COMP 579 : Reinforcement Learning Project Report

Yassir Mamouni
DIRO/Mila
McGill ID : 261152062
University of Montreal
yassir.mamouni@umontreal.ca

Santino Nanini
DIRO/Mila
McGill ID : 261152063
University of Montreal
santino.nanini@umontreal.ca

Vincent Commere-Ribourg
DIRO/Mila
McGill ID : 261165485
University of Montreal
vincent.commere-ribourg@umontreal.ca

Abstract

This project is inspired by the "Train a Mario-playing RL agent" [1] presented on the PyTorch website. The tutorial trains a reinforcement learning agent on the Super Mario Bros environment[3] so that it can play the game correctly.

The initial project implements a Double Deep-Q Network [8] agent. This agent learns its interactions with the environment, with the goal of maximizing the reward received for completing the level.

In this project, we re-implemented another agent, based on the Policy-Gradient algorithm. This agent is taken from *A Simple Guide To Reinforcement Learning With The Super Mario Bros Environment*[6].

The Super Mario Bros environment offers many visual modifications of its environment. However, both agent's algorithms mentioned above were trained and evaluated only on the standard version of the game. This project aims to compare the performance of the two pre-trained agents on many alterations of the Super Mario Bros environment.

Introduction

The aim of this project is to explore and compare the effectiveness of the Policy Gradient (PG) and Double Deep Q-Networks (DDQN) algorithms as reinforcement learning (RL) agents in playing the video game Mario.

To achieve this, we use the `gym_super_mario_bros` Python library, which provides a dedicated Mario game environment specifically designed to work with RL algorithms.
The agents' performance is evaluated based on their ability to move as far right as possible to complete the Mario levels with high scores.

The interest of implementing both agents is to compare robustness on environment changes for Value-Based (DDQN) and Policy-Based (Policy-Gradient) algorithms [5]. The DDQN agent has been pre-trained over 40.000 episodes and the Policy-Gradient agent over 20.000 episodes.

Having both pre-trained models avoids us long training time and saves us big computational resources.

The possible actions of the agents are : Going right ($\rightarrow$) and right + jump ($\nearrow$).

We conducted a performance evaluation of both agents by running them for 1000 episodes and analyzing their mean rewards and average number of steps per episode at intervals of 10 episodes. We then plotted the results for both agents for each change in the environment being tested.

We presented and discussed the performance of these agents in relation to their ability to play Mario and the metrics used to evaluate their performance.

The environment

The `gym-super-mario-bros` is a reinforcement learning environment within the OpenAI Gym toolkit. It is a platform for training and testing reinforcement learning agents in the context of the classic Super Mario Bros game.

The environment provides an interface for reinforcement learning algorithms to interact with the Super Mario Bros game through a global set of observations, actions, and reward functions. The observations are the states of the game that the agent observes, such as the position of Mario and the enemies, the score, and the time remaining. The actions are the possible moves that Mario can make in general, such as jumping or moving forward. The reward function provides feedback to the agent based on its actions and the current state of the game, and is used to train the agent to learn optimal strategies.

Figure 1: Super Mario Bros standard environment

Environment altercation

The `gym-super-mario-bros` has 3 variations of its environment :

1. **Downsampled ROM** which removes background and non primary elements of the game
2. **Pixelised ROM** which increases the pixelization of the image (smoothing effect)
3. **Rectangle ROM** which replaces each elements by rectangles

Figure 2: Super Mario Bros environment variations

The agents

Double DQN

The Double DQN (Double Deep Q-Network)[8] agent is a type of reinforcement learning agent that enhances the performance of the original DQN algorithm. As a value-based agent, it evaluates the value of every action in each state and chooses the action with the highest value to execute.

The Double DQN algorithm is a variation of the Q-learning algorithm [4] that uses a neural network to estimate the Q-value function, which represents the expected future reward for taking a particular action in a given state. To overcome the overestimation problem that can occur with standard Q-learning and DQN algorithms, Double DQN uses two separate neural networks - one for action selection and another for target estimation. By decoupling the action selection and target estimation, Double DQN reduces the overestimation of Q-values and improves the agent's overall performance. The target estimation network is periodically updated with the parameters of the action selection network to stabilize the learning process and enhance the accuracy of the Q-value estimates.
To update its Q-values, the DDQN algorithm uses the following Bellman equation :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \, \textit{argmax}_{a'} Q(s_{t+1}, a'; \theta); \theta') - Q(s_t, a'; \theta)]$$

Where:

- $Q(s_t, a_t)$ is the Q-value for taking action a_t in state s_t.
- r_{t+1} is the reward received after taking action a_t in state s_t and transitioning to state s_{t+1}.
- γ is the discount factor for future rewards.
- α is the learning rate or step size.
- $\textit{argmax}_{a'} Q(s_{t+1}, a'; \theta)$ is the action that maximizes the Q-value for the next state s_{t+1}, given the current action-value function parameters θ.
- θ are the parameters of the action-value function used for action selection and θ' are the parameters of the target action-value function used for target estimation.

Policy Gradient

In reinforcement learning, a policy is a function that maps states to actions. A policy-gradient [9] algorithm is a type of reinforcement learning algorithm that optimizes the policy function directly by using gradient descent to maximize the expected total reward. [2]

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta)$$

Where θ_t and θ_{t+1} are the policy parameters at time t and $t+1$, respectively, α is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the expected total reward $J(\theta)$ with respect to the policy parameters θ.

Function approximator

A deep neural network with the subsequent architecture has been utilized to parameterize both of the aforementioned algorithms.

4×32 ConvLayer
ReLU
32×64 ConvLayer
ReLU
64×64 ConvLayer
ReLU
Flatten
Linear layer
ReLU
Linear

Notice that the policy gradient agent uses an extra Softmax after the last layer.

Results of Experiments

The agents were evaluated on numerous environmental changes and the following metrics were employed to evaluate their performance across all experiments:

1. Average number of steps (per `k` episodes).
2. Average rewards (per `k` episodes).

A high average number of steps may indicate that the agent is stuck at a particular point in the level and cannot advance further until the end of the playtime, whereas a very low number of steps may indicate that the agent is unable to survive early game challenges. Plus, seeing that the reward function is based on the assumption that the goal of the game is to progress towards the right as far as possible, a higher average reward means a better progression in the game.
We ran both agents over 1000 episodes. An episode is done either when Mario dies (falls into a p//it, or is touched by an enemy) or when Mario reach the end of the level. We plotted both metrics in the Appendix section. Overall, from the results we witness that few changes of the environment can perturb both policy-based and value-based agents compared to their performance in the standard environment. Average reward mainly drops from a range of 1000-2000 to 400-900.

The **rectangle** environment seems to perturb the most the agents, this can be due to the fact that the elements are the most different from standard environment. Oppositely, the **downsampled** environment causes the least perturbation to the agents.

We observed changes in agent behavior when rendering the environment, with the DDQN agent appearing to be more prone to getting stuck on green pipes and unable to jump over them, while the Policy Gradient agent seemed to not take enemies into account.
These phenomena may be explained by the limited action space of both agents, where they can only move right or jump right. Introducing additional actions, such as stopping or moving backwards, may potentially improve agent performance despite the current reward function.

Conclusion

In this project, we conducted a performance comparison of two pre-trained agents in various altered environments of the `gym-super-mario-bros` environment. Specifically, we evaluated the Double DQN Agent and the Policy-Gradient Agent, both of which were pre-trained by other implementations for 40,000 and 20,000 episodes, respectively.
We tested these agents on 1000 episodes in each of the different environment changes and recorded the average reward and number of steps for every 10 episodes.
From our measurements, we observed that even minor changes in the environment can significantly affect the performance of an agent. Also, the effects of the changes varies depending on whether the agent is value-based (gets stuck on green pipes) or policy-based (does not take enemies into account). Although the Policy-Gradient agent was pre-trained for fewer episodes than the Double-DQN agent, it performed better overall. The number of episodes isn't always indicative for performance. Other factors such as the environment's complexity and the agent's architecture can have an impact .

Openings

Some of the possible continuities of this project can be the followings :

- **Try to compare with an Actor-Critic Based Agent:** Indeed, Actor-Critic algorithm takes the best of both Policy-based and Value-Based algorithms. In addition, one of the tutorial provided a Proximal Policy Optimization [7] (PPO), which is an Actor-Critic algorithm that combines value-based and policy-based methods to improve training stability by avoiding too large policy updates.
- **Bring other modification to the environment:** One idea for future research is to test the agent's robustness by introducing various changes to the environment, such as cutting slices of the frame or altering the color pattern. This type of experimentation could be valuable in complex tasks where generalizing models to different visual environments may be challenging. Conducting comparative studies beforehand could provide important insights for such endeavors.

References

[1] Yuansong Feng, Suraj Subramanian, Howard Wang, and Steven Guo. Train a mario-playing rl agent.

[2] Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. A closer look at deep policy gradients, 2020.

[3] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning, 2017.

[6] Nikita Schneider. A simple guide to reinforcement learning with the super mario bros.. environment, Jul 2021.

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

[8] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[9] Lilian Weng. Policy gradient algorithms, Apr 2018.

Appendix

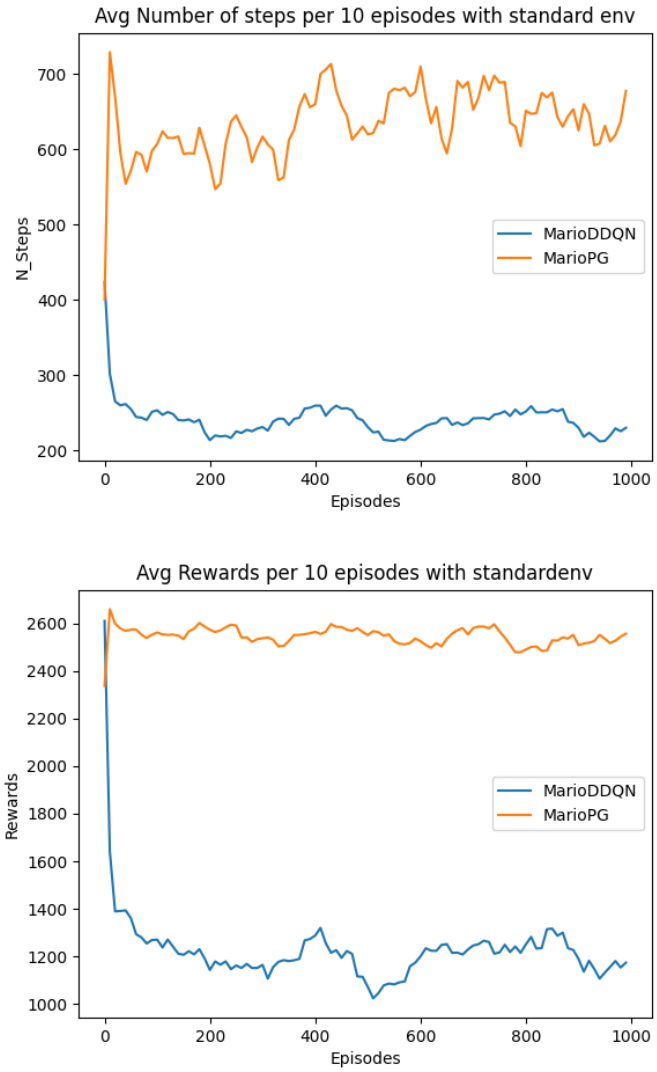

Figure 3: Number of steps and rewards per episodes for the standard version of the environment

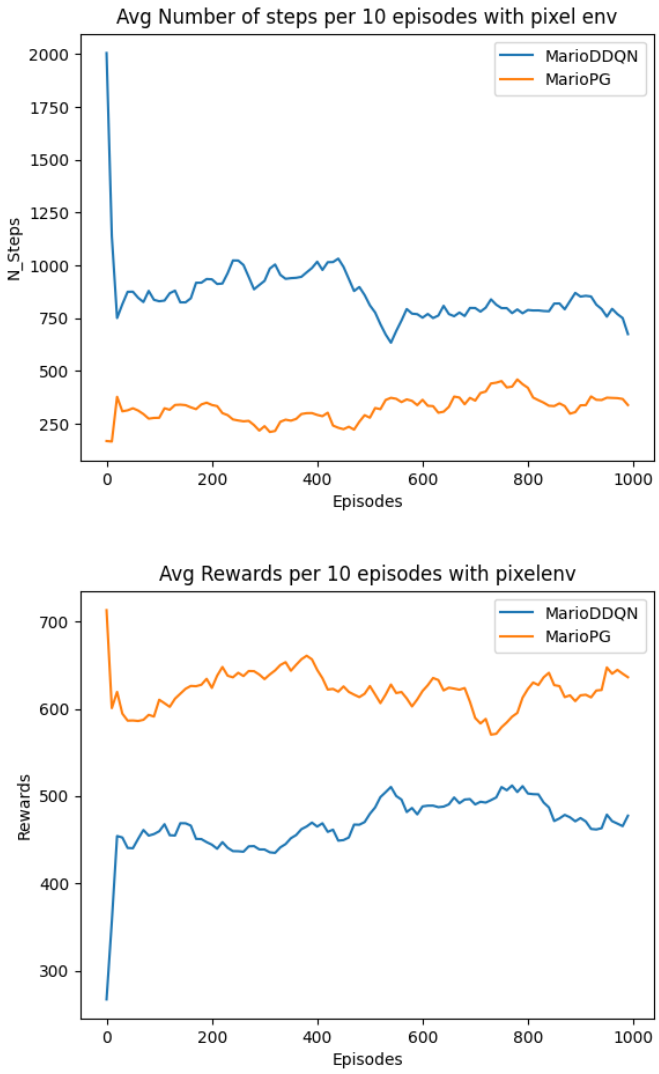

Figure 4: Number of steps and rewards per episodes for the pixel version of the environment

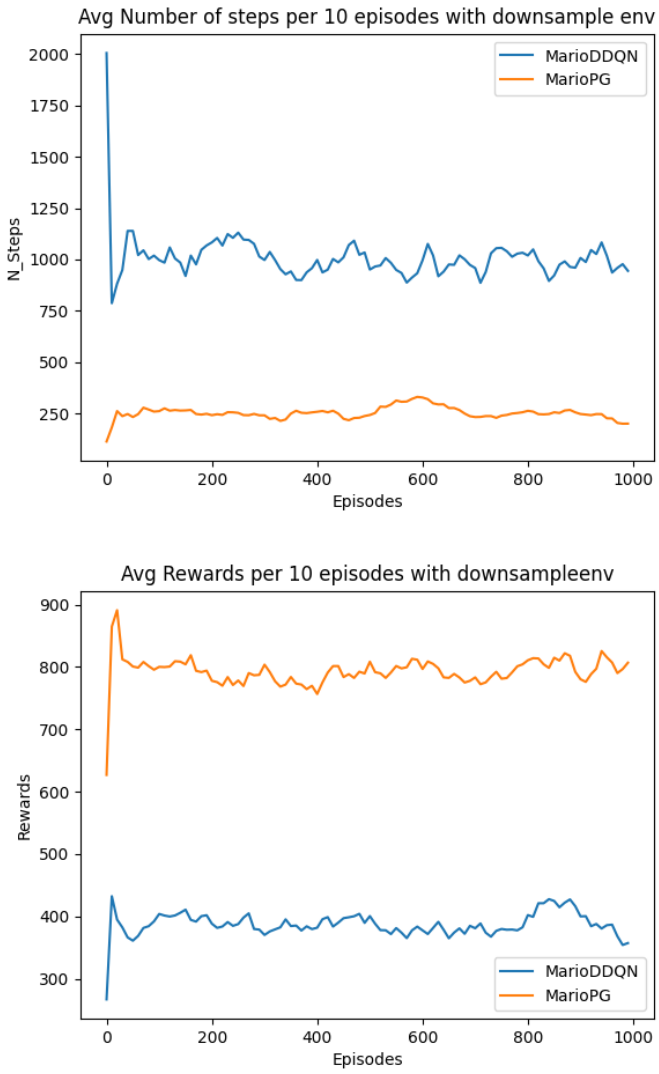

Figure 5: Number of steps and rewards per episodes for the downsampled version of the environment

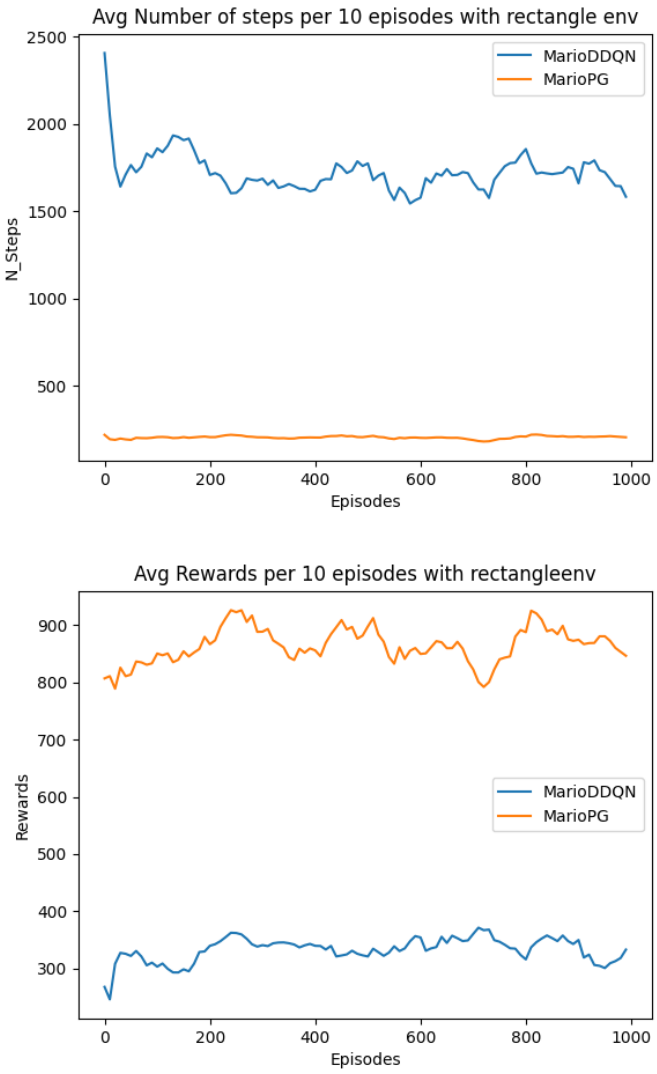

Figure 6: Number of steps and rewards per episodes for the rectangle version of the environment