Pong Project

Théau d'Audiffret, Roman Maria & Albane Vigier

January 2025 `https://github.com/Albeenee/RL-Pong.git`

1 Context

Pong is a classic Atari 2600 game and widely used in reinforcement learning. Its discrete action space and visual input make it a good illustration of deep RL algorithms. Since the seminal work of DeepMind introducing Deep Q-Networks (DQN) [3], Pong has been used to demonstrate the ability of agents to learn complex control policies directly from high-dimensional pixel inputs, hence why we chose this project. State-of-the-art methods, such as Double DQN [1], Dueling DQN [4], and Prioritized Experience Replay, have since been developed to improve learning stability and sample efficiency on such environments.

2 Environment

We used the Pong game environment from Gym. In this game, the agent is a paddle (on the right side) competing with another paddle (on the left side), exchanging a ball. Only two actions are possible: moving **UP** or **DOWN**, each paddle stays on a vertical axis (see Figure 1).

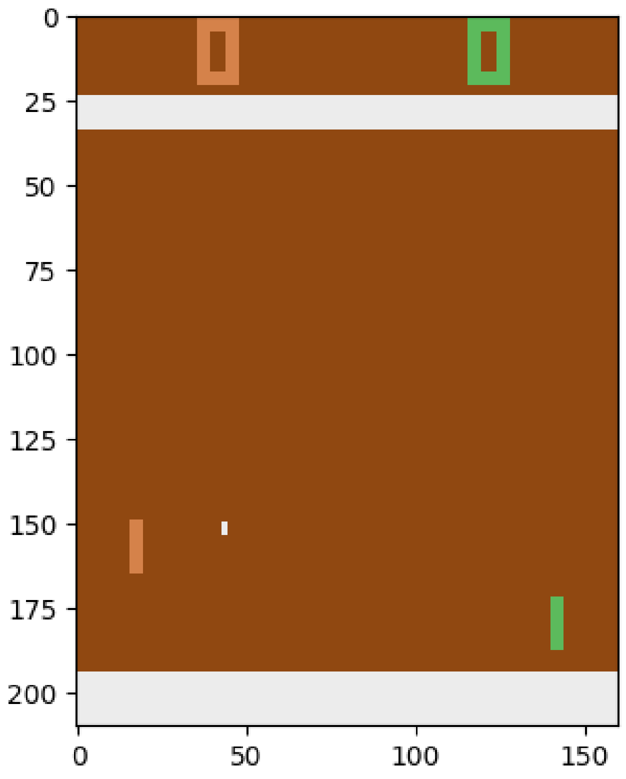

Figure 1: Gym Pong game environment

A player or team scores one point when the opponent hits the ball out of bounds or misses a hit. The first player or team to score 21 points wins the game. The original state returned by this environment is the 250x160x3 RGB image.The reward is updated when the ball crosses a paddle:

- +1 if opponent (right paddle),
- -1 otherwise (left paddle)

3 Methodology

Inspired by the Flappy-Bird Gym environment, our first intuition was to try to extract some meaningful features manually, such as the distance between the ball and the agent, the velocity and position of the ball, the velocity and position of the agent... However, we soon realized that this manual extraction was too complex, in particular because for synchronising the features extraction with the evolution of the game through time. The complexity of this game is also reinforced by the fact that there is more unpredictability due to the presence of another agent, which ultimately guided us towards Deep Learning solutions.

This led us to implement a **Deep Q-Network** (DQN) agent [2]. The workflow starts by **preprocessing** the environment's observations, where images are resized (84×84) and converted to grayscale. The agent stacks 12 consecutive frames to capture temporal dynamics, forming the state representation.

The core of the learning relies on a **Replay Buffer**, which stores past transitions (state, action, reward, next_state, done) to enable experience replay and stabilize training by breaking temporal correlations.

The agent follows an **epsilon-greedy** strategy to balance exploration and exploitation. Initially, actions are chosen randomly with high probability, but as training progresses, **epsilon decay** encourages more exploitation of learned policies.

The model is updated by sampling random batches from the replay buffer and minimizing the **Mean Squared Error** (MSE) between the predicted Q-values and the target Q-values calculated using a target network. This **target network** is periodically synchronized with the main model to improve stability during learning.

In order to track our agent evolution, we included in our implementation periodic saving of **model checkpoints** (every 10 000 episodes). This way we could test our model at different steps of the training process, and we countered the risk of loosing everything when the code crashed (which happened a few times since the model is wide and heavy to train).

4 DQN Model

4.1 General description

The DQN agent takes a stack of consecutive game frames as input and processes them through a convolutional neural network to extract meaningful spatial and temporal features, such as the position and motion of the ball and paddles. These features are then mapped through fully connected layers to estimate the action-value function, providing a Q-value for each possible action. The architecture is designed to efficiently handle visual input and approximate the optimal policy through reinforcement learning.

The Deep Q-Network (DQN) we used in this project is a Convolutional Neural Network (CNN) designed to process a stack of 12 grayscale frames (84x84 pixels each) as inputs.

4.2 Architecture

Convolutional layers

- **Conv1**
  - Input: (12, 84, 84)

  - 32 filters of size 8x8, applied with a stride of 4, reducing the spatial dimensions significantly.
  - Output feature map size: approximately (32, 20, 20) after downsampling

- **Conv2**
  - 64 filters of size 4x4, with a stride of 2
  - Further extracts mid-level spatial features such as ball and paddle positions.
  - Output feature map size: approximately (64, 9, 9)

- **Conv3**
  - 64 filters of size 3x3, applied with a stride of 1
  - Refines and deepens the extracted features to capture complex patterns relevant to Pong dynamics.
  - Output feature map size: approximately (64, 7, 7)

Each convolutional layer is followed by a **ReLU activation**, introducing non-linearity to the model. After the final convolutional layer, the output tensor is flattened into a vector of size $64 \times 7 \times 7 = 3136$.

Fully connected layers

- **FC1**
  - 512 units
  - Acts as a bottleneck, compressing the spatial features into a dense representation.
  - Followed by a ReLU activation.

- **FC2**
  - 1024 units
  - Expanding the feature representation for improved capacity to capture complex value functions.
  - ReLU activation applied.

- **Output layer (FC3)**
  - 6 units, corresponding to the 6 discrete actions available in the Pong environment
  - Outputs the Q-values for each action, without activation (raw scores used for action selection and learning).

This architecture is inspired by the original **DeepMind DQN** used on Atari games [3], but includes an extra fully connected layer (1024 units) compared to the standard design to provide additional representational power. The convolutional layers are responsible for detecting and encoding spatial features from the visual input, while the fully connected layers map these features to action-value estimates. This architecture is illustrated below in Figure 2.

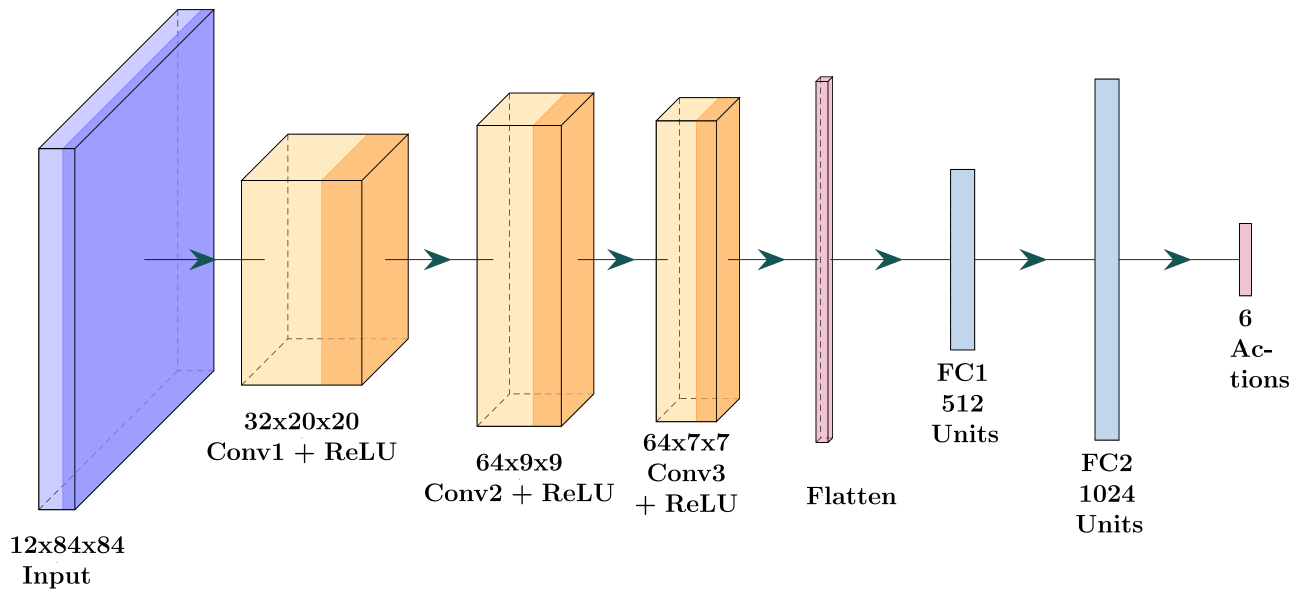

Figure 2: Deep Q-Network (DQN) architecture for Atari Pong.

5 Results

The training is plotted on figure 3. The scores are shifted (+21) so the best theoretical score that the agent could obtain in this configuration is 42, the worst being 0 (the agent's starting score).

The training took a lot of computational time: 1,750,000 episodes, 36 hours to obtain better performance. Also, the model demonstrated a very high sensitivity to hyper-parameters. We ended up choosing a Window length set to 12 instead of 6 (otherwise, it did not seem to learn), increased the batch size to 128 for stability and applied a learning rate decay to better capture local minima.

The model was very slow to converge: we needed at least 100,000 episodes to see if it was learning or not (to get the model out of its static state). This meant that a thorough search of the best parameters was impossible.We finally achieved the target score of 21: our engine now matches the Gym agent during a game. We expect that with more training it would eventually beat it.

By observing the agent playing (render_mode = 'human'), we discovered an unexpected behavior: instead of focusing on perfect defense, the agent learned to exploit a "glitch" by kicking the ball very strongly as soon as it was put into play, which we could describe as an offensive strategy.

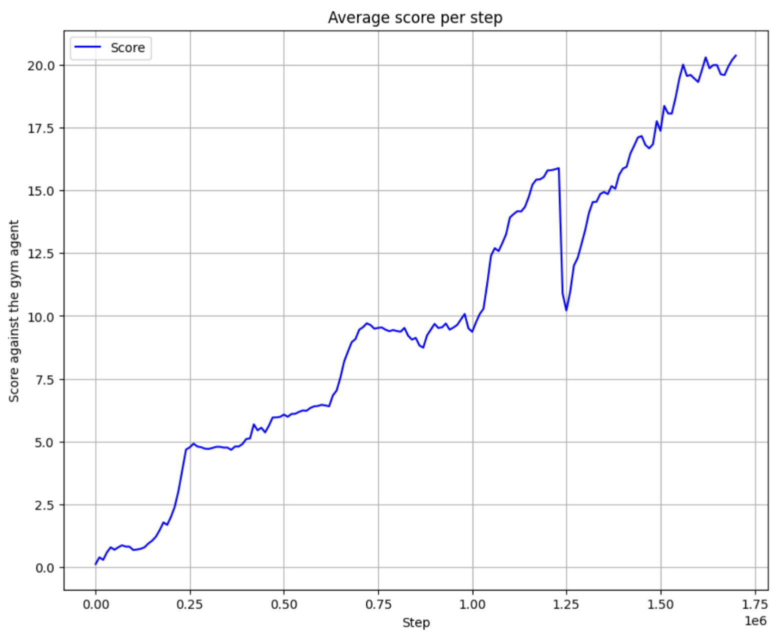

Figure 3: Average scores during training

6 Conclusion & Discussion

In this project, we successfully implemented a Deep Q-Network (DQN) agent to learn and play the Atari Pong environment using reinforcement learning techniques. However, despite achieving some learning progress, we faced a few limitations. Training such deep reinforcement learning models is computationally demanding and very time-consuming. The Pong environment, while simple in appearance, requires the agent to extract information from a $160 \times 210 \times 3$ ($\times 12$ if we count the time dimension) matrix in order to capture and predict complex ball/paddle dynamics across time. This makes it particularly sensitive to parameter tuning (learning rate decay, window length, batch size...).

Due to the absence of GPU resources, our training process was significantly slowed down. It typically took at least 24 hours to observe meaningful performance improvements and around 5 hours to even validate if a specific set of parameters was promising. This limited the number of experiments we could conduct and prevented us from performing a thorough grid search or testing more advanced DQN variants (such as Double DQN, Dueling DQN, or Prioritized Replay).

Overall, this project highlights both the potential and the challenges of applying deep reinforcement learning to vision-based control tasks under resource-constrained conditions.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv preprint* (2016). arXiv: `1509.06461 [cs.LG]`.

[2] Antonio Lisi. "Beating Pong using Reinforcement Learning — Part 1 DDDQN". In: *Analytics Vidhya* (2021).

[3] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint* (2013). arXiv: `1312.5602 [cs.LG]`.

[4] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *arXiv preprint* (2016). arXiv: `1511.06581 [cs.LG]`.