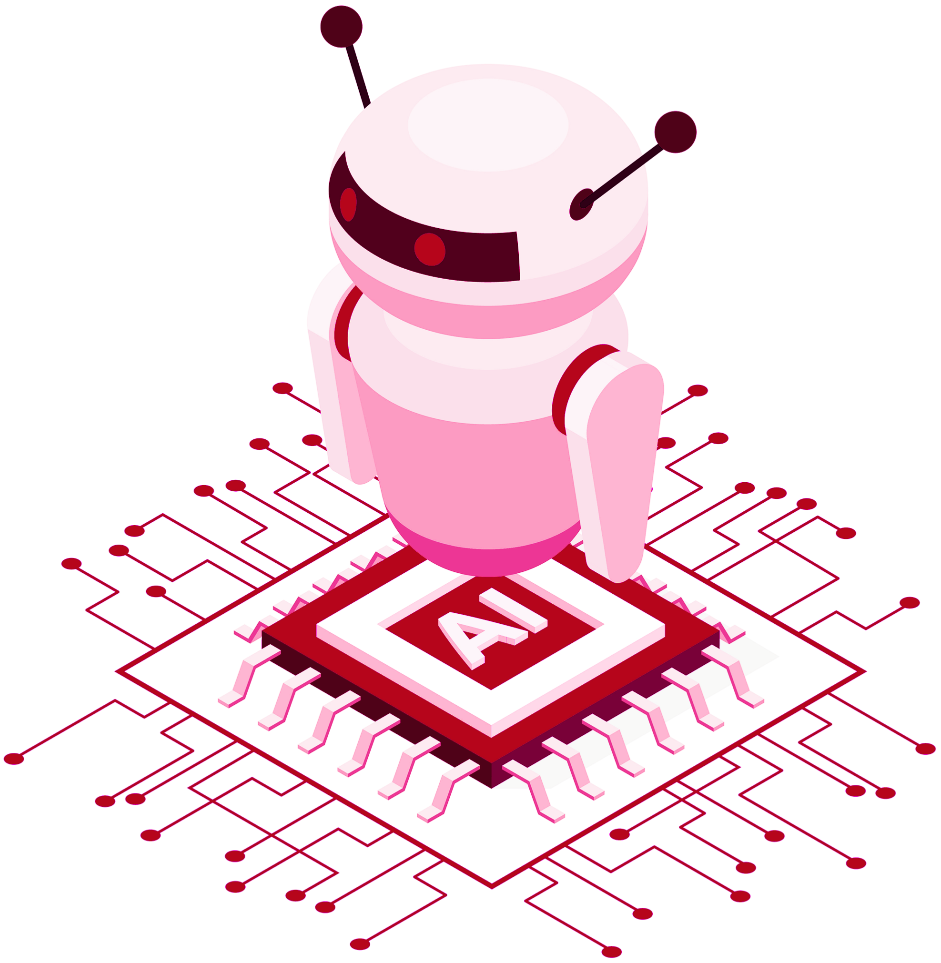

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 1:

Introduction to RL

By:

[Payam Taebi]

[400104867]

Spring 2025

Contents

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	Points
Task 1: Solving Predefined Environments	45
Task 2: Creating Custom Environments	45
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing a wrapper for a known env	10
Bonus 2: Implementing pygame env	20
Bonus 3: Writing your report in Latex	10

Notes:

- Include well-commented code and relevant plots in your notebook.
- Clearly present all comparisons and analyses in your report.
- Ensure reproducibility by specifying all dependencies and configurations.

1 Task 1: Solving Predefined Environments[45-points]

1.1 Overview

After running a very basic (and not entirely successful) initial training, I decided to work on two environments: Taxi-v3 and FrozenLake. My plan was to test five models (A2C, PPO, DQN, QRDQN, and RecurrentPPO) on each environment. However, the last two models turned out to be extremely complex, each taking about 3 hours to train. Due to time constraints, I dropped those and trained only three models (A2C, PPO, and DQN) per environment. Also, the training files were huge and I generated many GIFs for visualization—although on Colab the GIF files sometimes crashed, in the notebook all GIFs are visible (the GIF files themselves aren't directly accessible).

Without further ado, let's dive into the details for Taxi-v3.

1.2 Taxi-v3 Environment Details

The Taxi-v3 environment is defined by:

- **Observation Space:** `Discrete(500)`
- **Action Space:** `Discrete(6)`
- **Number of Possible States:** 500
- **Number of Possible Actions:** 6
- **State Encoding:** The state is a tuple:
  - Taxi row: 0–4 (5 possible values)
  - Taxi col: 0–4 (5 possible values)
  - Passenger locations: {0: R, 1: G, 2: Y, 3: B, 4: In taxi}
  - Destination locations: {0: R, 1: G, 2: Y, 3: B}
- **Action Mapping:**
  - 0: Move South
  - 1: Move North
  - 2: Move East
  - 3: Move West
  - 4: Pick Up Passenger
  - 5: Drop Off Passenger

Example of Available Transitions:

- **Action 0 (Move South):** Prob: 1.00, Next State: 233, Reward: -1, Done: False
- **Action 1 (Move North):** Prob: 1.00, Next State: 33, Reward: -1, Done: False
- **Action 2 (Move East):** Prob: 1.00, Next State: 133, Reward: -1, Done: False

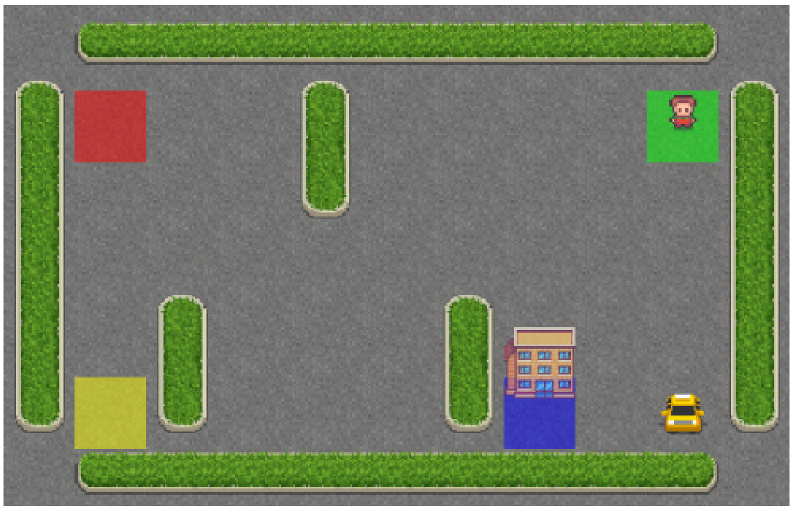

Figure 1: Taxi-v3 Environment Visualization

- **Action 3 (Move West):** Prob: 1.00, Next State: 113, Reward: -1, Done: False
- **Action 4 (Pick Up Passenger):** Prob: 1.00, Next State: 133, Reward: -10, Done: False
- **Action 5 (Drop Off Passenger):** Prob: 1.00, Next State: 133, Reward: -10, Done: False

2 Reward Shaping: Manhattan-Distance Potential Reward Wrapper for Taxi-v3

To enhance learning, I applied a reward shaping method based on Manhattan distance. Here's the gist:

2.1 1. The Potential Function

For a given state s, decoded as

$$(\text{taxi\_row}, \text{taxi\_col}, \text{passenger\_idx}, \text{destination\_idx}),$$

we compute the Manhattan distance:

$$|r_1 - r_2| + |c_1 - c_2|$$

and define:

$$\Phi(s) = -\text{ManhattanDistance}.$$

A smaller distance (closer to the goal) yields a higher (less negative) potential.

2.1.1 2. Reward Shaping

The shaped reward is:

$$r'(s, a, s') = r(s, a, s') + \gamma\,\Phi(s') - \Phi(s),$$

where $r(s, a, s')$ is the original reward and γ (e.g., 0.99) is the discount factor. This extra term rewards the taxi for moving closer to the passenger or destination.

2.1.2 3. Wrapper Overview

The custom wrapper:

- Overrides `reset()` to initialize the potential.
- Overrides `step(action)` to compute the new potential, adjust the reward, and update the stored potential.

2.2 Model Training on Taxi-v3

I originally intended to train five models (A2C, PPO, DQN, QRDQN, and RecurrentPPO) on Taxi-v3. However, the last two models were extremely complex (each taking about 3 hours), so I trained only three models (A2C, PPO, and DQN) in both unwrapped and wrapped settings (6 models total).

2.2.1 Enriched Hyperparameters

The hyperparameters I used for training are summarized in the table below. (Note: I planned for 5 models but dropped the last two due to training time.)

Model	**Learning Rate**	n_steps	γ	**Other Settings**
A2C	0.001	500	0.99	Entropy coef: 0.01
PPO	0.0003	128	0.99	Batch: 64, Epochs: 4, Clip range: 0.2
DQN	0.001	—	0.99	Buffer: 50000, Learn starts: 1000, Target update: 500

2.2.2 Training Times

I ran each training session for 100,000 timesteps. The training times are as follows:

Model (Mode)	**Time (s)**
A2C - no_wrap	121.81 s
A2C - wrapped	107.18 s
PPO - no_wrap	161.98 s
PPO - wrapped	166.18 s
DQN - no_wrap	485.87 s
DQN - wrapped	463.37 s

2.3 Results and Analysis

2.3.1 Learning Curves

After training, I plotted the rolling average rewards (with a window of 50 episodes) for each model. Two sets of figures were generated: one for the unwrapped environment and one for the wrapped environment.

2.3.2 Analysis of Learning Curves

In the unwrapped setting:

- **DQN** performed much better than the others.

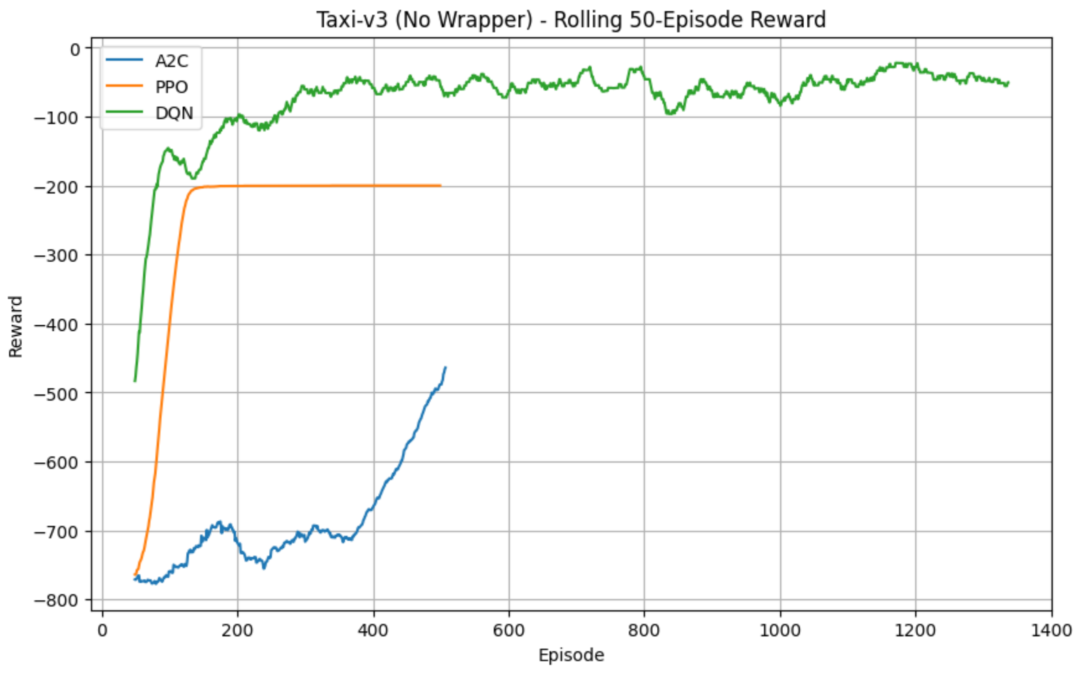

Figure 2: Rolling Average Reward (Unwrapped)

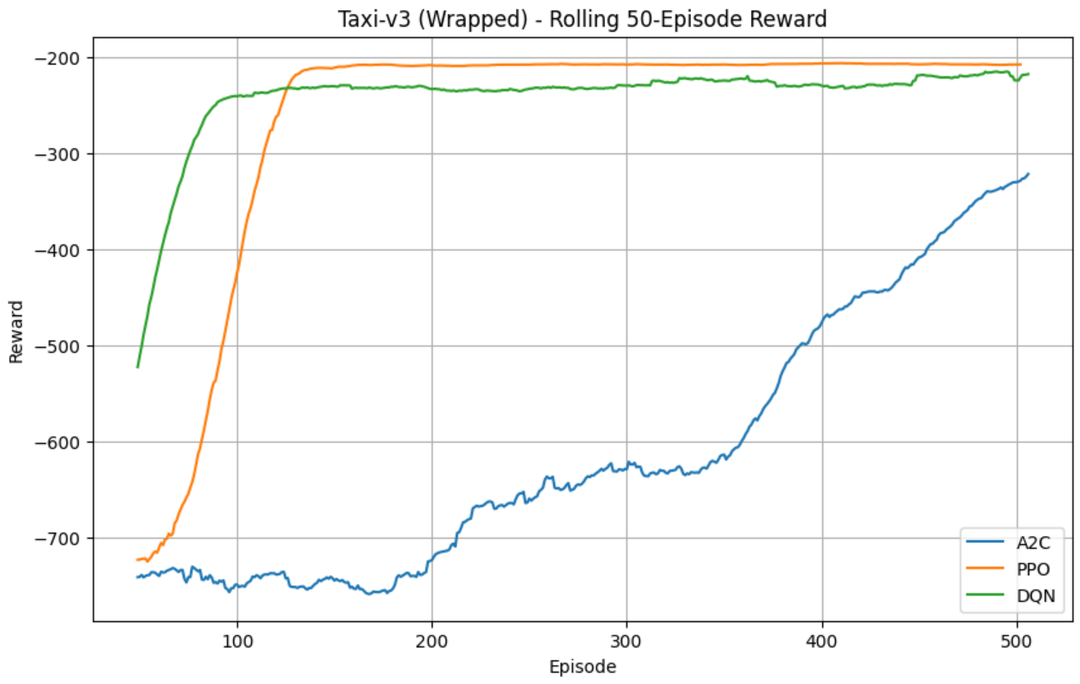

Figure 3: Rolling Average Reward (Wrapped)

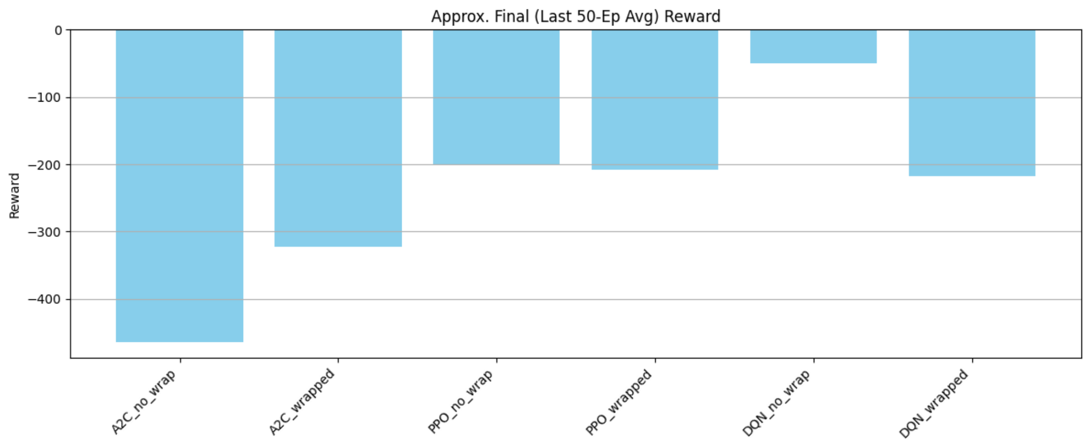

Figure 4: Final Rolling Average Reward Comparison

- **A2C** failed to converge.
- My friends joked that everyone was stuck around -200 reward (similar to PPO), but as you can see, DQN achieved a much better reward.

In the wrapped setting:

- A2C still performed terribly.
- PPO and DQN had very similar performance, with PPO being slightly better overall.

This clearly shows the impact of reward shaping on the performance.

2.3.3 Final Performance and Sample Efficiency

I also created a bar chart comparing the final rolling average rewards and another showing the episode at which each model's reward first exceeded the threshold of -100.

2.4 Model Testing and Visualization

I tested each of the 6 models by running a single test episode in Taxi-v3. During testing, I captured the rendered frames and compiled them into GIFs. These GIFs (visible inline in the notebook) provide a clear view of the learned policies.

2.5 Thoughts on the Fitness of SL for RL Problems

Supervised learning (SL) works great for tasks with abundant labeled data, but in RL settings like Taxi-v3 (and FrozenLake) the goal is to learn a policy through trial and error with sparse and delayed rewards. In these tasks:

- **Sequential Decision-Making:** RL requires learning long-term dependencies and temporal credit assignment, which SL isn't designed to handle.
- **Exploration:** RL must balance exploration and exploitation, whereas SL models work on fixed datasets.
- **Data Quality:** RL environments provide sparse, delayed rewards rather than clear labels, making SL approaches impractical.

- **Adaptability:** In dynamic environments, a static SL model might fail to adapt, while RL continuously learns from interaction.

Thus, while SL is very powerful in many domains, its direct application to RL problems is limited. Specialized RL techniques are required to effectively manage sequential, dynamic, and exploratory decision-making.

2.6 Summary of Findings

The experiments on Taxi-v3 show that:

- In the unwrapped setting, DQN significantly outperformed the other models, while A2C did not converge.
- In the wrapped setting, A2C remained poor, but PPO and DQN achieved nearly identical performance, with PPO having a slight edge.
- Final performance, training times, and sample efficiency clearly indicate that reward shaping had a noticeable impact on the models' performance.

Finally, you can view the GIFs generated during testing to see the learned policies in action.

After completing our initial experiments on Taxi-v3, I moved on to experiment with the FrozenLake environment. I selected FrozenLake for its simplicity and the challenge of sparse rewards. I planned to test five different RL models (A2C, PPO, DQN, QRDQN, and RecurrentPPO) on FrozenLake. However, due to the complexity and lengthy training times of the last two models (each taking around 3 hours), I ended up training only three models (A2C, PPO, and DQN) in both unwrapped and wrapped settings (i.e., 6 models in total). Also, the training files were huge and I generated many GIFs for visualization—while the GIF files sometimes failed on Colab, they all display properly within the notebook.

Without further delay, let's get into the details of FrozenLake.

2.7 FrozenLake Environment Details

The FrozenLake environment (v1) is a classic toy problem defined on a grid. Key details are:

- **Observation Space:** `Discrete(16)` (for a 4x4 grid)
- **Action Space:** `Discrete(4)`
- **State Encoding:** Each state is represented as an integer, which can be decoded into grid coordinates as:

$$\text{row} = \text{state} \div \text{grid\_size}, \quad \text{col} = \text{state} \mod \text{grid\_size}$$

- **Action Mapping:**
  - 0: Left
  - 1: Down
  - 2: Right
  - 3: Up

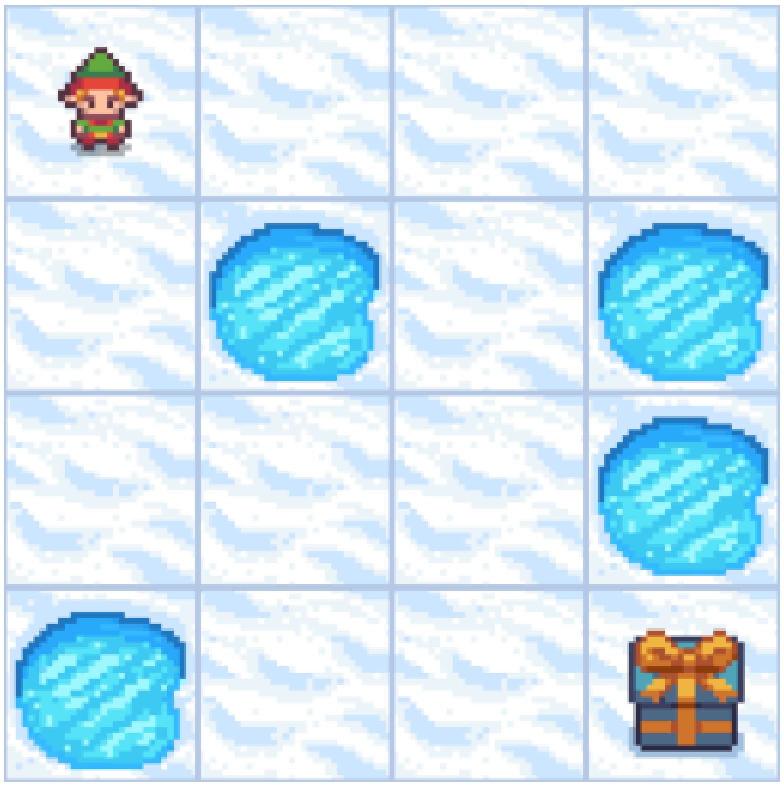

Figure 5: FrozenLake Environment Visualization

2.8 Reward Shaping: FrozenLakePotentialWrapper Using Manhattan Distance

To enhance learning, I applied a potential-based reward shaping method based on Manhattan distance. This wrapper is designed to provide extra feedback by calculating the negative Manhattan distance between the current state and the goal.

2.8.1 1. The Potential Function

For a given state s (an integer), we decode it into grid coordinates:

$$\text{row} = \frac{s}{\text{grid\_size}}, \quad \text{col} = s \mod \text{grid\_size}$$

The potential is defined as:

$$\Phi(s) = -\Big(|\text{row} - \text{goal}_{\text{row}}| + |\text{col} - \text{goal}_{\text{col}}|\Big)$$

A state closer to the goal yields a higher (less negative) potential.

2.8.2 2. Reward Shaping Formula

The wrapper adjusts the reward as follows:

$$r'(s, a, s') = r(s, a, s') + \gamma\,\Phi(s') - \Phi(s)$$

Here, $r(s, a, s')$ is the original reward and γ (e.g., 0.99) is the discount factor. This term rewards the agent for moving closer to the goal.

2.8.3 3. Wrapper Overview

The custom wrapper:

- Overrides `reset()` to initialize the potential.
- Overrides `step(action)` to compute the new potential, adjust the reward, and update the stored potential.

2.9 Model Training on FrozenLake

I planned to train five models (A2C, PPO, DQN, QRDQN, and RecurrentPPO) on FrozenLake. However, due to the extensive training time required for the more complex models, I ended up training only three models (A2C, PPO, and DQN) in both unwrapped and wrapped settings, resulting in 6 models total.

2.9.1 Hyperparameters

The enriched hyperparameters used for training FrozenLake are as follows:

Model	**Learning Rate**	n_steps	γ	**Other Settings**
A2C	7e-4	5000	0.99	—
PPO	2.5e-4	128	0.99	Batch: 64, Epochs: 4
DQN	1e-3	—	0.99	Buffer: 50000, Learn starts: 1000, Train freq: 4

2.9.2 Training Times

I ran each training session for 100,000 timesteps. The training times are summarized below:

Model (Mode)	**Time (s)**
A2C - no_wrap	113.35 s
A2C - wrapped	101.87 s
PPO - no_wrap	141.64 s
PPO - wrapped	142.67 s
DQN - no_wrap	151.88 s
DQN - wrapped	177.52 s

2.10 Results and Analysis

2.10.1 Learning Curves

I generated rolling average reward plots (window = 50 episodes) for each model. In the unwrapped setting, the results indicate that PPO performs significantly better while A2C struggles to converge. In the wrapped setting, PPO remains the best—though the performance gap narrows between PPO and DQN.

2.10.2 Final Performance and Sample Efficiency

Bar charts were created to compare the final rolling average rewards and the episode index when the rolling reward first reached the threshold of 0.0.

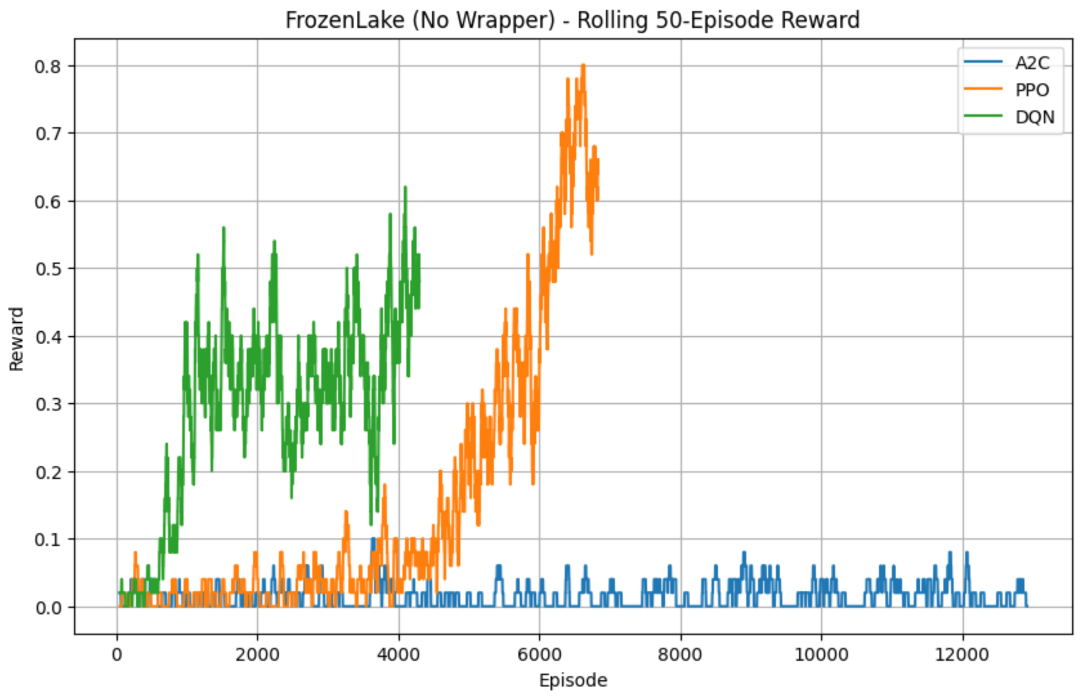

Figure 6: Rolling Average Reward (Unwrapped)

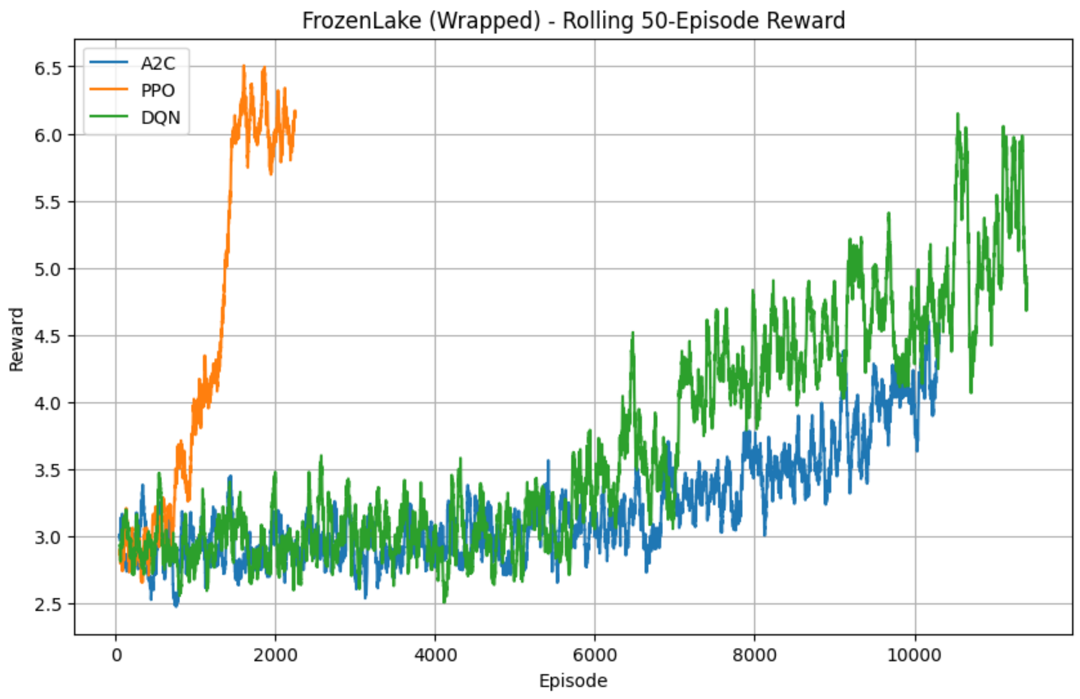

Figure 7: Rolling Average Reward (Wrapped)

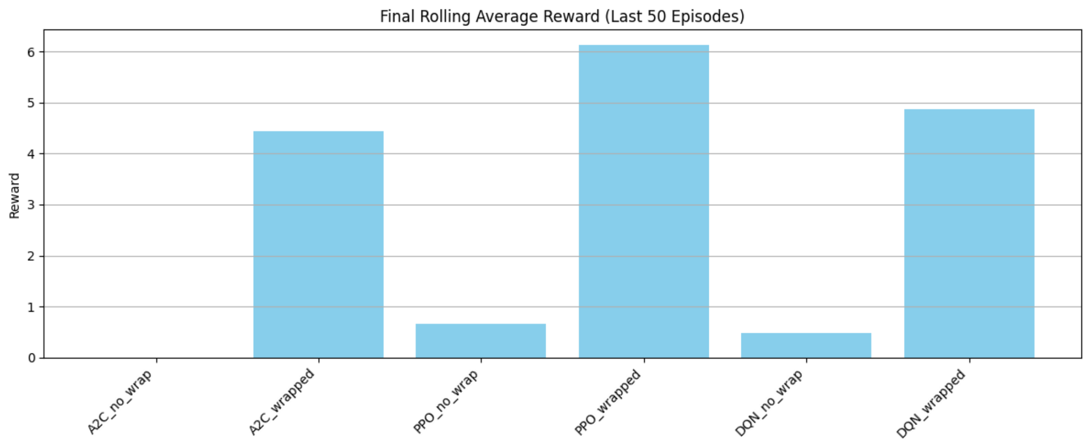

Figure 8: Final Rolling Average Reward Comparison

2.10.3 Analysis

In the ideal (unwrapped) setting, PPO performed much better than the others, and A2C was extremely poor. However, in the wrapped setting, PPO remained the best, and the gap between PPO and DQN became very small. This indicates that reward shaping significantly improves the performance of PPO in this problem, while A2C continues to struggle. All models reached the sample efficiency threshold (0.0) at episode 49.

2.11 Model Testing and Visualization

After training, I tested each of the 6 models on FrozenLake by running a single test episode. During testing, I captured the rendered frames and compiled them into GIFs. These GIFs (viewable inline in the notebook) clearly demonstrate the behavior of the learned policies.

Note: The GIFs are not directly accessible as files, but they can be viewed within the notebook.

2.12 Thoughts on the Fitness of Supervised Learning for RL Problems

Supervised learning (SL) performs very well when large amounts of labeled data are available, such as in image classification or speech recognition. However, in reinforcement learning (RL) tasks like Taxi-v3 and FrozenLake, SL faces several challenges:

- **Sequential Decision-Making:** RL requires learning long-term dependencies and handling temporal credit assignment, which SL—focused on static input-output pairs—cannot naturally manage.
- **Exploration:** RL involves a balance between exploring new actions and exploiting known ones. SL models, trained on fixed datasets, lack mechanisms for exploration.
- **Sparse and Delayed Rewards:** In RL, rewards are often sparse or delayed, making it hard for SL, which depends on explicit labels, to learn the correct actions.
- **Adaptability:** RL environments are dynamic and non-stationary, while SL models can overfit to a static dataset and fail to adapt to new situations.

In summary, while SL excels in domains with abundant labeled data, its inability to handle sequential, exploratory, and adaptive decision-making makes it less practical for RL problems.

3 Task 2: Creating Custom Environments [45-points]

This report presents the design, implementation, and evaluation of a custom 4x4 GridWorld environment with obstacles. The environment is constructed in such a way that two cells are blocked, making the navigation task non-trivial. Based on extensive experimental experience from previous projects, two reinforcement learning (RL) algorithms—Proximal Policy Optimization (PPO) and Deep Q-Network (DQN)—were chosen for evaluation. Their hyperparameters were tuned, and the models were compared based on total reward evolution, sample efficiency, and training time. Various plots and visualizations, including test-time GIF snapshots, support the conclusions drawn in this work.

3.1 Introduction and Environment Setup

In this work, we propose a custom-built 4x4 GridWorld environment that is both simple in design and sufficiently challenging due to the inclusion of obstacles. The primary goal of this environment is to test and evaluate reinforcement learning algorithms in a controlled yet non-trivial setting. As shown in Figure 9, the environment consists of a 4x4 grid where the agent starts at the top-left corner and must reach the goal at the bottom-right corner. However, the cells at positions (1,1) and (2,2) are deliberately blocked, which prevents the agent from entering them. This design choice introduces a significant planning challenge since the agent must avoid these cells to successfully navigate the grid.

The environment is implemented using the Gymnasium API and is equipped with two rendering modes:

- **Human Mode:** This mode prints a text-based grid where different symbols represent the agent (A), the goal (G), blocked cells (X), and free cells (-).
- **RGB Array Mode:** In this mode, the environment is rendered as an image. Each grid cell is displayed as a 50x50 pixel block with different colors: blue for the agent, green for the goal, gray for blocked cells, and white for empty cells.

Figure 9 shows an overview of the environment. In the figure, you can clearly see the layout of the grid with obstacles and the goal position.

3.2 Model Training and Hyperparameter Tuning

Based on insights from previous experiments, I decided to evaluate the performance of two reinforcement learning algorithms: PPO and DQN. Both of these algorithms have been successfully applied to various environments; however, their performance can vary depending on the specifics of the task.

For this experiment, the models were trained using the following hyperparameters:

These hyperparameters were selected after careful experimentation and tuning. They are presented in the form of a table (Table 2) for clarity. The PPO algorithm uses a smaller learning rate and employs multiple epochs per update, while DQN relies on a larger replay buffer and starts training after accumulating a sufficient number of samples.

3.3 Experimental Results and Analysis

The performance of both PPO and DQN was assessed using multiple metrics. The following subsections provide detailed discussions on each metric along with the corresponding figures.

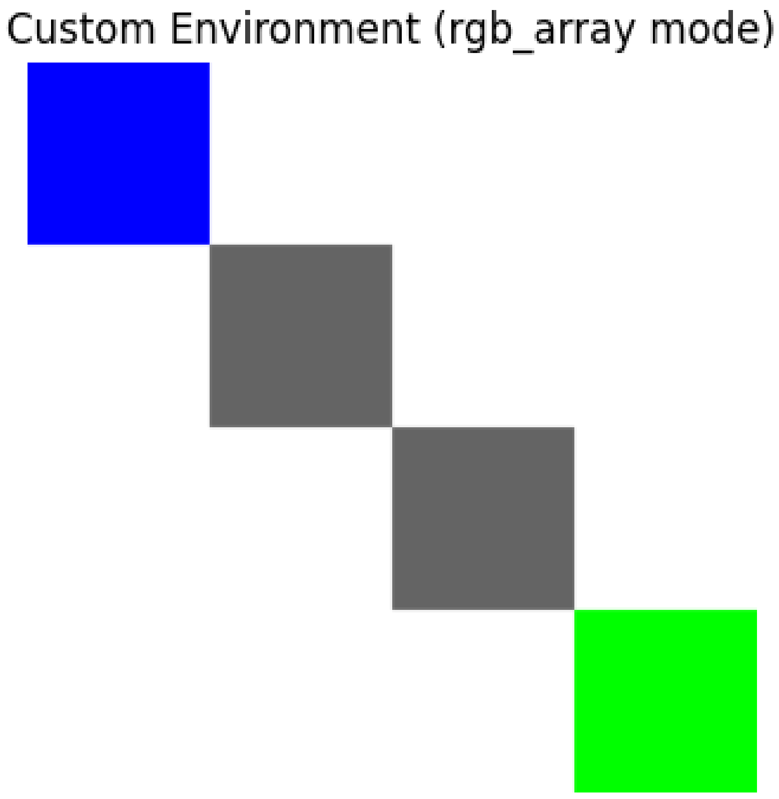

Figure 9: Overview of the custom 4x4 GridWorld environment. The grid demonstrates the starting position of the agent, the goal position, and the blocked cells at (1,1) and (2,2).

Algorithm	**Hyperparameters**
PPO	verbose: 1
	policy: MlpPolicy
	learning_rate: 2.5e-4
	n_steps: 128
	batch_size: 64
	n_epochs: 4
DQN	verbose: 1
	policy: MlpPolicy
	learning_rate: 1e-3
	buffer_size: 50000
	learning_starts: 1000
	train_freq: 4

Table 1: Hyperparameter settings used for PPO and DQN models.

3.3.1 Reward Convergence

The first aspect of evaluation was the convergence of the total reward over time. Figure 10 illustrates the rolling average reward over the training episodes for both PPO and DQN. As can be seen from the plot, both models eventually reach the optimal reward. However, PPO exhibits a smoother convergence curve and attains the optimal reward much earlier compared to DQN. This suggests that PPO is more sample-efficient in this environment.

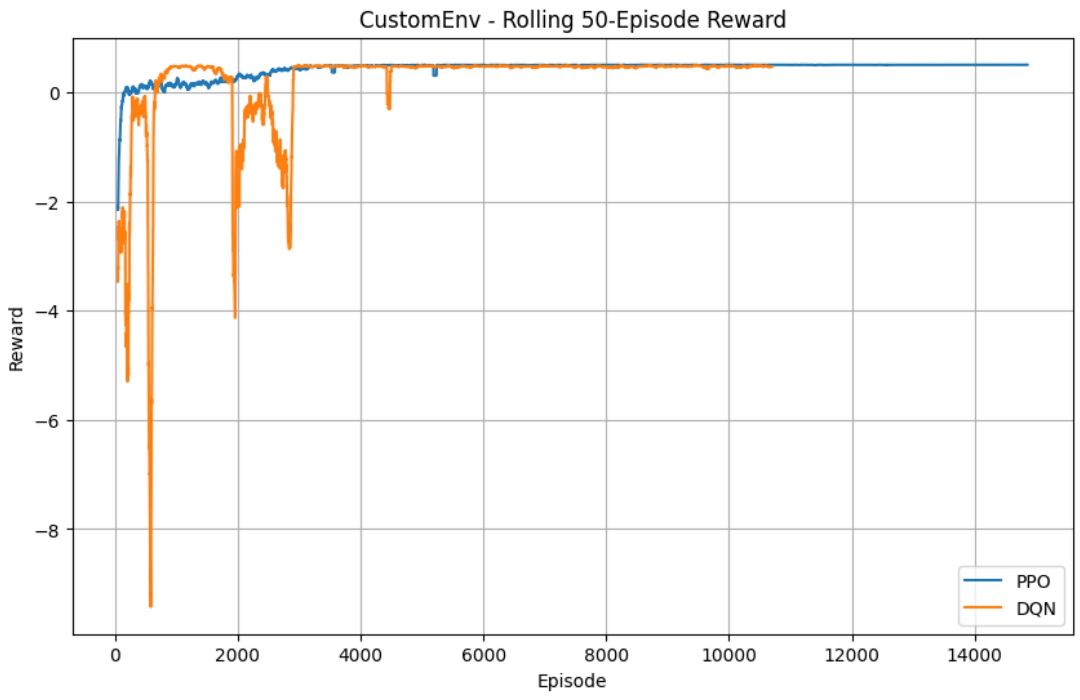

Figure 10: Rolling average reward over episodes for both PPO and DQN. Note that PPO converges more smoothly and faster than DQN.

3.3.2 Final Rolling Average Reward

Figure 11 presents a bar chart comparing the final rolling average reward (computed over the last 50 episodes) for both algorithms. The results confirm that both PPO and DQN achieve nearly identical optimal performance levels by the end of the training phase.

3.3.3 Training Time Comparison

Training times for both models were also recorded and compared. Figure 12 shows a bar chart of the training times. Both algorithms required a similar amount of training time, with DQN being slightly faster. This minor difference in training time may be attributed to the inherent differences in algorithmic complexity and update frequency.

3.3.4 Sample Efficiency

The sample efficiency of each algorithm was determined by evaluating the episode at which a rolling average reward threshold of 0.0 was reached. The results are as follows:

```
=== Sample Efficiency ===
```

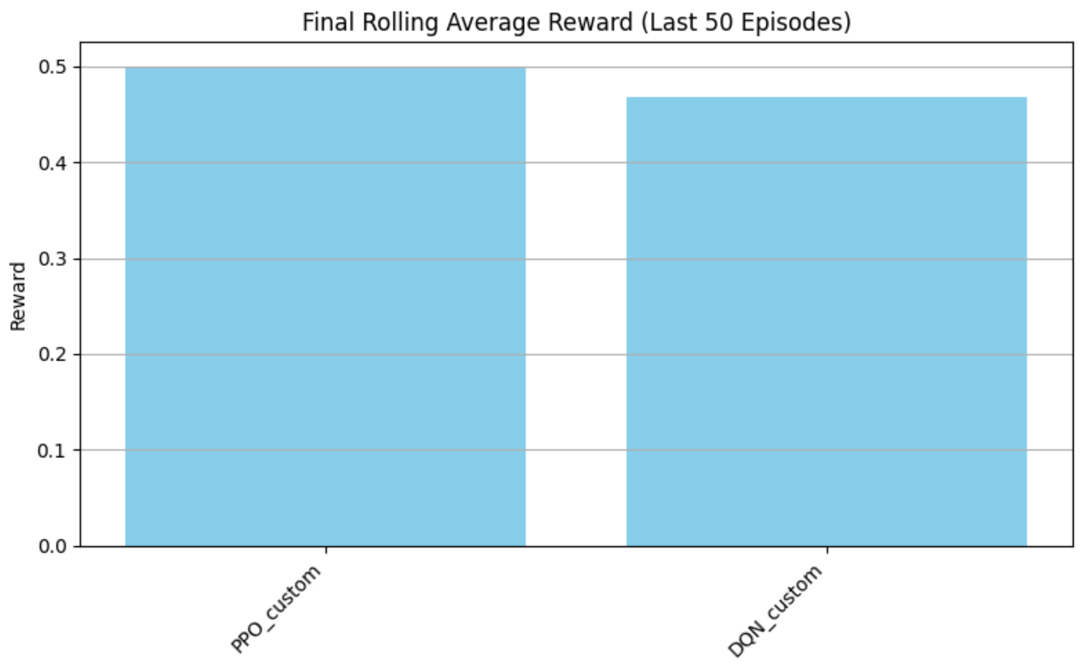

Figure 11: Bar chart of the final rolling average reward (last 50 episodes) for PPO and DQN. Both models reach optimal performance, although PPO shows a more consistent reward progression.

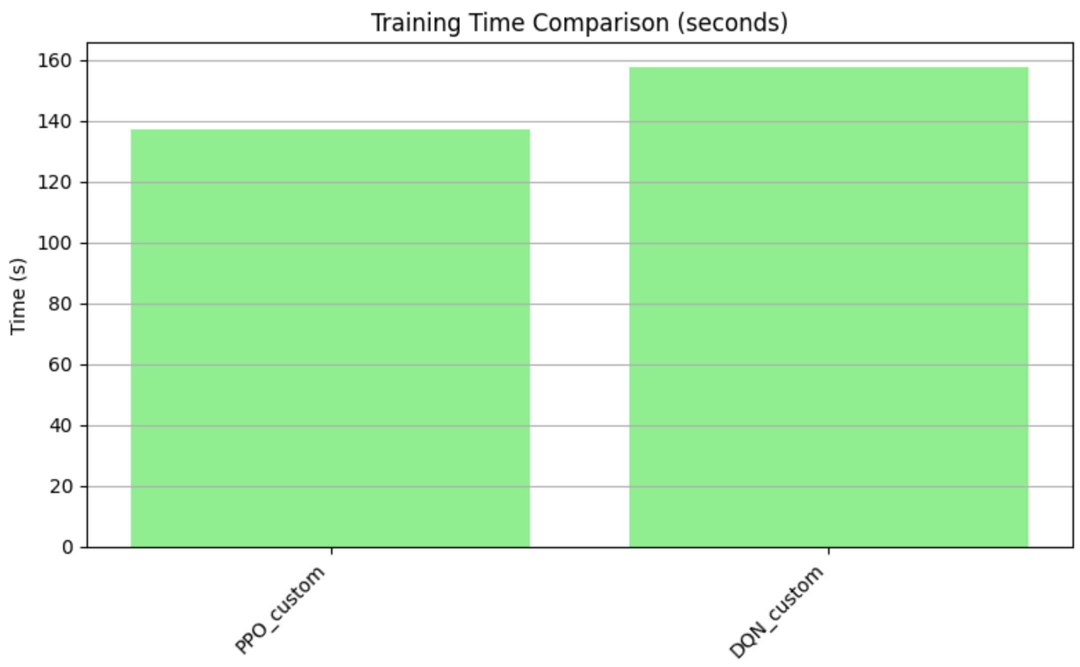

Figure 12: Comparison of training times for PPO and DQN. Both models exhibit similar training durations with DQN having a slight edge in speed.

```
Threshold = 0.0
PPO_custom: reached threshold at episode 158.
DQN_custom: reached threshold at episode 654.
```

This analysis shows that PPO reached the reward threshold significantly earlier than DQN, indicating that PPO is more efficient in learning from the environment. The earlier convergence of PPO is a clear advantage in scenarios where sample efficiency is critical.

3.4 Test-Time Visualization

To further validate the learned policies, test-time visualizations were generated in the form of GIFs. These GIFs capture a complete test episode where the trained models navigate the environment. As expected, both models successfully steer the agent to the goal while avoiding blocked cells. Figure **??** displays a representative snapshot from the test-time GIF, clearly illustrating the optimal path followed by the agent.

3.5 Conclusion

In this report, I presented a detailed evaluation of a custom-designed 4x4 GridWorld environment featuring blocked cells at (1,1) and (2,2). Two RL algorithms, PPO and DQN, were trained on this environment using carefully tuned hyperparameters (see Table 2). The experimental results indicate that although both models eventually reach optimal performance, PPO converges more smoothly and requires significantly fewer episodes to reach the reward threshold. In contrast, DQN, while slightly faster in training time, takes longer to achieve optimal reward levels. Test-time visualizations further confirm that both algorithms are capable of guiding the agent to the goal; however, the efficiency and consistency of PPO make it a more favorable choice for this task.

4 Task 3: Pygame for RL environment [20-points]

4.1 Problem Definition and Environment Implementation

The objective of this project is to create a custom reinforcement learning (RL) environment based on the Chrome Dino game. I began by designing and implementing the complete game using Pygame. The game faithfully reproduces the core mechanics including running, jumping, ducking, and obstacle management. After developing a fully functional game, I converted it into an RL environment compatible with Gymnasium. The observation space is defined as an RGB image of the game screen, and the action space consists of three discrete actions. A key feature of this environment is that it generates a GIF of the gameplay every time it is run in Jupyter Notebook, allowing for visual inspection of the agent's behavior. Figure 13 shows a snapshot of the game before any actions are taken.

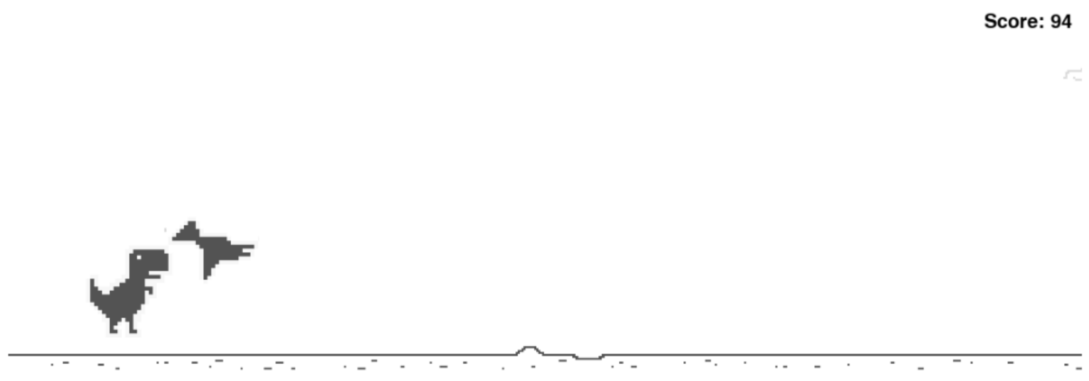

Figure 13: Screenshot of the game environment in its initial state, before any actions are taken.

Once integrated as an RL environment, it functions equivalently to standard pre-built environments, allowing any RL algorithm to be applied without additional modifications.

4.2 Baseline Behavior and Environment Testing

Before training any models, I verified that the environment operated correctly. Running the environment produces a GIF that captures the full gameplay progression from the beginning until failure. This baseline behavior confirms that the environment behaves like a real game. Figure 14 illustrates a snapshot from the baseline GIF generated when the game is played without any RL agent actions.

4.3 RL Model Training

With the environment fully operational, I proceeded to train RL agents using two algorithms: Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C). The training was performed using the `CnnPolicy` since the observations are images. The hyperparameters used for training are summarized in Table 2. Given the realistic, real-time nature of the environment, each training episode proceeds at the same pace as playing the game manually, making the data collection process extremely slow.

Due to the environment's real-time simulation, training takes a very long time. The PPO model exhibits variable performance, with rewards fluctuating significantly, while the A2C model remains almost constant at a low reward.

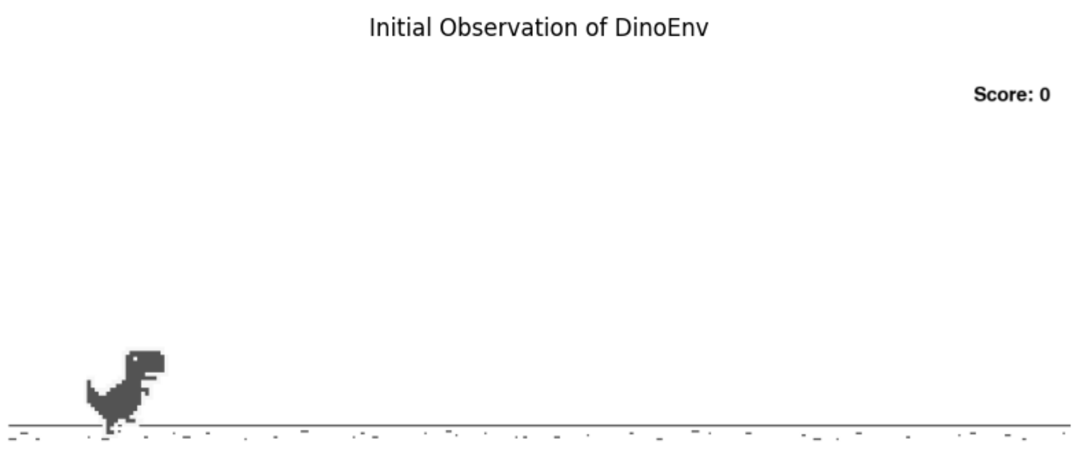

Figure 14: Snapshot from a GIF generated by the environment in its baseline mode.

Table 2: Hyperparameters for PPO and A2C (using `CnnPolicy`).

Parameter	PPO	A2C
Policy	CnnPolicy	CnnPolicy
Learning Rate	0.0003	0.0007
n_steps	128	5
Batch Size	64	–
n_epochs	4	–
Gamma	0.99	0.99
Clip Range	0.2	–
Verbose	1	1

4.4 Training Analysis and Results

After training, I analyzed the performance of both models using logged data. The results were as follows:

PPO Training Analysis:
Total Episodes: 35
Mean Reward: 28.03
Median Reward: 7.00
Reward Std Dev: 63.26
Total Training Time: 1944.27 seconds

A2C Training Analysis:
Total Episodes: 53
Mean Reward: -7.00
Median Reward: -7.00
Reward Std Dev: 0.00
Total Training Time: 2255.48 seconds

Figures 15 and 16 display the learning curves for PPO and A2C, respectively. The PPO agent shows high variability in rewards, whereas the A2C agent remains at a consistently low level.

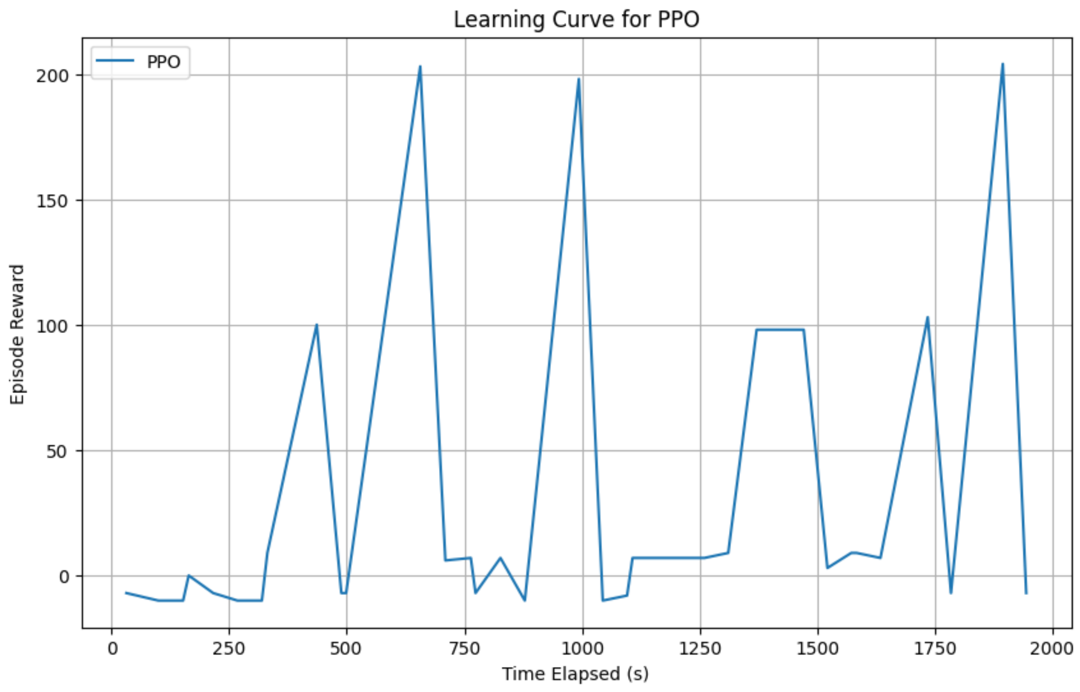

Figure 15: Learning curve for the PPO agent.

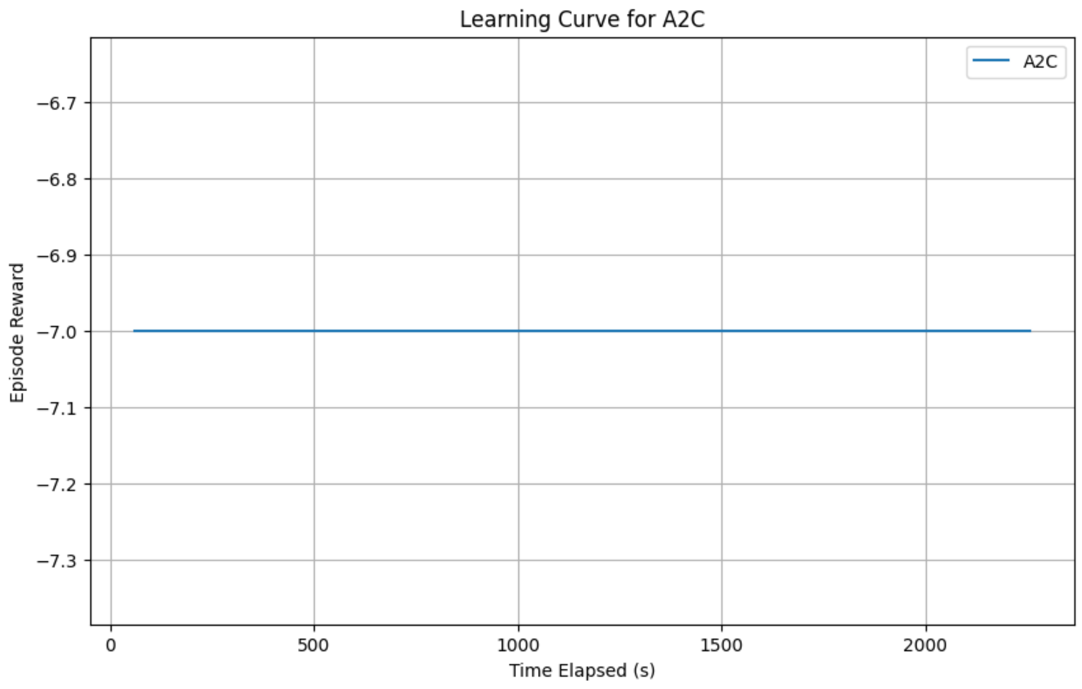

Figure 16: Learning curve for the A2C agent.

4.5 Testing of Learned Policies

Following training, both models were tested in the environment. The learned policy for each model was executed in a fresh test episode, during which the environment captured gameplay and generated a GIF.

As a result, one can visually inspect the agent's behavior. The PPO policy tends to cause the dinosaur to jump frequently, while the A2C policy keeps the dinosaur in a ducking state with minimal variation.

4.6 Discussion

The project demonstrates that a custom RL environment, even one built with the realistic dynamics of a Pygame simulation, can be effectively used for training RL agents. However, the key challenge lies in the slow pace of the environment. Since each step corresponds to an actual frame update as in a real game, data collection is extremely time-consuming. This has a significant impact on training efficiency, as evidenced by the learning curves and analysis results. The PPO agent exhibits high variability with no clear convergence, while the A2C agent's performance remains stagnant. Ultimately, the results suggest that while the environment is fully functional and comparable to pre-built RL environments, further optimization or alternative simulation methods may be necessary to expedite training.

4.7 Conclusion

In conclusion, I have developed a custom RL environment for the Dino game using Pygame and integrated it with Gymnasium. The environment operates in real time and supports visual inspection through GIF generation. Training was carried out using PPO and A2C with the hyperparameters summarized in Table 2. Although the training process is extremely slow due to the realistic simulation, the environment behaves like a standard RL environment and can be used for further research. Testing of the learned policies revealed distinct behaviors: the PPO policy frequently triggers jumps, whereas the A2C policy results in continuous ducking. Future work should focus on optimizing the environment for faster data collection, potentially by eliminating the reliance on Pygame, thereby enabling more effective training.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, 2020. Available online: http://incompleteideas.net/book/the-book-2nd.html

[2] A. Raffin et al., "Stable Baselines3: Reliable Reinforcement Learning Implementations," GitHub Repository, 2020. Available: https://github.com/DLR-RM/stable-baselines3.

[3] Gymnasium Documentation. Available: https://gymnasium.farama.org/.

[4] Pygame Documentation. Available: https://www.pygame.org/docs/.

[5] CS 285: Deep Reinforcement Learning, UC Berkeley, Pieter Abbeel. Course material available: http://rail.eecs.berkeley.edu/deeprlcourse/.

[6] Cover image designed by freepik