IDAT2502 Project in Applied Machine Learning

Emil Slettbakk *emilsle@stud.ntnu.no*
Norwegian University of Science and Technology,
Department of Computer Science,
NO-7491 Trondheim, Norway

John Ivar Eriksen *jierikse@stud.ntnu.no*
Norwegian University of Science and Technology,
Department of Computer Science,
NO-7491 Trondheim, Norway

Abstract

This report evaluates the performance of two implementations[1] of reinforcement learning algorithms, Proximal Policy Optimization (PPO) and Double Deep Q-Network (DDQN), on the Gym Super Mario Bros. environment. The study aimed to highlight the strengths and weaknesses of both algorithms by training for 30 000 episodes and testing on 1 000 episodes. DDQN showed steady learning, while PPO struggled, highlighting challenges in its design for discrete tasks. Recommendations for addressing these issues and future directions are discussed.

Acronyms

Below is a list of frequently used acronyms in this report.

PPO	Proximal Policy Optimization.
DQN	Deep Q-Network.
DDQN	Double Deep Q-Network.
RL	Reinforcement learning.
CPI	Clipped Surrogate Objective.
AI	Artificial Intelligence.
LLM	Large Language Model.
SMB	Super Mario Bros.

1 Introduction

Reinforcement learning (RL) has proven highly effective in training agents for complex, sequential decision-making tasks. Video game environments, especially ones with dynamically changing obstacles and reward structures like Super Mario Bros. (SMB), offer a solid framework for testing RL algorithms.

Among the numerous RL algorithms available, Proximal Policy Optimization (PPO) and Double Deep Q-Network (DDQN) represent two widely studied approaches. PPO, a policy-gradient method, is optimized for continuous control tasks but is often adapted for discrete tasks as well. DDQN, a value-based approach, is particularly efficient in discrete environments, as it refines traditional Q-learning to better handle overestimation bias.

This study compares policy-gradient and value-based methods in a discrete task. This is done to show how different RL algorithms will perform differently at the same task.

[1] The source code for the project can be found at `https://github.com/buhund/idatt2502-assignment_09-ml_project`

2 Related Work

Proximal Policy Optimization (PPO) is a relatively recent addition to policy gradient methods in reinforcement learning, contributing significantly to recent advances, including those seen in large language models (LLM's) and reinforcement learning (RL). Introduced by John Schulman (2017b), PPO addresses limitations of earlier methods by balancing simplicity, data efficiency, and performance. The key innovation of PPO lies in the use of a **clipped surrogate objective**, defined as:

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

where $r_t(\theta)$ represents the probability ratio, $\hat{A}_t$ is the advantage estimate, and ϵ is a hyperparameter constant. This objective ensures policy stability by penalizing large updates that could otherwise destabilize training, while enabling multiple epochs of mini-batch updates[2].

Unlike Trust Region Policy Optimization (TRPO), a predecessor to PPO which enforces trust regions to limit policy updates, PPO avoids the complexity of second-order optimization while achieving comparable or superior performance across a wide range of tasks. John Schulman (2017b) demonstrated that PPO excels in continuous control benchmarks, and performs competitively in discrete environments like Atari games, particularly when considering its simplicity and ease of implementation. These qualities have established PPO as a benchmark algorithm in RL research and applications.

Double Deep Q-Network (DDQN) is an enhancement of the Deep Q-Network (DQN) algorithm, developed to tackle the issue of overestimation bias often seen in Q-learning. Traditional DQN, despite its effectiveness in various tasks, can overestimate action values, potentially leading to suboptimal policies. As introduced by Hado van Hasselt (2015), DDQN reduces this bias by decoupling the selection and evaluation of actions. In DDQN, the action selection is based on the main network, while the evaluation of that action is performed by a separate target network. This separation effectively decreases overestimation, yielding more stable and reliable training results. The update rule for DDQN is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q\left(s', \arg\max_a Q(s',a;\theta); \theta^-\right) - Q(s,a) \right]$$

In this formula, r represents the reward received after taking action a in the current state s, and s' is the next state reached after taking action a. The term $Q(s', \arg\max_a Q(s',a;\theta);\theta^-)$ captures the Q-value of the action selected by the main network (parameterized by θ) in the next state s', but evaluates it using the target network (parameterized by θ^-). Here, α is the learning rate, controlling the extent of the update, and γ is the discount factor, which balances the immediate and future rewards.

Several prior studies have evaluated PPO and DDQN in gaming environments, e.g. Jain (2023); Nguyen (2022), and found both algorithms to perform quite well. However, these studies often lack detailed versioning, posing challenges for replication.

3 Methods

The project uses Python and its machine learning library PyTorch, with environments provided by the Gym library. Specifically, the `Gym Super Mario Bros` was selected as the testing environment for this study. Due to resource constraints with regards to available computational equipment, only the first level of the game, `SuperMarioBros-1-1-v0`, was used for training and evaluation.

Early challenges included ensuring compatibility between dependencies and Python versions. To ensure reproducibility, the `requirements.txt` file includes the key dependencies used in the project.

The training data consisted of 30 000 episodes on both algorithms, sped up with `frame_stack`. This makes the agent observe fewer intermediate states, while receiving the accumulated reward of the actions, allowing

[2]The clipped surrogate objective modifies the probability ratio to lie within $[1-\epsilon, 1+\epsilon]$, allowing PPO to combine frequent updates with computational efficiency. For more details, refer to John Schulman (2017b).

for significantly faster training, with minimal loss in agent learning. Refer to Table 1 for the environment setup.

Env parameter	Value	Description
`env_name`	`gym_super_mario_bros`	Environment name
`world`	`1`	SMB world
`stage`	`1`	SMB stage
`env_version`	`v0`	SMB environment version
`actions`	`SIMPLE_MOVEMENT`	Complexity of movement available to the agent
`num_ep_training`	30 000	Number of episodes for **training** the agent
`num_ep_testing`	1 000	Number of episodes for **testing** the agent
`frame_stack`	4	Number of frames to stack/skip (no skip = 1)[3]

Table 1: Environment parameters for Gym Super Mario Bros.

The Hyperparameters are used for tuning the agent to perform in a selected environment, and their optimal values will differ between algorithms because each algorithms will use the parameters for different purposes. For example, the epsilon value in DDQN is dynamic and decreases throughout training to reduce exploration and increase exploitation of learned behaviors, while in PPO epsilon (`clip_rate`) is a constant that defines the bounds withing which the policy updates are considered valid. Specifically, `clip_rate` in PPO ensures that the policy's probability ratio does not deviate too much from its previous value, thereby stabilizing training and preventing overly large updates that could destabilize learning.

The hyperparameters used for PPO and DDQN, are summarized in Table 2 and Table 3, respectively.

Hyperparameter for PPO	Value	Description
`learning_rate`	`1e-4`	Learning rate (0.0001)
`clip_rate`	0.20	Epsilon value (constant)
`gamma`	0.90	Discount factor
`gae_lambda`	0.05	Generalized Advantage Estimation discount
`entropy_coef`	0.01	Entropy coefficient to encourage exploration
`vf_coef`	0.50	Value function coefficient in loss calculation

Table 2: Hyperparameters used for PPO.

Hyperparameters for DDQN	Value	Description
`learning_rate`	0.00025	Learning rate
`gamma`	0.9	Discount factor
`epsilon`	1.0	Initial exploration rate (ϵ)
`eps_decay`	0.99999975	Reduction factor in exploration (ϵ)
`eps_min`	0.05	Minimum exploration rate (ϵ)

Table 3: Hyperparameters used for DDQN.

In order to have as basis for comparison, the metrics referenced in Table 4, which are built in to the SMB environment, was selected for logging.

After completing the determined number of training episodes, each agent would be tested across 1 000 episodes. In testing state, any learning would be disabled, ensuring the agent ran only on the knowledge accumulated throughout the training episodes. The metrics to be compared would be recorded and averaged across the 1 000 episodes to ensure that no outliers in the training would have an outsized impact on the comparison.

[3] `frame_stack` is also known as `action_repeat` or `frame_skip`.

Metric	Description
`flag_get`	True = Completed level. False = Did not complete level.
`x_pos`	Progress through the level along x-axis (left-to-right).
`time`	Time spent completing the level.

Table 4: Metrics chosen for comparing DDQN and PPO performance.

4 Results and Discussion

4.1 Results

The training of both algorithms, DDQN and PPO, was conducted over 30 000 episodes on the `Gym SuperMarioBros-1-1-v0` environment. Upon evaluation across 1 000 test episodes, the results showed a clear contrast in performance between the two approaches.

4.1.1 DDQN results

The DDQN implementation demonstrated consistent learning and steady progress throughout the training period. By the end of the training phase, the agent frequently achieved significant horizontal progress (`x_pos`), and occasionally completed the level (**flag_get**). Testing confirmed that the trained DDQN agent was able to navigate the environment effectively, avoiding the early obstacles, using the gained knowledge to handle the increasing complexity as the level progressed. Metrics showed an improvement in `x_pos` and `flag_get` rates across episodes, along with a decrease in average completion time (`time`) as training progressed.

Metric	Reward	Actor Loss	Critic Loss	x_pos	Flag Get	Total Time
Sum	-	-	-	-	28	1 920 345
Average	850.50	-0.018573	2 482.30	768.40	-	65
Median	840.00	-0.018600	2 300.00	760.00	-	40

Table 5: Summary statistics for the metrics recorded during DDQN agent training.

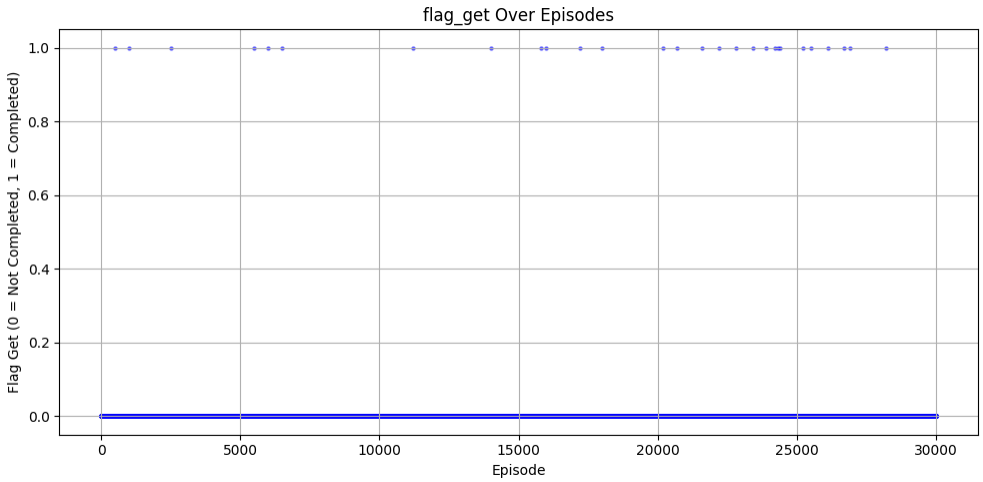

Figure 1: Scatter plot of DDQN `flag_get` values across episodes. This plot shows the distribution of successful level completions (flag achieved) with 28 completions out of 30 000 episodes. This shows that level completion gets progressively better towards the end.

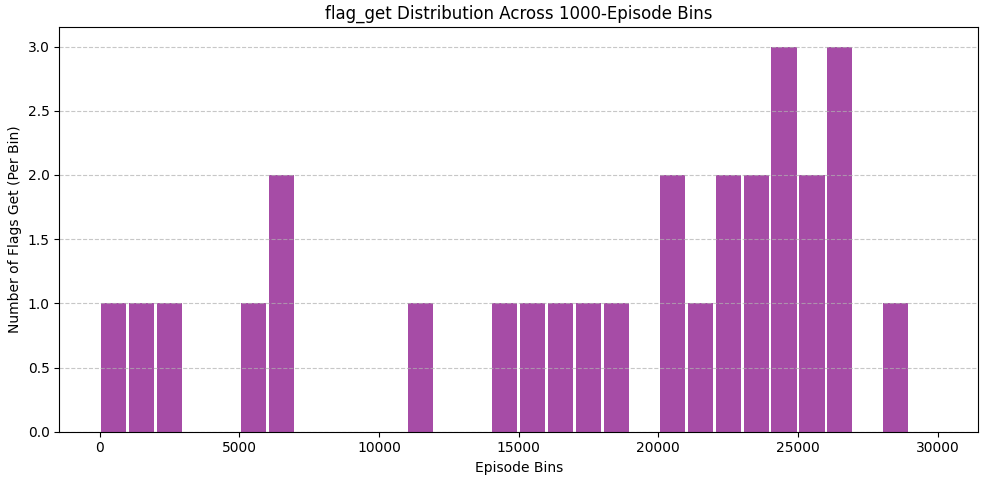

Figure 2: Heat-map plot of DDQN `flag_get` values across episodes, divided into bins of 1 000 episodes, highlighting where the successful runs appeared.

As shown in Figure 1, the distribution demonstrates the agent's gradual improvement in completing levels. Initially, successful completions (flags) are rare, reflecting the agent's struggle with the environment. However, as training progresses, the completions become more frequent, especially toward the later episodes. Figure 2 further illustrates this trend by dividing the 30 000 episodes into bins of 1 000, highlighting an increase in successful runs as the agent gains more experience and improves its navigation skills.

Given this positive trend, it is likely that with extending, the agent would achieve a higher completion rate and exhibit more stable and reliable progress through the level, as it continues to learn and reduce the epsilon value of the algorithm.

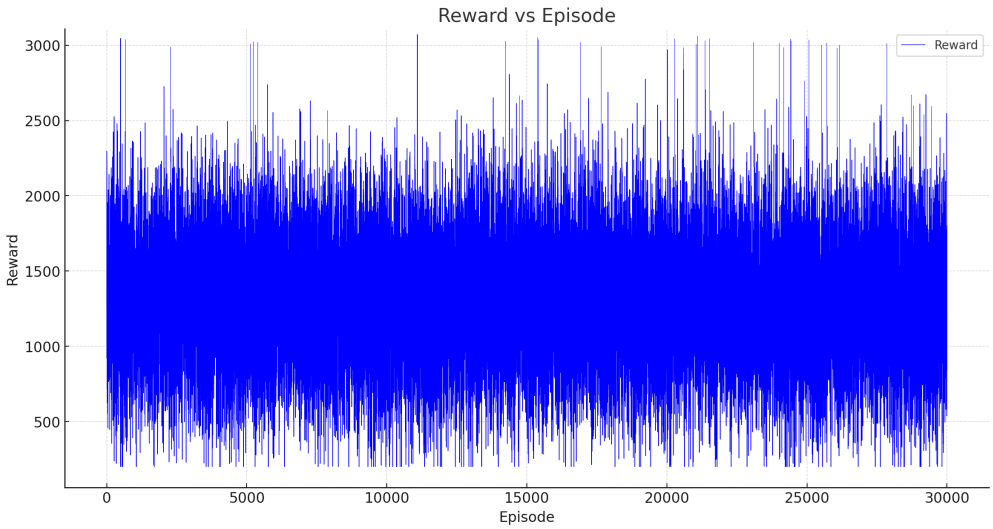

Figure 3: Timeline plot of DDQN reward values across episodes.

4.1.2 PPO results

The PPO implementation, however, showed substantial learning challenges. Despite being trained for the same 30 000 episodes, the agent consistently failed to progress past early obstacles, such as the second pipe or dying to Goombas. Metrics collected during training and testing showed minimal improvement in

`x_pos`, almost non-existent `flag_get` (8 out of 30 000, i.e. 0,03%), and erratic `time` values due to early terminations of the episode. Even after 50,000 episodes, PPO's poor performance continued, suggesting issues in implementation or hyperparameter configuration.

Metric	Reward	Actor Loss	Critic Loss	x_pos	Flag Get	Total Time
Sum	-	-	-	-	8	1 871 467
Average	580.61	-0.021595	2 563.67	743.63	-	62
Median	576.50	-0.021712	2 257.23	720.00	-	34

Table 6: Summary statistics for the metrics recorded during PPO agent training.

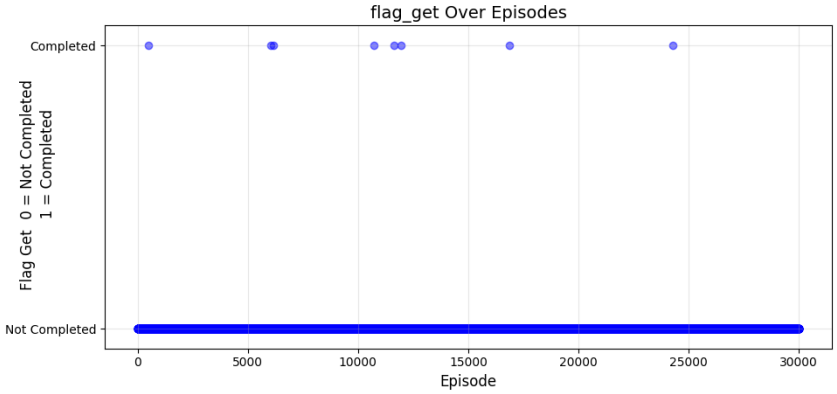

Figure 4: Scatter plot of PPO `flag_get` values across episodes. This plot shows the sparse distribution of successful level completions (flag achieved) with only 8 completions out of 30 000 episodes.

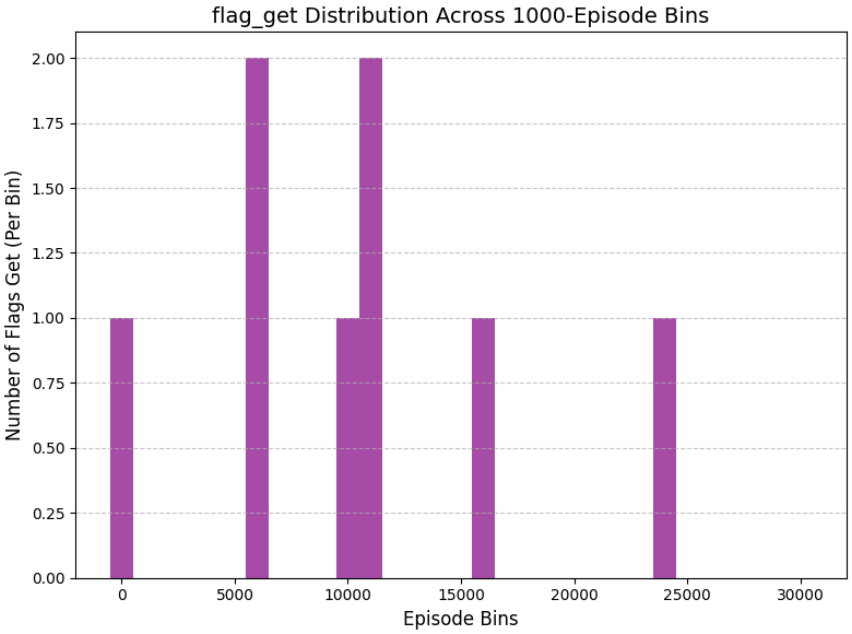

Figure 5: Heat-map plot of PPO `flag_get` values across episodes, divided into bins of 1 000 episodes, highlighting where the successful runs appeared.

As shown in Figure 4, the sparse distribution highlights the agent's struggle to complete levels successfully. Figure 5 show how the successful runs were distributed, by dividing the 30 000 episodes into bins of 1 000 each. From the plot, it does not appear that the successful episodes are a result of accumulated knowledge, since they are spread evenly across the episodes, but rather seems like the agent "got lucky". This is different from the DDQN trend, which show a gradual improvement with completions (flags) becoming more frequent towards the end of the training, as the agent accumulates knowledge.

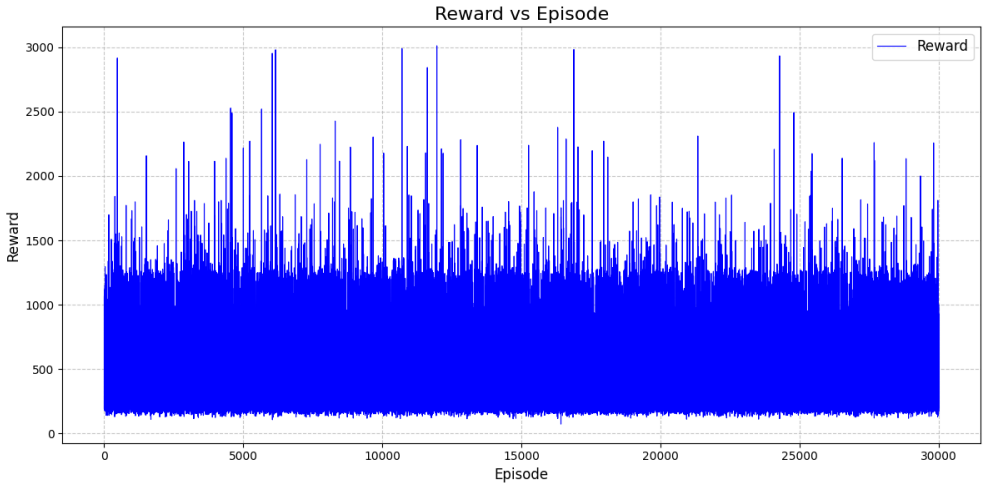

Figure 6: Timeline plot of PPO reward values across episodes.

From Figure 6, it is evident that although occasional spikes suggests improvement, the learning is inconsistent, and the scarcity of higher-reward episodes indicates a significant imbalance in the agent exploration and exploitation strategies. This pattern suggests that either the hyperparameters are inadequately tuned, or that the algorithm itself is not well-suited for the specific challenges posed by the environment. These findings point to the need for further investigation into both model configuration and algorithm suitability.

4.2 Discussion

The results of this study underscore the differences in the suitability and performance of DDQN and PPO in the discrete and dynamic environment of Super Mario Bros. DDQN, by effectively mitigating overestimation bias and balancing exploration and exploitation through epsilon decay, demonstrated clear signs of learning. Although only 28 out of 30 000 episodes were completed, the increasing trend of successes indicates steady progress and suggests that extended training could yield even better results.

PPO, in contrast, struggled to adapt. With only 8 successful completions across 30 000 episodes, its performance appeared erratic. The scattered distribution of successes implies these were likely due to random chance rather than effective learning. PPO's design as a policy-gradient method typically aligns well with continuous control tasks, such as robotic or vehicle movement, where stability in policy updates is advantageous. However, its reliance on the clipped surrogate objective may have hindered adaptability in this discrete task, where precise action selection is crucial. Additionally, the sparse rewards structure of the Super Mario Bros. environment likely compounded these difficulties, offering insufficient feedback for effective optimization.

Hyperparameter sensitivity likely played a significant role in PPO's under-performance. Parameters such as `clip_rate` and `gae_lambda` directly influence the stability and effectiveness of policy updates. Suboptimal tuning of these values likely led to poor generalization and inefficient updates, as evidenced by the erratic reward patterns. Furthermore, the low entropy coefficient (0.01) may have limited exploratory behavior, causing premature convergence to suboptimal policies.

In summary, DDQN's superior performance highlights the importance of aligning algorithms with the environment and task characteristics. Proper tuning of hyperparameters and adapting reward structures to provide more feedback are essential for enabling effective learning.

4.3 Limitations

As previously mentioned, the results in this study is impacted by limited resources, both in terms of computational power for training the agents and in manpower to implement the model, review and optimize the code. Using sub-optimal hardware, running mainly on CPU, restricts the effectiveness of agent training. There were not enough resources to experiment properly with the PPO model's hyperparameters to find better configurations. Some tweaks were attempted, but assessing their effectiveness required hours of training and in general seemed to have negligible impact on the overall learning.

5 Conclusion and Future Work

The experiment showed that Double Deep Q-Network (DDQN) significantly outperformed Proximal Policy Optimization (PPO) in the Gym SMB environment. DDQN converged faster and made steady progress, achieving level completion fairly frequently after around 30,000 episodes. This result suggests that DDQN adapted well to the discrete, obstacle-filled environment. In contrast, PPO showed limited progress, often failing at initial obstacles and potentially diverging. This lack of improvement may be due to PPO's design for continuous control tasks, which may not align well with the discrete challenges in the Mario environment.

A key limitation of this project was access to computer hardware optimized for reinforcement learning, as well as a small team size. These constraints restricted the ability to thoroughly tune hyperparameters or explore alternative configurations that could have improved PPO's performance. With additional resources, it would have been possible to experiment further with algorithm parameters, test more reinforcement learning models, and run evaluations across multiple levels, potentially revealing configurations where PPO might perform more effectively.

Future work could extend training or test additional levels to assess adaptability. Further tuning of PPO's settings, such as reward structure and exploration strategies, might help it better tackle the game's discrete obstacles. Enhanced logging, including loss tracking and obstacle-specific metrics, could also provide a clearer view of each agent's strengths and weaknesses. Additionally, testing other RL algorithms, like Asynchronous Advantage Actor-Critic (A3C) or Temporal Difference (TD), could offer deeper insight into the strengths and limitations of different approaches.

References

David Silver Hado van Hasselt, Arthur Guez. Deep reinforcement learning with double q-learning. *Cornell University, arXiv*, 2015. URL `https://arxiv.org/abs/1509.06461`.

Sanyam Jain. Ramario: Experimental approach to reptile algorithm – reinforcement learning for mario. *Cornell University, arXiv*, 2023. URL `https://arxiv.org/abs/2305.09655`.

Filip Wolski Prafulla Dhariwal Alec Radford John Schulman, Oleg Klimov. Proximal policy optimization (article), 2017a. URL `https://openai.com/index/openai-baselines-ppo/`.

Filip Wolski Prafulla Dhariwal Alec Radford John Schulman, Oleg Klimov. Proximal policy optimization algorithms (paper). *Cornell University, arXiv*, 2017b. URL `https://arxiv.org/abs/1707.06347`.

Viet Nguyen. Mario ppo, 2022. URL `https://github.com/vietnh1009/Super-mario-bros-PPO-pytorch`.