AI-Capstone: Project2 Report

I-Ting Chu, 111550093

1. Introduction

In this project, we aim to explore and compare the performance of different reinforcement learning (RL) algorithms across a variety of environments provided by OpenAI Gym. Specifically, I focus on three representative environments of increasing complexity: *Classic/Pendulum-v1*, *Box2D/LunarLander-v3*, and *Atari/Pong-v5*. To address these tasks, we evaluate three widely used RL algorithms: PPO, DQN, and A2C. Our goal is to understand the strengths and limitations of each algorithm in different settings, as well as to investigate how various hyperparameters and implementation choices affect their learning behavior and convergence properties.

The following links show the trained agents performing in their respective environments:

- **Pendulum-v1 (PPO)**: link
- **LunarLander-v3 (PPO)**: link
- **Pong-v5 (DQN)**: link

2. Methods

In this section, we provide a brief overview of the three reinforcement learning algorithms applied in our experiments: Proximal Policy Optimization (PPO), Deep Q-Network (DQN), and Advantage Actor-Critic (A2C). Each of these methods represents a different class of RL algorithms and offers unique mechanisms for policy optimization.

2.1. PPO

Proximal Policy Optimization (PPO) is a policy gradient method that optimizes a clipped surrogate objective to prevent large policy updates and ensure training stability. The clipped objective is defined as:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, and $\hat{A}_t$ is the estimated advantage. The clipping parameter ϵ (commonly 0.1 or 0.2) limits the change in policy to improve stability. [4]

2.2. DQN

Deep Q-Network (DQN) is a value-based method that learns to approximate the optimal Q-value function $Q^*(s, a)$ using a neural network. The loss function for DQN is the mean squared error between the predicted and target Q-values:

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

Here, Q_{θ^-} is the target network, which is periodically updated to the current network Q_θ, and γ is the discount factor. Experience replay and target networks help stabilize training. [2]

2.3. A2C

Advantage Actor-Critic (A2C) is an on-policy algorithm that combines value-based and policy-based learning. It uses the advantage function $\hat{A}_t = R_t - V(s_t)$ to reduce variance in policy updates. The actor and critic networks are updated with the following losses:

Actor loss (policy gradient):

$$L^{\text{actor}} = -\log \pi_\theta(a_t|s_t) \cdot \hat{A}_t$$

Critic loss (value regression):

$$L^{\text{critic}} = (R_t - V_\phi(s_t))^2$$

The total loss is often the weighted sum:

$$L = L^{\text{actor}} + c_1 L^{\text{critic}} - c_2 \mathcal{H}(\pi_\theta)$$

where $\mathcal{H}(\pi_\theta)$ is an entropy bonus that encourages exploration, and c_1, c_2 are weighting coefficients. [1]

3. Implementation Detail

In this section, we describe the implementation details of our experiments, including the framework we used and the environments selected for testing. All experiments were implemented in Python using the Stable-Baselines3 (SB3) library.

Environment	Characteristics
Pendulum-v1	3D state, continuous action, dense reward, 200 steps
LunarLander-v3	8D state, discrete action (4), dense reward, 1000 steps
Pong-v5	$210 \times 160 \times 3$ image, discrete action (6), sparse reward, 10000 frames

Table 1. Environment summary: state/action space, reward type, and episode length.

3.1. Stable-Baselines3 (SB3)

We use Stable-Baselines3 (SB3) [3] version 2.6 as our main reinforcement learning framework. SB3 provides reliable and well-tested implementations of many modern RL algorithms, including PPO, DQN, and A2C. It offers a clean API for training and evaluation, as well as integration with Gymnasium environments and vectorized environments for efficient simulation.

3.2. Testing Environments

To evaluate the performance of each RL algorithm, we selected three diverse environments from different categories in OpenAI Gymnasium. The summary of their key properties is shown in Table 1. Also, the screenshots of each environment are shown in Figure 1, 2.

Figure 1. Screenshots of Pendulum-v1, LunarLander-v3.

3.2.1 Classic Control: Pendulum-v1

Pendulum-v1 is a continuous control task where the goal is to keep a frictionless pendulum upright by applying torque. The task features a low-dimensional, continuous state space and a continuous action space. The reward is shaped and dense, penalizing angle deviation and control effort.

3.2.2 Box2D: LunarLander-v3

LunarLander-v3 is a physics-based 2D landing task. The agent must control a lander to safely land on a pad with discrete actions. The environment provides shaped rewards for progress and penalties for crashing or using excessive fuel.

3.2.3 Atari: Pong-v5

Pong-v5 is a classic Atari 2600 game and a standard benchmark for discrete, pixel-based deep RL. The agent observes raw pixel images and learns to control the paddle to defeat the opponent. Rewards are sparse and given only when a point is scored.

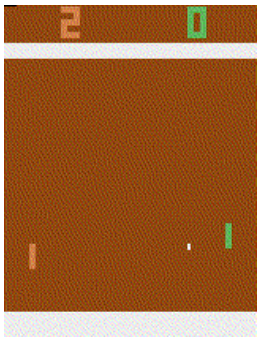


Figure 2. Screenshots of Pong-v5.

3.3. Metrics

To evaluate training performance, we used `EvalCallback` provided by Stable-Baselines3 [3]. During training, the model was periodically evaluated on a separate environment, and the best-performing model was saved. The evaluation setup is as follows:

```
eval_callback = EvalCallback(
    eval_env,
    eval_freq=5000,
    deterministic=True,
    render=False,
)
```

We also monitored training progress using TensorBoard, focusing on:

- **Loss curves** of the policy and value networks.
- **Episode rewards** over time to assess convergence speed and stability.

3.4. Hyperparameters

For fair comparisons across different algorithms and environments, we adopted the following default hyperparameter settings unless otherwise stated:

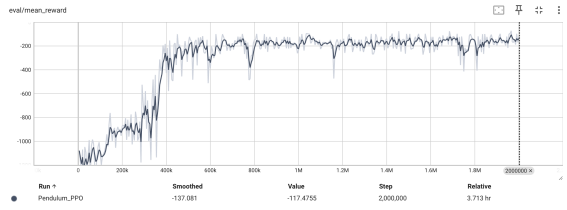
- **Random seed**: 42
- **Learning rate**: 3×10^{-4}

- **Optimizer**: Adam
- **Policy type**:
  - `CnnPolicy` for Pong-v5 (trained on `cuda`)
  - `MlpPolicy` for Pendulum-v1 and LunarLander-v3 (trained on `cpu`)
- **gamma**: 0.99
- **Total timesteps**: 2,000,000
- **Evaluation frequency**: 5,000 steps

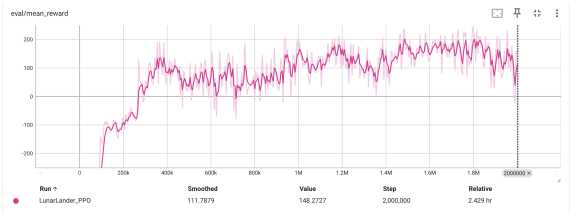
4. Experiment

4.1. Environments

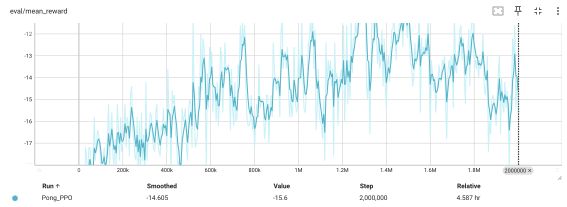
To compare convergence speed across environments, we trained PPO with the same setting (batch size = 128, total steps = 2,000,000) on Pendulum-v1, LunarLander-v3, and Pong-v5. The results are shown in Figure 3.



(a) Pendulum-v1



(b) LunarLander-v3



(c) Pong-v5

Figure 3. PPO reward curves for different environments. Each was trained for 2 million timesteps with batch size 128.

From the reward curves, I observe the following:

- **Pendulum-v1** converges rapidly and stably, reaching near-optimal performance at around 500k steps, with rewards oscillating in a narrow range.
- **LunarLander-v3** also converges fairly quickly around 400k steps, reaching a reward of 100. However, it exhibits higher variance and oscillation, typically between 100 and 200.
- **Pong-v5** shows the slowest and least stable learning progress. The reward curve exhibits large fluctuations and fails to reach high performance within 2M steps.

We hypothesize that the differences in convergence behavior stem from inherent properties of the environments (see Table 1):

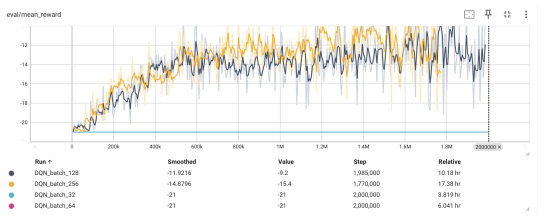
- **State space complexity**: Pendulum has the lowest-dimensional state (3D), LunarLander is moderate (8D), while Pong requires interpreting raw pixel images of size $210 \times 160 \times 3$.
- **Observation type**: Pong uses raw image input, placing heavy reliance on CNN feature extraction, while the other two environments provide structured, low-dimensional observations.
- **Reward sparsity**: Pong provides sparse rewards (only on point scores), which leads to more difficult credit assignment for policy updates.
- **Episode length**: Pong episodes last significantly longer, requiring the agent to interact over more frames before receiving reward signals, delaying learning feedback.

4.2. Batch Size

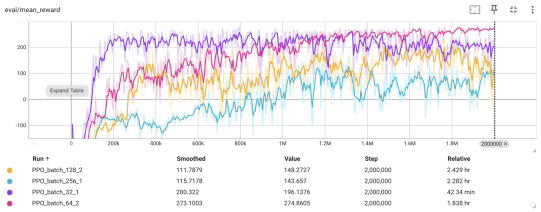
To investigate the effect of batch size on learning, we conducted experiments using the same algorithm (either PPO or DQN) across four batch sizes: 32, 64, 128, and 256. The experiments were repeated for three different environments: Pong-v5, LunarLander-v3, and Pendulum-v1. The results are visualized in Figure 4.

4.2.1 Pong-v5 (DQN)

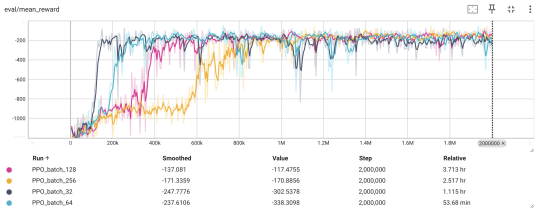
When using small batch sizes (32 or 64), the agent completely fails to learn, with the mean reward remaining flat at the minimum value of -21 throughout training. This is likely due to the sparse nature of rewards in Pong—successful experiences are rare in the early training phase. A small batch size increases the chance of drawing only uninformative transitions, making it hard for the Q-network to bootstrap meaningful updates. Batch size 128 starts to learn slowly, and 256 shows marginal improvement in stability.



(a) DQN on Pong-v5



(b) PPO on LunarLander-v3



(c) PPO on Pendulum-v1

Figure 4. Effect of different batch sizes (32, 64, 128, 256) on reward convergence in three environments.

4.2.2 LunarLander-v3 (PPO)

In contrast, smaller batch sizes (e.g., 32, 64) appear to converge to higher average rewards than larger ones in LunarLander. A plausible explanation is that smaller batches introduce more stochasticity in gradient estimation, which could act as an implicit exploration mechanism or regularization, especially in moderately shaped environments like LunarLander. Larger batch sizes may overfit to fewer high-reward strategies early on, becoming less flexible later.

4.2.3 Pendulum-v1 (PPO)

I observe that smaller batch sizes lead to significantly faster convergence in Pendulum. The convergence points approximately double with each increase in batch size (around 100k, 200k, 400k, and 800k steps). One possible reason is that Pendulum has a low-dimensional, dense-reward setting, and its dynamics are relatively smooth. Small batch sizes can quickly capture enough useful gradients to adjust the policy without over-smoothing updates. In contrast, larger batches average over many redundant samples, delaying the policy shift toward the optimal region.

4.3. Algorithms

To compare the learning performance of different algorithms under the same setting, I ran PPO, DQN, and A2C on Pong-v5 with a fixed batch size of 128 and 2 million timesteps. The results are shown in Figure 5.

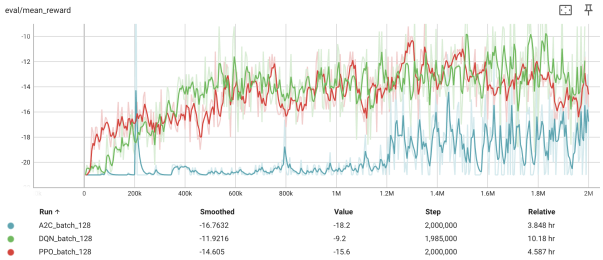


Figure 5. Mean episode reward of PPO, DQN, and A2C on Pong-v5 with batch size 128.

From the reward curves, we observe the following:

- **PPO and DQN** both demonstrate gradual learning, with DQN achieving slightly higher final rewards than PPO. However, the difference in performance is not significant.
- **Training time**, on the other hand, shows a clear gap. DQN took more than twice the training time compared to PPO to reach similar performance. This is likely due to the extra cost from the experience replay buffer and frequent Q-network updates.
- **A2C** underperforms significantly. While it shows some learning trend, the average reward remains close to the minimum for most of the training period, with high variance and no stable improvement. This suggests that A2C struggles in environments with sparse rewards like Pong, especially without extra tricks like entropy tuning or parallel environments.

These results indicate that for sparse-reward environments like Pong, value-based methods (DQN) and policy-based methods (PPO) can both be effective, but PPO may be more practical due to its faster training. A2C, while efficient in shaped-reward environments, is less suitable for Atari-style games without significant modification.

4.4. DQN Experiments

4.4.1 Hard Update vs. Soft Update

In Stable-Baselines3, DQN uses a hard update strategy by default, where the target network is updated every fixed number of steps. To evaluate the impact of soft updates, we modified the update method using the parameter `tau`, which controls the Polyak averaging factor (See Table 2).

Configuration	τ	Target network update
Hard update	1.0	Copied from the main network every fixed interval.
Soft update	0.005	Updated slowly using Polyak averaging: $\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$.

Table 2. Comparison of DQN update strategies on Pong.

Group	Final ϵ	Fraction	Notes
A	0.05	0.1	Fastest convergence; favors early exploitation.
B	0.01	0.5	Longer exploration phase.
C	0.10	0.3	Balanced between exploration and exploitation.

Table 3. Comparison of ϵ-greedy decay schedules in DQN on Pong.

We set `tau = 0.005` to enable soft updates while keeping the batch size at 128. As shown in Figure 6, the learning performance deteriorates significantly under soft update—the agent fails to learn a meaningful policy and reward remains flat near the minimum.

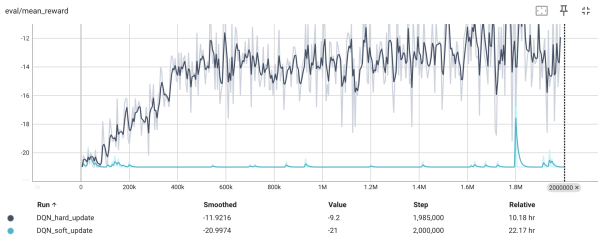


Figure 6. Performance comparison of DQN with hard update (default) and soft update ($\tau = 0.005$).

This result suggests that for environments like Pong with sparse rewards, target stability is important, and soft updates may be too conservative to enable efficient Q-value propagation.

4.4.2 Exploration vs. Exploitation Tradeoff

To analyze the effect of ϵ-greedy decay scheduling, we tested three groups of exploration settings with the same initial $\epsilon = 1.0$ but different final values and decay fractions (see Table 3).
From the experimental results (see Figure 7), we observe the following:

- **Group A** converges the fastest and shows stable learning behavior early on, but plateaus at a slightly lower reward (~ -13.4). The early exploitation allows quick policy refinement but may lead to local optima due to insufficient exploration.
- **Group B** achieves the highest final reward (~ -11.8), though its learning process is initially slower. The prolonged exploration phase helps gather a diverse set of transitions, allowing the Q-network to generalize better and avoid premature convergence.
- **Group C** underperforms in both reward and learning speed. Its final reward is the lowest (~ -14.3), and its episode length stabilizes at a lower level. The decay schedule might be too shallow to encourage deep exploration, but not aggressive enough to exploit effectively.

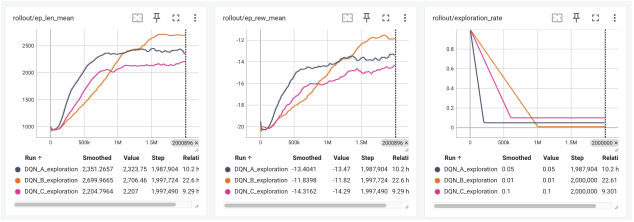


Figure 7. Reward, episode length, and exploration rate for different ϵ-decay schedules.

We conclude that in sparse-reward environments like Pong, a longer exploration phase (as in Group B) may lead to better long-term performance, despite a slower start. Fast decay (Group A) can offer quicker results but risks suboptimal convergence. Balanced decay (Group C) might fail to provide a strong signal in either direction, resulting in mediocre performance.

5. Discussion

5.1. RL Algorithm and Environment Specificity

Through the experiments, it became evident that no single RL algorithm or configuration universally performs best across all environments. For instance, PPO works exceptionally well in low-dimensional, dense-reward settings like Pendulum, but struggles in sparse-reward tasks like Pong without proper tuning. DQN performs better in Pong but requires more training time and is sensitive to batch size and ϵ decay scheduling. This highlights the importance of adapting algorithm choices and hyperparameters based on the characteristics of the environment—state representation, reward density, and episode dynamics all play a crucial role.

5.2. Using CPU or GPU?

Contrary to initial expectations, using a GPU does not always result in faster training. For lightweight models such

as `MlpPolicy` applied to environments like Pendulum or LunarLander, CPU execution can be more efficient. This is because the overhead of transferring data between CPU and GPU can exceed the speedup gained from parallelization, especially when the model is small or the batch size is limited. On the other hand, environments with image inputs (e.g., Pong) and policies like `CnnPolicy` benefit significantly from GPU acceleration.

5.3. Lessons Learned

This project provided a deeper understanding of the nuances in training reinforcement learning agents. The experiments showed that:

- Sparse rewards are particularly challenging for value-based methods, and careful tuning of exploration strategies is crucial.
- Hyperparameters such as batch size, update frequency, and exploration schedules have dramatic effects on convergence speed and final performance.
- Empirical experimentation is indispensable—many insights (e.g., GPU being slower in small models) would not be obvious without hands-on trials.

Overall, the experience reinforced the idea that reinforcement learning is highly context-sensitive, and success depends not just on choosing a good algorithm, but on tailoring it to the environment's properties and the computational setup.

References

[1] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016. 1

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. 1

[3] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Noah Ernestus, and Raphaël Dormann. Stable-baselines3 documentation, 2021. https://stable-baselines3.readthedocs.io/en/master/index.html. 2

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 1