AI-Capstone: Project 2

Reinforcement Learning with PPO and Double DQN

111550034 黃皓君

1. Introduction

In this project, I trained two agents on two different Gymnasium tasks:

- A **Double DQN** agent on the discrete-action Atari game **BeamRider-v5**, chosen because value-based methods excel at high-dimensional visual inputs and discrete control.
- A **PPO** agent on the continuous-action Mujoco environment (**Ant-v5, HalfCheetah-v5, Walker2d-v5)**, chosen because policy-gradient methods handle continuous action spaces and complex physics smoothly.

Each model was selected to match the key characteristics of its environment: DDQN for stable Q-value learning in an Atari shooter, and PPO for stable policy optimization in a locomotion task.

2. Method

2.1 Double Deep Q Network (DDQN)

Traditional Q learning with a neural network ("DQN") tends to overestimate action values because the max operator in the Bellman update uses the same network for both selecting and evaluating the best action.

Core idea of DDQN: Use two networks—one for selecting actions (online network) and one for evaluating them (target network)—so that the update

$$Q_{\text{online}}(s_t, a_t) \leftarrow Q_{\text{online}}(s_t, a_t) + \alpha \left[r_{t+1} + \gamma\, Q_{\text{target}}\left(s_{t+1}, \arg\max_{a'} Q_{\text{online}}(s_{t+1}, a')\right) - Q_{\text{online}}(s_t, a_t) \right].$$

Action selection via $\arg\max_{a'} Q_{\text{online}}(s_{t+1}, a')$.

Value evaluation via $Q_{\text{target}}(s_{t+1}, \cdot)$

Target network parameters are periodically synced to the online network to stabilize learning.

2.2 Proximal Policy Optimization (PPO)

Policy-gradient methods directly adjust a parameterized policy $\pi_\theta(\, a_t \mid s_t \,)$ to maximize expected return.

Key concepts:

1. **Probability ratio**

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

2. **Clipped surrogate objective**

   Limits $r_t(\theta)$ to $[1-\epsilon, 1+\epsilon]$ to prevent large policy shifts:

$$[L^{\text{CLIP}}(\theta) = E_t\big[\min\big(r_t(\theta)\,\widehat{A_t},\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\widehat{A_t}\big)\big].]$$

   $\widehat{A_t}$ is the advantage estimate, typically computed via Generalized Advantage Estimation (GAE).

3. **Full loss**

   Adds value-function error and an entropy bonus to encourage exploration:

$$[L(\theta) = -L^{\text{CLIP}}(\theta) + c_1\,E_t[(V_\theta(s_t) - R_t)^2] - c_2\,E_t\big[\mathcal{H}\big(\pi_\theta(\cdot \mid s_t)\big)\big].]$$

   R_t is the empirical return,

   c_1 and c_2 are coefficients for value loss and entropy bonus, respectively.

3. Task Descriptions

3.1 Atari

BeamRider-v5:

Environment: Atari 2600 shooter where you pilot a constantly forward-moving spacecraft, steering sideways to destroy enemy ships, dodge debris, and avoid enemy fire.

Action space: Discrete(9) – actions include NOOP, FIRE, and eight movement/fire combinations.

Observation space: Box(0, 255, (210, 160, 3), uint8) – raw RGB video frames.

Reward: Points awarded for each enemy destroyed; positive for successful hits, zero otherwise.

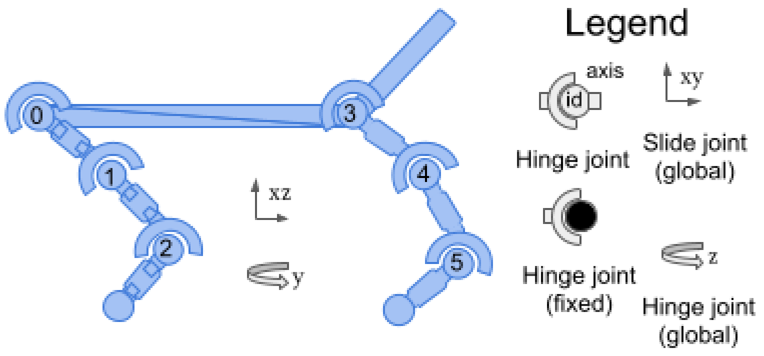
3.2 Mujoco

I started from a simple Multi-Joint robot Cheetah since it seems easy to train and will not die due to balancing.

Half Cheetah:

Environment: A 2D “HalfCheetah” robot that must learn to run forward by applying torques at hip and shin joints.

Action space: Box(-1.0, 1.0, (6,), float32) – torques for six hinge joints.



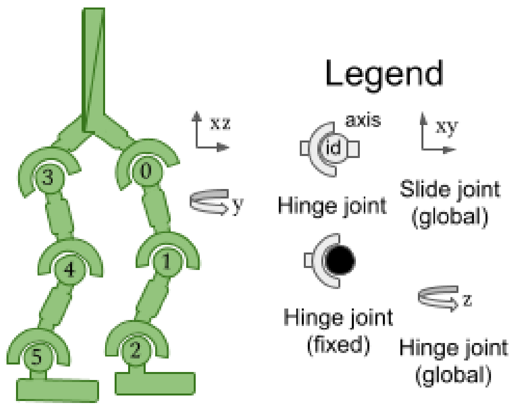
Observation space: Box(-Inf, Inf, (17,), float64) by default (positions & velocities of body parts).

Reward: forward_reward - ctrl_cost.

After Cheetah I tried a harder Walker robot that need more balancing effort.

Walker2d:

Environment: A 2D bipedal robot ("Walker2d") with two legs; the task is to walk forward by applying torques at hip, knee, and ankle joints.



Action space: Box(-1.0, 1.0, (6,), float32) – torques for six hinge joints.

Observation space: Box(-Inf, Inf, (17,), float64) by default (positions & velocities of joints).

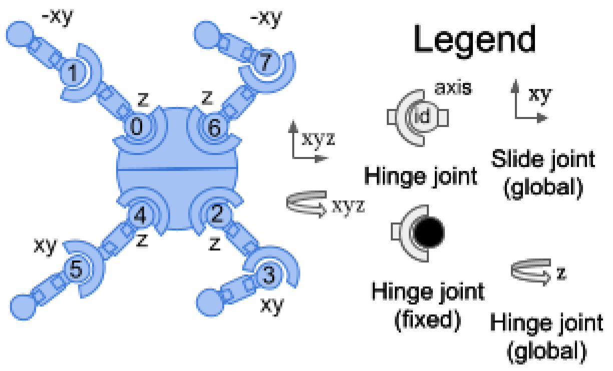
Reward: healthy_reward bonus + forward_reward - ctrl_cost.

Since above two robots are all in 2D environment. I want to test my model if it can handle 3D environment.

Ant:

Environment: A 3D quadruped "Ant" robot that must learn to run forward by applying torques at hip and ankle joints.

Action space: Box(-1.0, 1.0, (8,), float32) – torques for 8 hinge joints.



Observation space: Box(-Inf, Inf, (105,), float64) by default (position values, velocities of joints, center of mass based external forces on the body parts.)

Reward: healthy_reward + forward_reward - ctrl_cost - contact_cost.

4. Implementation

4.1 DDQN on BeamRider-v5

Environment & Preprocessing: ALE/BeamRider-v5, RGB → grayscale 81×81, stack 4 frames (custom FrameStack).

Network:

- Conv layers: (32, 8×8, stride 4) → (64, 4×4, stride 2) → (64, 3×3, stride 1)
- FC512 → action-value outputs.

Replay Buffer: capacity 100 000, batch = 32.

Exploration: ε-greedy, ε decays from 1.0→0.1 over 100 000 steps.

Target Net: sync every 1 000 steps.

Optimizer: Adam (lr 1e-4), MSE loss on TD target.

Logging: TensorBoard episode rewards; checkpoint & video every 50 episodes.

4.2 PPO on Mujoco

Environments: 32 async env(ex. HalfCheetah-v5), reward scaling 0.005.

Agent:

- Actor: MLP with hidden sizes interpolated between input_dim×20 and action_dim×10, outputs Gaussian mean (Tanh) + learnable log-std.
- Critic: symmetric MLP → scalar value.

Buffer: stores 2048 steps × 32 envs, computes GAE (γ=0.995, λ=0.98).

Training:

- 3 000 epochs; each collects 2 048 × 32 samples.
- 20 PPO update iterations per epoch, minibatch = 16384.
- Clip ε=0.1; entropy coef = 1e-4; value coef = 1.0.

Logging: TensorBoard mean reward & LR; save best model on improvement; record video every 25 epochs.

5. Experiments

5.1 DQN vs DDQN

Standard DQN tends to *overestimate* action values because it uses the same network both to pick the best next action and to evaluate its value:

$$y_{DQN} = r_{t+1} + \gamma \max_{a'} Q_{\text{online}}(s', a')$$

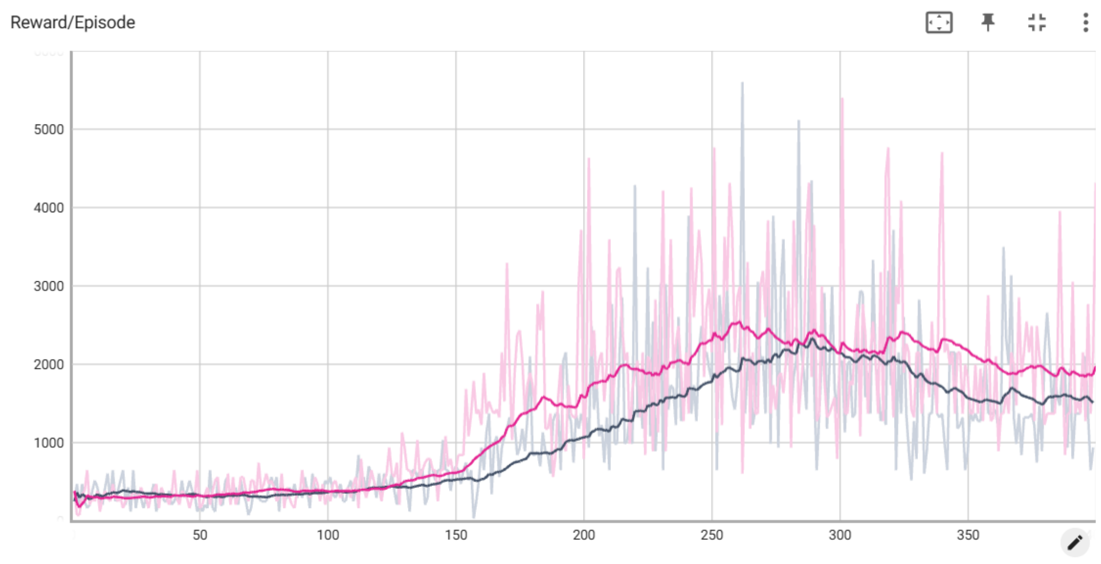
This maximization bias can lead to overly optimistic targets and unstable learning. **Double DQN** fixes this by **decoupling** selection and evaluation:

$$y_{DDQN} = r_{t+1} + \gamma\, Q_{\text{target}}\left(s', \max_{a'} Q_{\text{online}}(s', a')\right)$$

This separation reduces overestimation bias, resulting in more accurate value estimates, smoother updates, and ultimately higher average and peak performance with lower variance, as reflected in our results above.
The comparison is shown below:

Methods	Mean Reward	Std Dev	Max Reward	Min Reward
DQN	1806.51	860.73	5602	528
DDQN	2169.88 (+20.1 %)	928.62	5404 (−3.5 %)	616 (+16.7 %)

Compare the 200–400 episodes.
Plot of the result:



Colors: pink - DDQN, blue - DQN

Analisis:

- **Mean Reward:** DDQN improves average by **20.1 %**.
- **Max Reward:** Slightly lower peak (−3.5 %), but still comparable.
- **Min Reward:** Worst-case score up by **16.7 %**, showing fewer low-reward outliers.

5.2 PPO Network Capacity Ablation

Assess how actor–critic MLP capacity impacts PPO learning speed and performance on **HalfCheetah-v5**, considering only epochs 1–500.

Configurations:

- **Small:** hidden sizes = *input_dim* × 10 → *action_dim* × 5
- **Baseline:** hidden sizes = *input_dim* × 20 → *action_dim* × 10
- **Large:** hidden sizes = *input_dim* × 40 → *action_dim* × 20

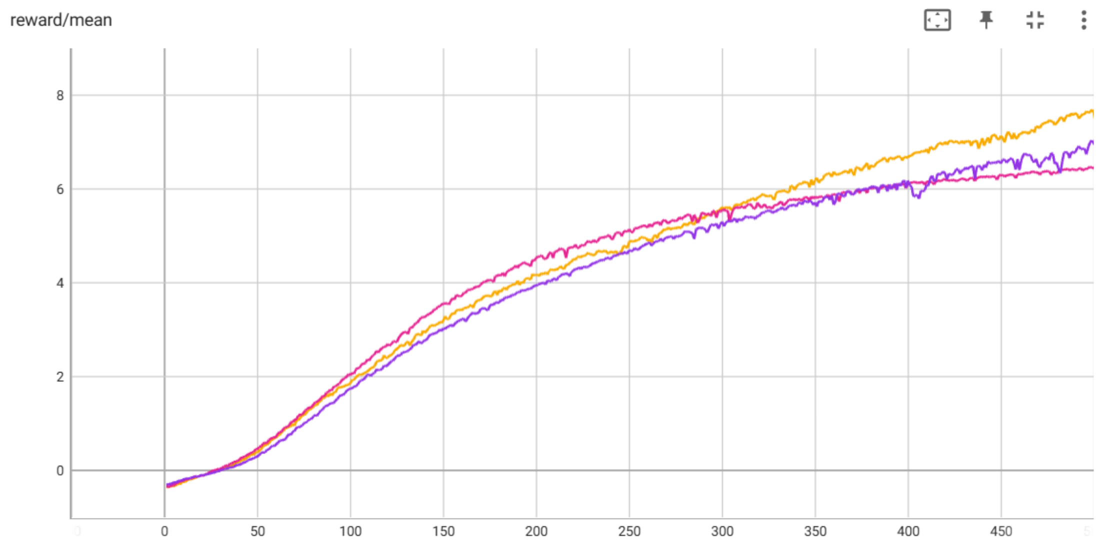
Metrics (computed over the last 10 epochs, Steps 491–500):

Configuration	Final Mean Return	Std Dev	Epoch to 80 % Baseline
Small	7.61 (+10.3 %)	0.0535	297 (–8.3 %)
Baseline	6.9023	0.0856	324
Large	6.4395 (–6.7 %)	0.0235	295 (–8.9 %)

Final Mean Return = average reward over epochs 491–500.

Epoch to 80 % = first epoch where return ≥ 5.5219.

Plot of the result:



Colors: orange - Small, purple - Baseline, pink - Large

Analysis:

- The **Small** network boosts the final mean return by **10.3 %** over the Baseline, with very low variance (std = 0.054).
- The **Large** network learns fastest initially (reaching the 80 % mark at epoch 295, **8.9 %** sooner than Baseline) but ends up **6.7 %** below the Baseline's final return.
- The Baseline configuration sits between the two, taking 324 epochs to hit the threshold.

Conclusion

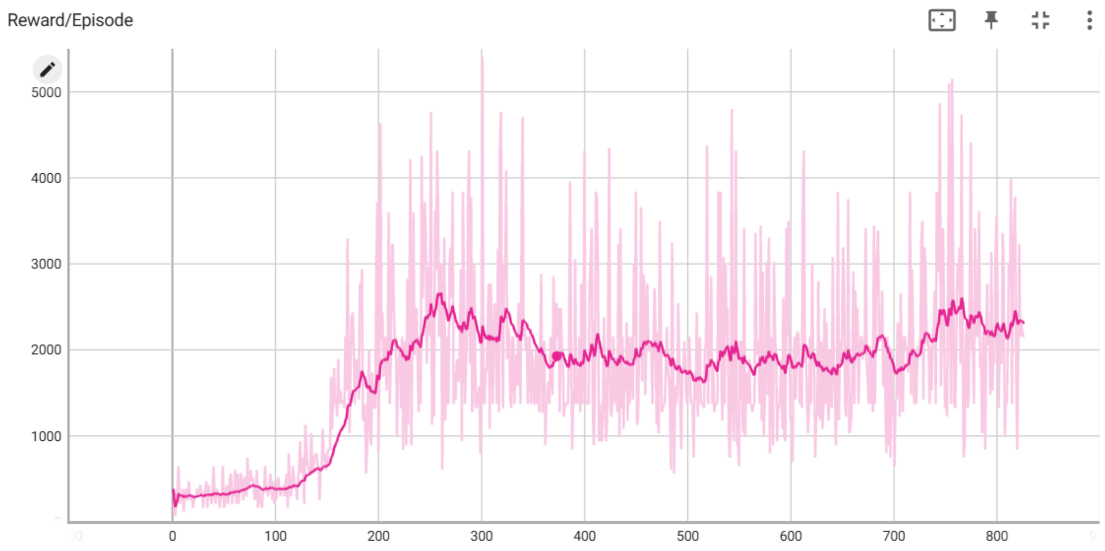
A compact MLP (Small capacity) not only accelerates learning (–8.3 % fewer epochs to threshold) but also improves ultimate performance (+10.3 %), due to its leaner parameterization that reduces overfitting, lowers variance, and boosts sample efficiency. Over-parameterization (Large capacity) can speed early progress but may hinder convergence within a limited 500-epoch budget by wasting capacity on excess parameters that yield diminishing returns. For HalfCheetah-v5 under a 500-epoch budget, adopt the **Small** network — its **10 %**

return gain and **8 % faster** learning make it the most effective choice.

6. Result

6.1 Atari Beamrider DDQN

Plot of the result for 800 episodes:



Playing vedio: [Gymnasium Atari Beamrider DDQN]()

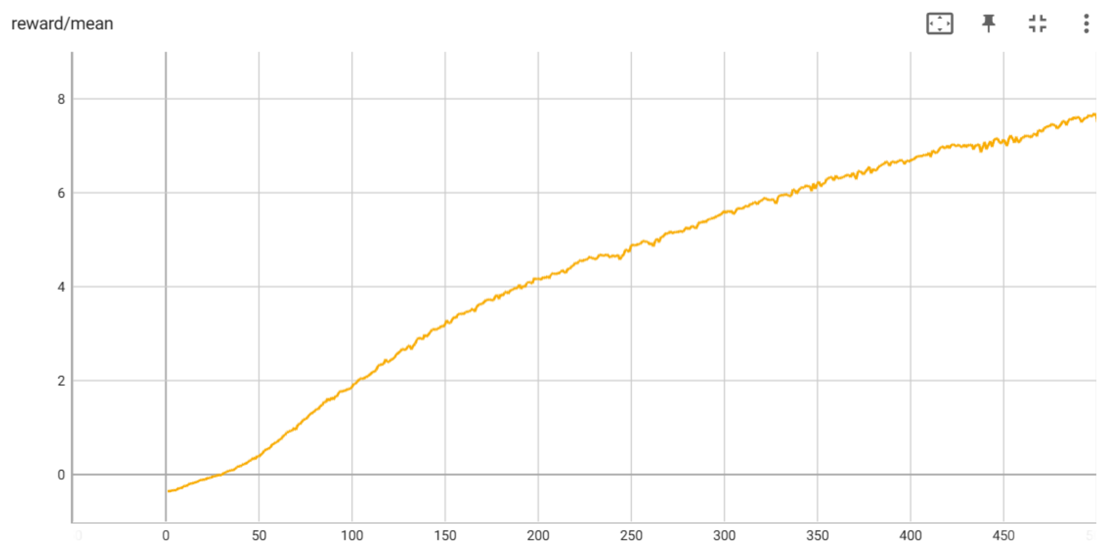
6.2 Mujoco PPO Network

6.2.1 HalfCheetah (Epoch 50 vs. 450)

- Epoch 50:
  - The agent produces large, jerky oscillations of the torso and legs.
  - Forward motion is very slow and intermittent: often it overextends one leg and loses balance, dragging itself forward.
  - No clear "running" gait—more a series of uncontrolled hops.
- Epoch 450:
  - You'll see a smooth, cyclical gait with alternating hip and ankle flexion.
  - The torso remains level and roughly parallel to the ground, indicating good balance.
  - Forward velocity is consistent: each stride contributes reliably to forward progress with minimal wasted motion.

Summary: By 450 epochs, PPO has learned coordinated control of six joints to achieve stable running, whereas at 50 epochs it was still exploring basic leg movements.

Reward plot:



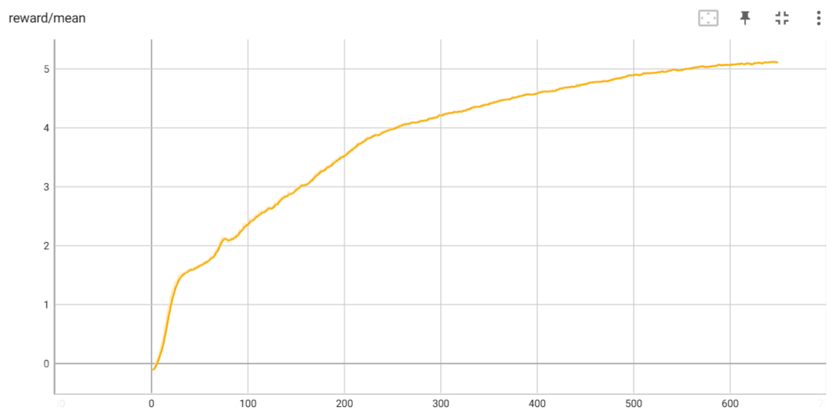
Playing vedio: Gymnasium Mujoco Half-Cheetah PPO

6.2.2. **Walker2d** (Epoch 50 vs. 650)

- Epoch 50:
  - Early steps are tentative: the agent leans too far forward or backward and often falls within a few timesteps.
  - Leg movements look "stiff" and out of sync—knees and hips both extend or flex at the same time, giving a rigid, crab-like appearance.
- Epoch 650:
  - A fluid, balanced bipedal gait emerges: knees bend as the opposite hip extends, mimicking human-like walking.
  - The center of mass oscillates minimally, and the walker maintains an upright posture.
  - Forward speed is steady and the agent seldom loses balance.

Summary: PPO transitions from uncoordinated stumbles to a stable two-legged walk by ~650 epochs, mastering timing between hip and knee joints.

Reward plot:



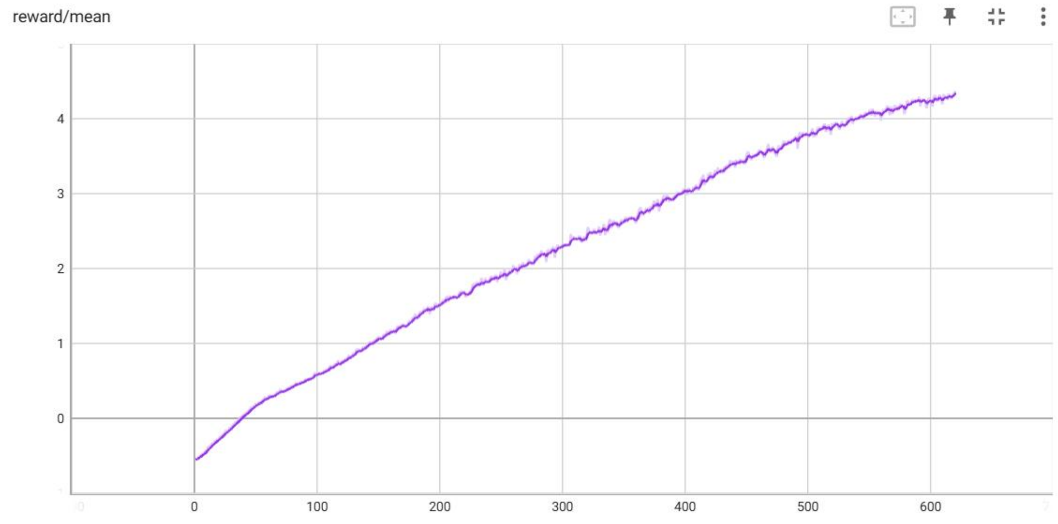
Playing vedio: Gymnasium Mujoco Walker2D PPO

6.2.3. Ant (Epoch 50 vs. 600)

- Epoch 50:
  - Legs move almost independently and out of phase, so the body often spins in place or tips over.
  - There's no clear directional intent—sometimes it flails all eight legs simultaneously, sometimes only two or three.
- Epoch 600:
  - You'll observe a clear, multi-legged walking pattern: diagonal pairs of legs move in synchrony.
  - The torso remains roughly level and travels forward in a straight line, with minimal sideways drift.
  - Speed and stability are both greatly improved compared to the early video.

Summary**:** Learning for the 8-legged Ant requires finding coordinated phase relationships; PPO achieved a stable, efficient gait by ~600 epochs.

Reward plot:



Playing vedio: [Gymnasium Mujoco Ant PPO]()

6.2.4. Summary

Across all three environments, PPO shows the classic progression from random or uncoordinated movements in the first few dozen epochs to smooth, efficient locomotion by several hundred epochs.

Key indicators of maturity are:

1. Balance: Torso remains level.
2. Rhythm: Joints move in a consistent, repeating pattern.
3. Efficiency: Forward velocity becomes steady with low variance.

Watching these side-by-side highlights how PPO gradually refines joint coordination, trading exploratory flailing for a reliable gait tuned to each environment’s dynamics.

7. Discussion

7.1 Learnings from the Experiments

Throughout this project, I not only learned the theoretical ideas behind DQN and PPO, but also gained valuable hands-on experience implementing, tuning, and testing reinforcement learning agents. For example, I personally observed how addressing DQN’s overestimation bias with Double DQN helped stabilize training , boosting average reward by over 20%. I also experimented with different MLP sizes and saw surprising results—sometimes smaller networks performed better due to faster convergence and less overfitting.

Setting up TensorBoard helped me track all of this more clearly. Overall, these experiments helped me understand how algorithms work not just in theory, but in practice, and how to design better training setups based on both data and intuition.

7.2 Remaining Open Questions

- **Prioritized Replay:** Could adding priority to the replay buffer speed up DDQN on sparse-reward stages?
- **Cross-Task Hyperparameter Transfer:** Will our tuned PPO settings work well on other, unseen environments?
- **Hybrid Method:** Might combining value-based and policy-gradient approaches (e.g. DDPG, SAC) improve efficiency or stability?
- **Scaling to Real-World Robotics:** What practical hurdles appear when moving from simulation to real robots?

Reference

DDQN: Tackling Overestimation Bias in Deep Reinforcement Learning | by Dong-Keon Kim | Medium
[RL] Proximal Policy Optimization(PPO) - HackMD
Github/ProfessorNova/PPO-Humanoid.

Appendix

Github link: qwer521/Gymnasium-RL