Reinforcement Learning - Pacman

Ciuperceanu Vlad-Mihai, Huzum Andreea, Miu Tudor-Gabriel
Group 351

1 Introduction

Pacman is a video game released in 1980 on Atari consoles, which quickly became a symbol of the video game industry and an important part of its history.

This project aims to explore the application of Reinforcement Learning in the Pacman game, using different algorithms to improve the agent's performance. We will analyze their efficiency and compare the results obtained.

2 Environment

We will use the Pacman environment from Atari Gymnasium (Arcade Learning Environment).

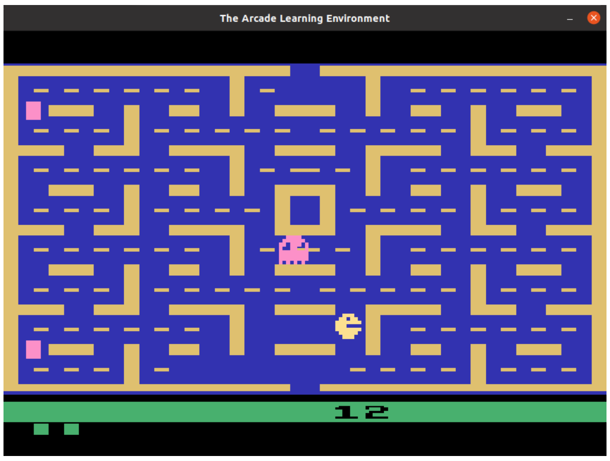


Figura 1: ALE/Pacman-v5

2.1 Interaction with the Environment

Like any Atari game in the framework, states are given as RGB frames of 250x160 pixels from the game.

The player has 5 actions to choose from: NOOP (does not change the position) or UP, DOWN, LEFT, RIGHT (for the 4 directions).

The reward corresponds to the score, being 1 for each pellet eaten. For the 4 power pellets, the reward is 50, and each ghost brings 100 points. In most frames, due to the game's mechanics, the reward will be 0, as no addition to the score is recorded.

In the returned information, the number of remaining lives can also be found, visible in the corner of each frame.

2.2 Preprocessing

To improve performance, especially in methods that use neural networks, preprocessing the environment and the observation space is helpful.

We chose to reduce the frame size to 84x84 pixels, converting the image to grayscale. Also, we chose not to use each frame individually for efficiency, grouping them in sets of 4 or applying a frameskip of 4 to the environment. For the first option, we implemented the transformations manually, while for the second, we used the AtariPreprocessing wrapper from Gymnasium:

```
env = wrappers.AtariPreprocessing(env, frame_skip=4,
                grayscale_obs=True, screen_size=84)
```

We preferred to preprocess the data this way as it is the standard method with the best results for Atari environments, also used in DeepMind's paper.

3 Random Agent

First, we implemented an agent that takes actions randomly, to get an idea of how the environment works and to have a reference point for implementing the algorithms.

This agent achieved a score of 13.2, obtained as the average reward after running 10 episodes.

4 Monte Carlo Tree Search

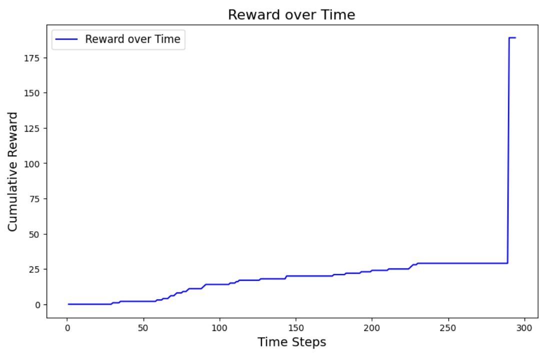
An algorithm that uses Reinforcement Learning principles is Monte Carlo Tree Search, which we chose to implement as another reference for training agents.

MCTS is suitable in the context of large environments, defined by states, actions, and rewards. It is based on exploring a decision tree of states, using "statistics" obtained from the game experience. It does not require specific knowledge of the environment and works with minimal information about the states. At the same time, it maintains a balance between exploration and exploitation through the node selection formula - UCT (Upper Confidence Bound for Trees).

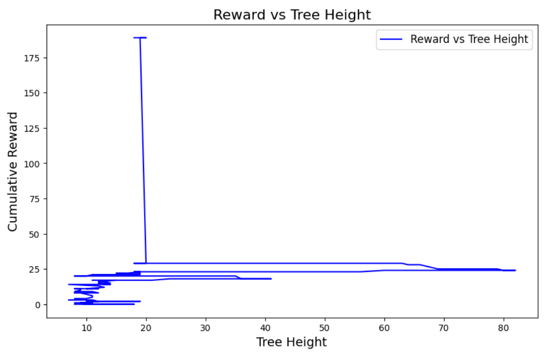
Of course, it also has disadvantages, including the high execution time (due to the creation of the state tree), the large number of episodes needed to discover the appropriate actions, high variance (due to randomly chosen actions), and the memory required for execution (also due to the created tree).

However, after running for approximately 3 hours, we obtained a very good result, reaching a reward of 189 after 294 actions decided.

The increase was incremental until it reached the state where it eats a Power Pellet, followed by an encounter with a ghost, which resulted in a massive increase in reward.



(a) Reward over time



(b) Reward for tree height

Figura 2: Reward after finding actions

It can be observed that, over time, a large depth of the tree is needed (reaching up to 80) to find the decisions with maximum potential, after which it returns to a height of 20, following the discovery of the optimal action.

This is a method that yielded results, but it is limited by the disadvantages of execution time and required memory, as mentioned above.

5 Q-Learning

A first attempt was to implement Q-Learning without using neural networks for function approximation.

Even though we preprocessed the environment using AtariPreprocessing to reduce dimensions, aiding learning and state retention, we still encountered a memory issue, with the Q-table size growing rapidly.

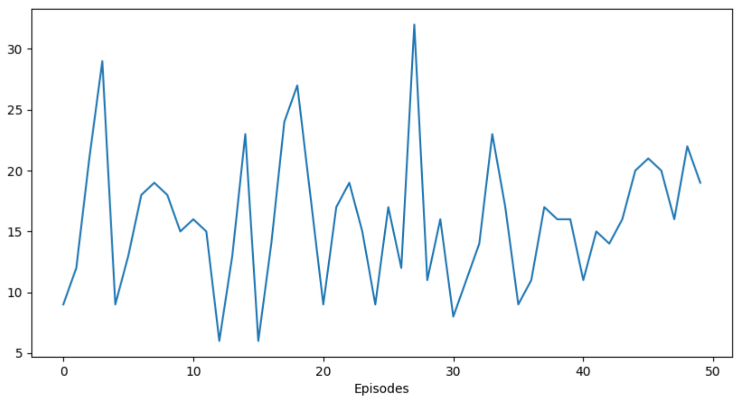


Figura 3: Reward over time

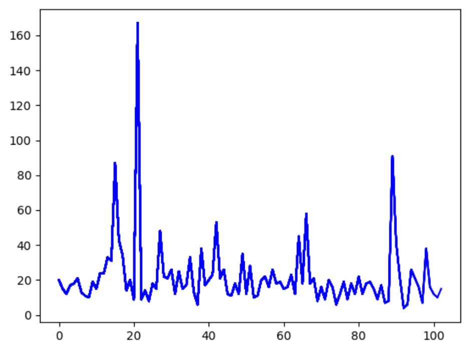
As can be seen, the number of episodes needs to be large to achieve convergence, but the execution time was significant even for 50 episodes.

6 DQN - Deep Q-Learning

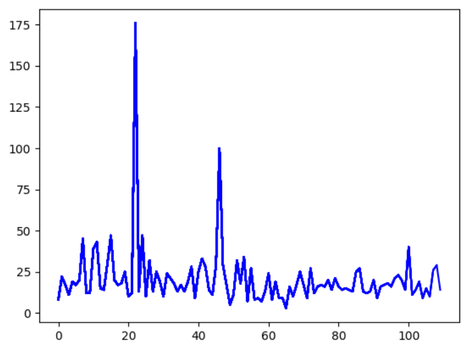
A first improvement was the use of a DQN for Q-function approximation.

We applied preprocessing to the environment to aid the neural network's learning process. Additionally, we normalized the pixel values in the image for better convergence. It is worth noting that, during the game, losing a life is not penalized, so we manually introduced a reward of -10 for such situations.

By modifying the learning rate, we obtained the following rewards:



(a) Learning Rate of 0.001

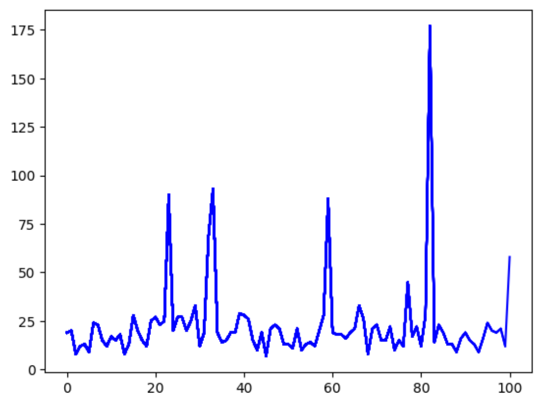


(b) Learning Rate of 0.01

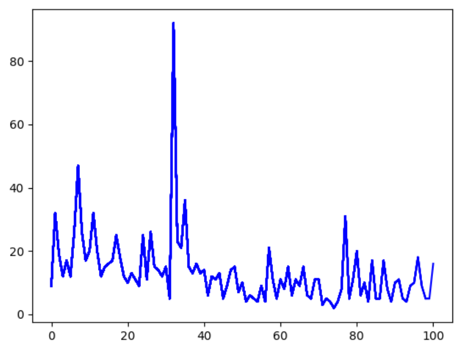
Figura 4: Reward over time, choosing a batch of 32 in experience replay

It can be observed that, due to random actions, both models reach high rewards at the beginning (likely finding the sequence of actions as in the MCTS method), but then the reward starts to decrease. However, for a smaller reward, a similar spike occurs later, around episode 90. This means the model starts to learn, but it needs more episodes.

Then, we thought we could aid the learning process by adding a reward of 0.05 for staying alive (regardless of gaining a pellet), since the reward, in the absence of a pellet, is 0. Combined with the number of frames to process, this can hinder the Q-function approximation, affecting the calculated gradients. Additionally, we modified the batch size chosen from the replay memory. The results obtained were as follows:



(a) Learning Rate of 0.005



(b) Learning Rate of 0.01

Figura 5: Reward over time, choosing a batch of 64 in experience replay

It can be observed that, depending on the learning rate, the obtained reward does not necessarily increase, possibly due to a discrepancy between the values of the rewards received in the game (possibly needing a larger penalty for losing a life or a smaller bonus for keeping it for an action). Additionally, a larger number of episodes is needed to observe how the agent learns, but the algorithm requires approximately 3 hours of runtime for the 100 episodes. However, for a learning rate of 0.005, an increase in the high scores obtained can be observed, followed by one in episode 100. Thus, it seems the model learns to obtain higher and higher rewards, but more episodes are needed to approach convergence.

7 DDQN - Double Deep Q-Learning

The next improvement in performance and convergence was the implementation of a DDQN.

This time, we modified the bonus and penalty related to keeping/losing a life to 0.01 and -20, respectively.

If in the previous algorithms we used AtariPreprocessing with a frameskip of 4, in this case, we chose to group 4 frames to constitute a state. The learning rate was 0.0005, favoring a more secure increase in learning.

After running 250 episodes for approximately 3 hours and 30 minutes, we obtained the following result:

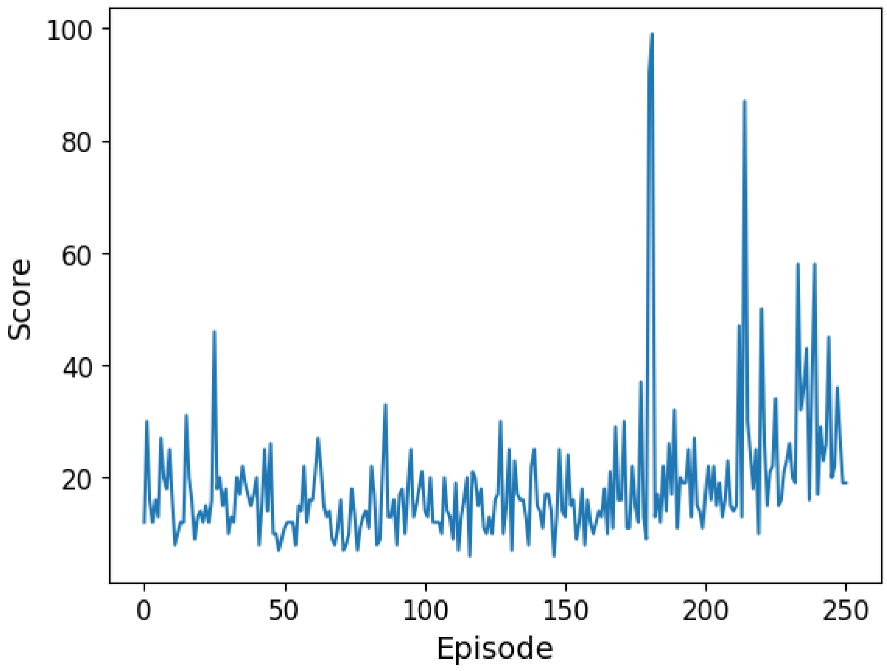


Figura 6: Reward over time

It can be observed that, starting with the first spike, an upward trend begins, indicating that, for a larger number of episodes, the agent will start to achieve higher and higher scores.

8 Stable Baselines3

As observed earlier, training DQN and DDQN models takes an extremely long time, even when using GPU for the neural network portions, so we are limited to a relatively small number of episodes. In most papers related to this topic, several million episodes are needed to achieve convergence and obtain significant scores.

Therefore, we chose to use a specialized library for Reinforcement Learning algorithms - Stable Baselines3.

This library provides model architectures, such as the one we used - DQN with CNN-Policy, but also vectorized environments. With these, we can train the agent on multiple environments in parallel, increasing both the performance obtained and the required runtime.

Thus, after running for 3.3 million steps, with a replay memory of 100,000, on 16 environments in parallel for 3 hours, we obtained an average reward of 183.6 after 10 played episodes.

9 Conclusions

As best seen in the DDQN method graph, which achieved the best result (excluding Stable Baselines3), the agents start to learn, achieving good scores, but they need more episodes and more training time.

In the end, the implementation of Reinforcement Learning algorithms presented a variety of possible solutions and obtained performances. Each method came with its challenges but also significant optimization opportunities. However, regardless of these differences, all approaches contributed to a deeper understanding of how agents can learn and adapt to complex environments.

10 References

- https://ale.farama.org/environments/pacman/
- https://stable-baselines3.readthedocs.io/en/master/guide/examples.html