

# How to crack (kind of) the videogame “Breakout”

A project for the *Reinforcement Learning* course

---

Angela Carraro

DSSC - UNITS



Teach a computer agent to play the Atari 2600 game Breakout via the Reinforcement Learning technique of Double (Deep) Q-Learning.

We will use images of the screen game to make our agent learn a policy that can allow it to score a sufficient number of points in the game (how many depends on the computing power and the time at your disposal).

**Language** Python  
**Game Environment** Gym  
**ML framework** PyTorch  
**Cluster** Ulysses



## The Game

---

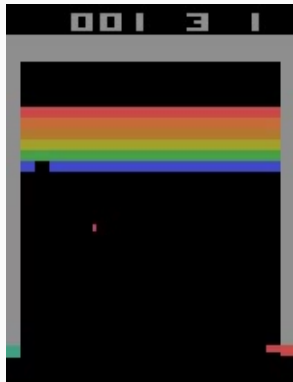


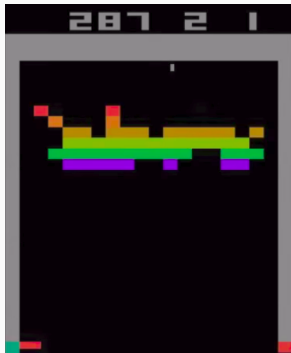
The game begins with 6 rows of different colors of 18 bricks each. After firing the ball (red button on the Atari console), the player must knock down as many bricks as possible by using the walls and/or the paddle below to bounce the ball against the bricks and eliminate them.

If the player's paddle misses the ball's rebound, they will lose a life, the ball will disappear from the screen and they would have to press the red controller button to serve another ball.

The color of a brick determines the points you score when you hit it with your ball. In the official Atari 2600 game [rules](#), there is stated that:

- Blue and green bricks earn 1 point each.
- Yellow and light orange bricks earn 4 points each.
- Dark orange and red bricks score 7 points each.





**Figure 1:** Optimal strategy to solve this game: make a tunnel around the side, and then allow the ball to hit blocks by bouncing behind the wall.

The score is displayed at the top left of the screen (maximum for clearing one screen is 448 points), the number of lives remaining is shown in the middle (starting with 5 lives), and the “1” on the top right indicates this is a 1-player game.

The paddle shrinks after the ball has broken through the red row and hit the upper wall.

The ball speed increases at specific intervals: after four hits, after twelve hits, and after making contact with the orange and red rows.

We will use the environment provided by the library [Gym](#), with documentation (scarce) available [here](#).

In the base environment of Breakout, *Breakout-v0*, each action is repeatedly performed for a duration of  $k$  frames, where  $k$  is uniformly sampled from  $\{2, 3, 4\}$ .

There are different options you can specify when setting the environment. If you look at the *atari\_env* [source code](#) (which is explained in this [link](#)), you can add one of these options after the game name *Breakout*:

- *-v0* or *-v4*: *-v0* has *repeat\_action\_probability* of 0.25 (meaning 25% of the time the previous action will be used instead of the new action), while *-v4* has 0 (always follow your issued action).
- *Deterministic*: it keeps a fixed frameskip of 4, while for the env without DETERMINISTIC the frameskip  $k$  is uniformly sampled from  $\{2, 3, 4\}$  (code [here](#)).
- *NoFrameskip*: a fixed frameskip of 1, which means we get every frame, so no frameskip.

## Using Reinforcement Learning

---



We will use the environment *BreakoutDeterministic-v4* since the .

The **state**: an RGB image of the screen, which is an array of shape (210, 160, 3).

The set of possible **actions**:

- NOOP → do nothing (“no operation”, as described [here](#))
- FIRE → throw the ball
- RIGHT → move right
- LEFT → move left

The environment is deterministic, so we always do the action we want to do.

The **reward**: an integer number with the score of the destroyed brick.

The **end of an episode** occurs when the agent finishes all the 5 lives or when it clears the screen from all the bricks.



If for each observation we stack two consecutive frames we can see the direction and the velocity of the ball. If we stack three frames we will be able to tell also the acceleration of the ball. DeepMind stacked 4 frames, probably to be sure to have all necessary information.

Then I used a sequence of four game frames stacked together, making the data dimension  $(4, 84, 84)$ . This to make the agent understand both the velocity and the acceleration of the ball.

## The Journey of Learning

---



What could be done better?





- Efficient handling of the replay memory: instead of saving 4 frames for the state and 4 frames for the next state in each experience, use the fact that 3 of them are equal to save some space.
- Implement a dueling architecture, so to generalize learning across actions (see the paper of [Wang et al.](#)).
- Use the environment *Breakout-ram-v0* ([documentation](#)), in which the observation is the content of the 128 bytes of RAM of the Atari machine. In this way the agent can learn from the individual bits.

Thank you for your attention!



# References

---

-  Marc G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: [10.1613/jair.3912](https://doi.org/10.1613/jair.3912). arXiv: [1207.4708](https://arxiv.org/abs/1207.4708).
-  Marlos C. Machado et al. “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”. In: *arXiv preprint* (2017). arXiv: [1709.06009](https://arxiv.org/abs/1709.06009).
-  Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: [10.1038/nature14236](https://doi.org/10.1038/nature14236).
-  Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv preprint* (2013). arXiv: [1312.5602](https://arxiv.org/abs/1312.5602).

To the game!

