluated value-based and policy-based reinforcement learning methods on environments with different action space characteristics. Specifically, we tested Dueling DDQN on Pong-v5 as a representative discrete-action task, and PPO on Hopper-v4, Ant-v4, and HalfCheetah-v4 as continuous-control benchmarks. Our experiments show that:

- **Dueling DDQN** can achieve relatively high performance on simple Atari tasks when combined with sufficient training and appropriate exploration scheduling.
- **PPO** demonstrates stable learning in continuous-control environments, though its final performance depends on careful hyperparameter tuning and is still prone to converging to local optima.

Overall, the results highlight that value-based methods excel in discrete, low-dimensional tasks with well-defined sparse rewards, while policy-based methods like PPO are more suitable for high-dimensional, continuous tasks. However, both approaches face challenges such as sample inefficiency, local optima, and sensitivity to hyperparameters. Future work could include combining multiple improvements (e.g., better exploration, trust region constraints, or reward shaping) to further enhance agent robustness and generalization.