SupeRL mAIrio

Venturini Daniele (1991255),
Rufo Simone (1885652),
Pamfile Diana (1943337),
Zanoni Leila (2033176)

June 4, 2024

1 Introduction

The field of Artificial Intelligence has seen very fast advancements, especially in the branch of Reinforcement Learning (RL). Our approach to this field was to work on a project that implements a RL agent to play Super Mario Bros, learning from the consequences of his actions, and how these actions maximize, or not, the rewards returned. Our goal was to find out the best effective strategies for training this agent, also by overcoming the limitations encountered during the training of the model.

For this project, we used a simplified version of a Convolutional Neural Network (CNN), a type of Deep Neural Network (DNN), used for learning and decision-making in a game environment by mimicking the information processing of the brain. Our goal was to complete the first level of the Super Mario Bros game, with the Double Q-learning algorithm, which combines Deep Neural Netowrks and Q-learning, enabling agents to learn optimal policies in complex environments.

2 Methodology

2.1 Reinforcement Learning

The Reinforcement learning technique helps an AI agent learn a specific behavior through repeated "trial-and-error" interactions with a dynamic and complex environment. This approach allows the agent to make a series of choices in order to maximize a reward metric for the activity, without being explicitly programmed for such operation and without human intervention. Based on the collected observations, actions, and rewards from the environment, a training algorithm proceeds to meticulously adjust the agent's policy. This process involves evaluating the outcomes of the agent's actions, analyzing the received rewards, and modifying the policy to enhance future performance. Through continuous training and fine-tuning, the agent becomes more adept at navigating the environment, making more informed and effective decisions over time.

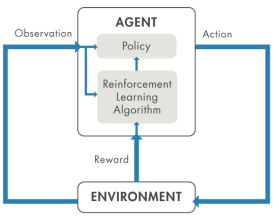

Figure 1: Reinforcement learning model

2.1.1 RL workflow

- **Problem Formulation:** Define the activity that the agent must learn, including how the agent interacts with the environment and any primary and secondary objectives the agent needs to achieve.
- **Environment Creation:** Interpret the environment in which the agent operates, including the interface between the agent and

the environment and the dynamic model of the environment.

- **Reward Definition:** Specify the reward signal the agent uses to measure its performance against the activity's objectives and how this signal is calculated by the environment.
- **Agent Creation:** Create the agent, as well as defining a representation of the policy and configuring the agent's training algorithm.
- **Agent Training:** Train the agent's policy representation using the defined environment, reward, and training algorithm.
- **Agent Validation:** Evaluate the performance of the trained agent by simultaneously simulating the agent and the environment.
- **Policy Deployment:** Deploy the trained policy representation by using, for example, generated GPU code.

Figure 2: Reinforcement learning workflow

Training an agent through reinforcement learning is an iterative process. Decisions and outcomes from subsequent phases may require returning to a previous phase of the training workflow. For instance, if the training process does not converge to an acceptable policy within a reasonable time frame, it may be necessary to update one of the following elements before retraining the agent:

- Training settings
- Training algorithm configuration
- Policy representation
- Reward signal definition
- Observation and action signals
- Environment dynamics.

2.2 Double Deep Q Network

In our project, we employed the Double Deep Q-Network (DDQN) algorithm to train our Super Mario agent. The Double Deep Q-Network is an advanced variation of the Deep Q-Network (DQN) algorithm that establishes an innovative approach to address the overestimation bias of Q-values. Building upon the core principles of Q-learning, DDQN enhances the accuracy and stability of the learning process by utilizing separate neural networks for action selection and evaluation, resulting in a more robust and reliable training process.

2.2.1 What is Q-learning?

Q-learning is a type of reinforcement learning that allows a model to iteratively learn and improve over time by selecting the correct actions. Q-learning is particularly intriguing because it does not require prior knowledge of the environment. The agent can start without knowing anything about the world it operates in and, through exploration and experimentation, is able to develop an optimal strategy to achieve its goals.

2.2.2 How does Q-learning work?

The Q-learning models operate in an repetitive process involving multiple components working together to assist in training a model. The iterative process involves the agent learning by exploring the environment and updating the model as the exploration continues. The multiple components of Q-learning include the following:

- **Agents:** The agent is the entity that acts and operates within an environment.
- **States:** The state is a variable that identifies the current position in an environment of an agent.
- **Actions:** The action is the agent's operation when it is in a specific state.
- **Rewards:** A foundational concept within reinforcement learning is the concept of providing either a positive or a negative response for the agent's actions.

- **Episodes:** An episode is when an agent can no longer take a new action and ends up terminating.

- **Q-values:** The Q-value is the metric used to measure an action at a particular state.

2.2.3 Deep Q-Network

In 2013, the DeepMind team introduced the Deep Q-Network, or DQN, which is an innovative extension of Q-learning. The DQN combines traditional Q-learning with deep neural networks; This combination has opened new doors in reinforcement learning, finally enabling the resolution of problems that were previously considered too difficult or complex. DQN is a simpler implementation that utilizes a single Q-network for both target and current Q-values.

The DQN has two principal limitations:

- **Overestimation Bias of Q-values:** The DQN uses a single neural network both to select and to evaluate actions, which can lead to a systematic overestimation of Q-values. This bias can compromise the algorithm's performance, resulting in sub-optimal policies.

- **Training Instability:** Training of the DQN can be unstable, with oscillations in Q-values that can delay or compromise the algorithm's convergence.

To address these issues, the DDQN utilizes two separate neural networks.

2.2.4 Double Deep Q-Network

DDQN, or Double Deep Q-Network, is a reinforcement learning technique that extends the concept of Deep Q-Network (DQN). In DDQN, instead of using a single neural network to estimate both the best action and its value, we use two separate neural networks.

Our DDQN implementation employs two Convolutional Neural Networks (CNNs), referred to as Q_{online} and Q_{target}. The Q_{online} network is responsible for selecting the optimal action for a given state and is updated frequently. Meanwhile the Q_{target} network evaluates the value of the chosen action and is updated less frequently to reduce overestimation bias.

Both CNNs share the same architecture for feature extraction but maintain different weights in their fully connected layers. To ensure stability and prevent divergence in value estimates, the weights of Q_{target} are periodically synchronized with those of Q_{online}.

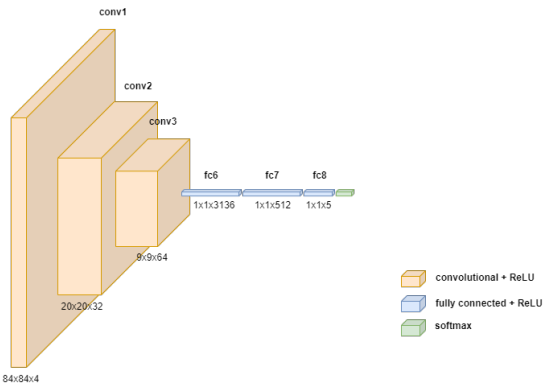

Figure 3: Graphical representation of the CNNs

In the learning process, there are involved two values:

$Q_{\textbf{predicted}}$ - the predicted optimal Q^* for a given state s

$$Q_p = Q^*_{\text{online}}(s, a)$$

$Q_{\textbf{target}}$ - aggregation of current reward and the predicted Q^* in the next state s'

$$a' = \arg\max_a Q_{\text{online}}(s', a)$$

$$Q_t = r + \gamma Q^*_{\text{target}}(s', a')$$

Since the next action a' is not known, we use the action a' that maximizes Q_{online} in the next state s'.

To update the weights, we take some samples of states, actions and next states from a replay buffer, calculate Q_{p} and Q_{t} and then compute the loss with the Mean Squared Error (MSE):

$$\text{Loss} = \alpha \left(Q_t - Q_p\right)^2$$

where α is the learning rate of the optimizer (Adam), set to 0.0005

2.3 Gym library

The library we used to emulate the game is gym-super-mario-bros, an OpenAI Gym environment specifically designed for "Super Mario Bros." and "Super Mario Bros. 2" on the Nintendo Entertainment System (NES). This library utilizes the nes-py emulator, which is responsible for accurately emulating the NES environment.
This library simplifies the interaction with the game by providing environment wrappers, customizable action spaces, and a structured reward system.
The environment wrappers pre-process the game state, transforming the frames into a format suitable for analysis by the Reinforcement Learning algorithm. Originally, each game state is represented by a 3x240x256 size array. To reduce data dimensionality, we used built-in environment wrappers that convert the RGB images to grayscale and downsample each state to a square image, resulting in a new size of 1x84x84. We then used a custom wrapper that repeats the same action for four frames, exploiting the fact that consecutive frames exhibit minimal variation. The rewards accumulated during these four frames are aggregated into the fourth frame. Finally, we stack four consecutive states together to provide the RL model with temporal context, allowing it to identify the direction in which Mario is moving.
The customizable action space enables a range of movements, from basic actions like moving left and right to complex actions such as running and jumping simultaneously. For our project, we restricted the available actions to those that move Mario to the right. This approach is sufficient to complete the first level and helps reduce training time by limiting the actions that the RL model can choose from.
The reward system is designed to train Mario with the objective of moving as far to the right as possible, doing so quickly, and avoiding death, thereby completing the level. After each action taken, a reward value r is returned to indicate the effectiveness of the action. This value helps the reinforcement learning algorithm determine whether the action was beneficial or detrimental.

The reward r is calculated as $r = v + c + d$. The variable v represents the difference in the x-coordinate between states, giving a positive value only if Mario is moving to the right. The variable c represents the difference in the game clock between states, penalizing Mario for standing still. Lastly, the variable d is a death penalty that penalizes Mario for dying in a state. The reward value r is then clipped to the range $[-15, 15]$ to prevent disproportionately rewarding actions that might be beneficial only in the immediate future.

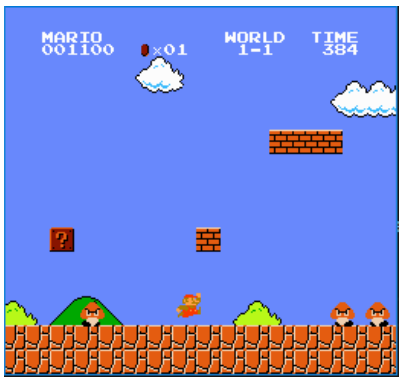

Figure 4: The resized environment where Mario plays

2.4 Implementation

2.4.1 Libraries

We used these libraries for defining and training the model:

* **torch-rl:** is a reinforcement learning library built on PyTorch. We mainly used it to train our agent and to define our RL models.
* **gym-super-mario-bros:** is a package that integrates the classic Nintendo game Super Mario Bros with OpenAI Gym. This package wraps the game environment so that it conforms to the standard OpenAI Gym interface, making it easy to use with existing RL algorithms and libraries.
* **tensordict:** is a utility package in PyTorch designed to handle structured data ef-

ficiently, especially for use cases in reinforcement learning. We mainly used it to save the states with the actions that Mario took.

2.4.2 Algorithms

```
for e in range(EPISODES):
```

This line starts a loop that runs for a given number of episodes. Each iteration of the loop represents a complete episode of the game.

```
state = env.reset()
```

At the beginning of each episode, the environment is reset to its initial state, and the initial state is stored in the variable state.

```
    total_reward = 0
    max_x = 0
```

Variables are initialized to track:

- **total reward:** Accumulates the total reward for the current episode.
- **max x:** Tracks the maximum x-position reached by Mario.

```
    while True:
```

This is a inner loop for each step in the episode. Each iteration of this loop represents an episode in the game.

```
    action = mario.act(state)
```

The agent (Mario) chooses an action based on the current state using its policy. The act method chooses an action based on the epsilon-greedy policy, processes the input state, predicts action values using a neural network, updates the exploration rate, increments the step counter, and returns the chosen action.

```
    next_state, reward, terminated,
    truncated, info = env.step(action)
    done = terminated or truncated
```

The chosen action is performed in the environment. This interaction returns:

- **next state:** The new state after the action.
- **reward:** The reward obtained from performing the action.
- **done:** Combines the value of terminated, a boolean indicating if the episode has naturally ended, and truncated, a boolean indicating if the episode was forcefully ended (e.g., due to a time limit).
- **info:** Additional information about the environment.

```
    mario.cache(state, next_state, action,
    reward, done)
```

The agent saves the current experience (state, action, reward, next state, and done flag) into the agent's replay buffer. This stored experience is later used to train the agent, helping to improve learning efficiency, break correlations in the training data, and stabilize the training process.

```
    mario.learn()
```

The agent learns from its stored experiences, updating its knowledge (e.g., Q-values or policy) and eventually sync the policy between $Q_{\text{predicted}}$ and Q_{target}.

```
    state = next_state
    total_reward += reward
    max_x = max(info["x_pos"], max_x)
```

The state is updated to the next state. The total reward for the episode is incremented by the current reward, and the maximum x-position reached is updated if the current x-position is greater.

```
    if done or info["flag_get"]:
        break
```

The loop breaks if the episode is done or if a specific condition is met (in this case, if the info dictionary contains the key "flag get", which is set when Mario reaches the goal).

3 Results

In this section, we present the findings from our project where we developed and trained an AI agent to play Super Mario using RL. Mario underwent a training process spanning over 80,000 episodes and lasting multiple days. This section details the performance metrics of Mario, including its ability to complete levels, the average rewards obtained, and the observed learning curve over multiple episodes. Through visualizations and quantitative data, we demonstrate the progression of Mario's learning in the game scenario.

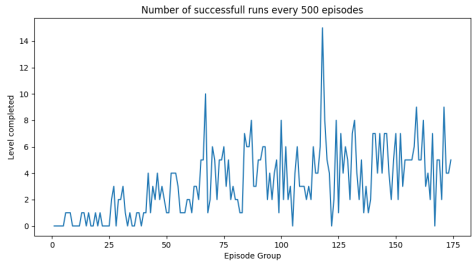

Figure 5: Number of times Mario completed the first level every 500 episodes. Initially, Mario rarely finishes the level, but over time, the completion rate increases.

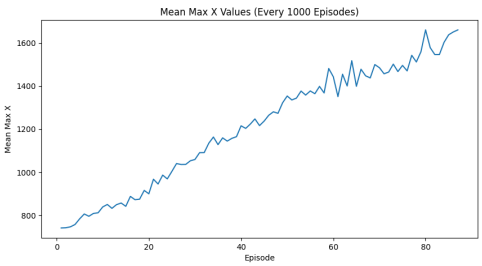

Figure 6: The average maximum distance reached by Mario every 1000 episodes. Over time, we can observe a steady increase in the maximum distance.

The results of our project indicate that the reinforcement learning approach was effective in teaching Mario to play Super Mario with a notable degree of skill. Over the course of training, Mario demonstrated significant improvement in its ability to complete levels, optimize rewards, and adapt to new challenges presented by the game environment. The combination of Q-learning, deep neural networks, and experience replay proved to be a robust framework for this task.

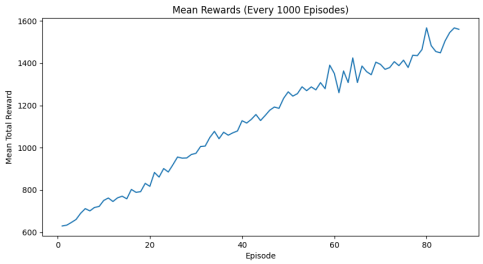

Figure 7: The average maximum distance reached by Mario every 1000 episodes. We can observe that the graph closely resembles the mean maximum distance graph, as the rewards are related to the distance traveled

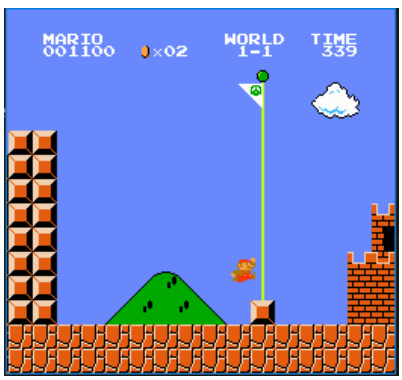

Figure 8: Mario reaches the flag of the first level

4 References

- ⋆ "Deep Reinforcement Learning Doesn't Work Yet" by Alex Irpan
  https://www.alexirpan.com/2018/02/14/rl-hard.html
- ⋆ "Deep Reinforcement Learning with Double Q-learning"
  by Hado van Hasselt, Arthur Guez, and David Silver
  https://arxiv.org/abs/1509.06461
- ⋆ "Reinforcement Learning: An Introduction"
  by Richard S. Sutton and Andrew G. Barto
  https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
- ⋆ "AI Learns to Speedrun Mario"
  Video by SethBling
  https://www.youtube.com/watch?v=OQitI066aI0