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

A project for the *Reinforcement Learning* course

Angela Carraro

DSSC - UniTS

Aim of the project

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

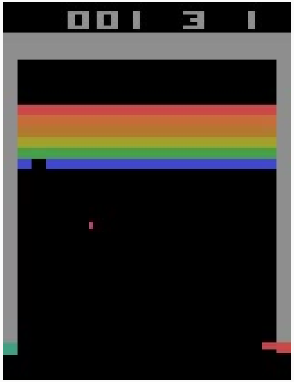
Breakout

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.



Breakout

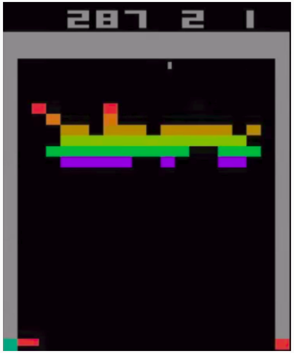


Figure 1: Optimal strategy to solve this game: make a tunnel around the side, and then allow the ball to hit blocks by bouncing behind the wall.

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.

The specifications

We will use the environment of OpenAI's library Gym [doc (scarce) available here].

In the environment a frame-skipping technique is implemented. More precisely, the agent sees and selects actions on every k-th frame, instead of every frame, and its last action is repeated on skipped frames.

There are different options you can specify when setting the environment. If you look at the ***atari_env*** source code (explained well 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

The MDP

I have used the environment ***`BreakoutDeterministic-v4`***, since there will be no stochasticity (no randomly repeated action) and no time distortion (no random frameskip).

The state: an RGB image of the screen, which is an array of shape $(210, 160, 3)$, with a 128-colour palette.

The set of possible actions:

- NOOP $\longrightarrow$ do nothing (“no operation”, as described here)
- FIRE $\longrightarrow$ throw the ball
- RIGHT $\longrightarrow$ move right
- LEFT $\longrightarrow$ move left

The reward: an integer number with the value 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.

Preprocessing the state

Working directly with raw Atari 2600 frames can be demanding in terms of computation and memory requirements. I've applied a basic preprocessing step aimed at reducing the input dimensionality.

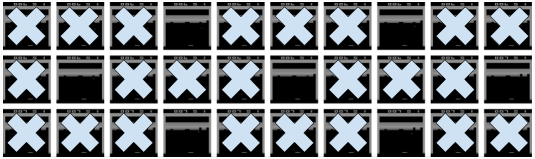
Firstly, I've converted the colors to grayscale, so to reduce the observation vector from $210 \times 160 \times 3$ to $210 \times 160 \times 1$. Then I've cropped the image, removing the score in the top of the screen, and lastly I've downsampled it to a 84×84 square. In this way we'll use significantly less memory while still keeping all necessary information.

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 also be able to tell the acceleration of the ball. I have chosen to stack 4 frames, like DeepMind did in [3], to be sure to have all necessary information.

So for each observation we are going to store the last 4 frames returned from the environment, and the input shape of the Neural Network will be $4 \times 84 \times 84$.

The frameskip

So we take every 4 consecutive frames? Actually, NOPE. In the environment `Deterministic` there's is also a frame skipping parameter, which is also set at 4. So only one every four screenshots is considered, and then we stack the "consecutive" frames that will be the input to the Neural Network. Consider the following sequence of screenshots, with the skipped frames denoted with an "X" over them:

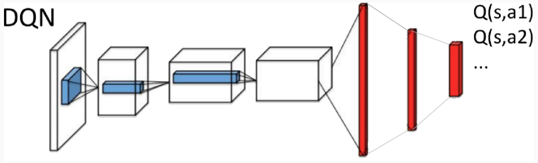


In the image there are only seven non-skipped frames: let's denote them as $x_1, x_2, \ldots, x_7$. We will use $s_1 = (x_1, x_2, x_3, x_4)$ as one state, then for the next state we will use $s_2 = (x_2, x_3, x_4, x_5)$. And so on and so forth. This is done to better discern motion, and since the network can see up to 12 frames ago you maximize the information it receives.

The Deep Q-Network

The DQN learns an approximation of the Q-table, which is a mapping between the states and actions that an agent will take. For every state we'll have four actions that can be taken. The environment provides the state, and the action is chosen by selecting the larger of the four Q-values for every state-action pairs predicted in the output layer.

- Input: $4 \times 84 \times 84$
- Conv2d: $32 \times 20 \times 20$
- Conv2d: $64 \times 9 \times 9$
- Conv2d: $64 \times 7 \times 7$
- Linear: $(7 \cdot 7 \cdot 64) \times 1$
- Linear: 512
- Output: 4



I've used a time decreasing ε-greedy strategy, dependent on the current time-step, to balance exploration and exploitation.

Double Q-Learning

Using only one network, when our weights update, our outputted Q-values will update, but so will our target Q-values since the targets are calculated using the same weights. So, as our Q-values move closer and closer to their targets Q-values, the targets Q-values continue to move further and further because we're using the same network to calculate both of these values. This makes the optimization appear to be chasing its own tail, which introduces instability. To solve this problem we will use two networks:

policy network its objective is to approximate the optimal policy by finding the optimal Q-function. It outputs an estimated Q-value for each possible action from the given input state.

target network its objective is to obtain the $\max$ Q-value for the next state, so to plug this value into the Bellman equation in order to calculate the target Q-value for the first state. It is a clone of the policy network: its weights are frozen with the policy network's weights, and we update them using the policy network's new weights every certain amount of time steps (an hyperparameter).

The Journey of Learning

Comparison with other projects

Unlike Cart-pole, Breakout does not have a specified reward threshold at which it's considered solved. Some consider it to be achieving consistently $200+$ reward.

The Deepmind paper trained for — as written in [3] —

> *a total of 50 million frames (that is, around 38 days of game experience in total)*

and it achieved a reward of $401.2\,(\pm 26.9)$ [Extended Data Table 2].

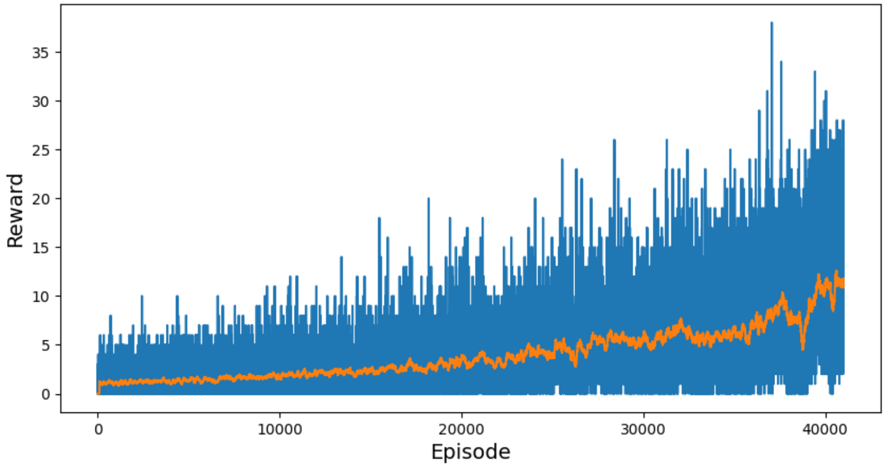
Looking around, on one post I've seen that it takes around $5K$ episodes or $1M$ time steps to start learning (getting a reward of around 10).

Another post trained for $5M$ steps, which took about ~ 40h on Tesla K80 GPU or ~ 90h on 2.9 GHz Intel i7 Quad-Core CPU, with a reward average of ~ 62.

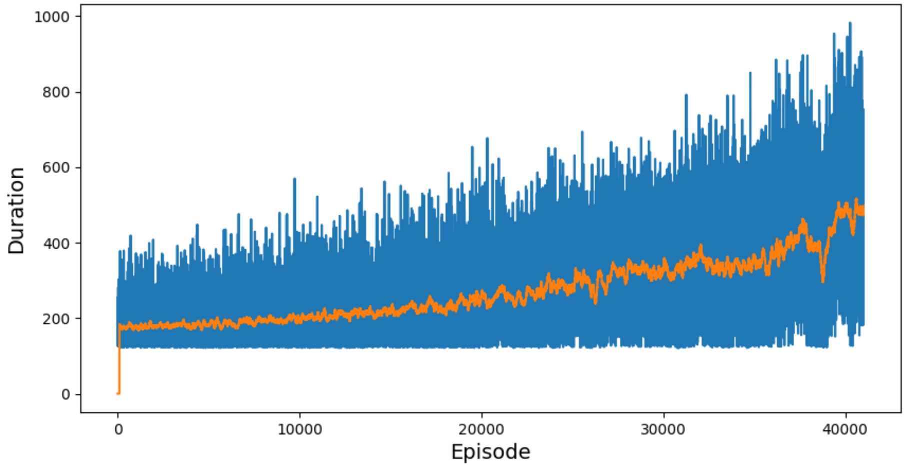
Instead, on Ulysses I started to gain a reward of around 10 at $40K$ episodes or $10M$ time steps.

After 102 job, for a total of $112:33:31$ hours, I got:

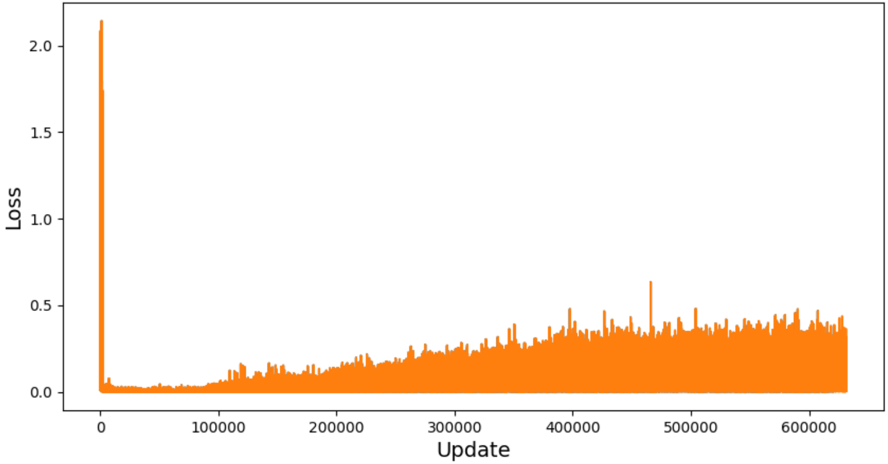
The rewards are growing



The durations are growing



The loss is decreasing (incomplete)



Improvements

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, which splits the network into two separate streams (one for estimating the state-value $V(s)$ and the other for estimating state-dependent action advantages $A(s, a_i)$), 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*. arXiv: *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*.

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*.

Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv preprint* (2013). arXiv: *1312.5602*.

To the game!