SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

Atari Breakout with $\text{LTL}_f/\text{LDL}_f$ Goals

ELECTIVE IN ARTIFICIAL INTELLIGENCE: REASONING ROBOTS

Professor:
Giuseppe De Giacomo

Students:
Ivan Bergonzani
Michele Cipriano
Armando Nania

Academic Year 2017/2018

Contents

1 Introduction

The aim of the project is to extend the work introduced in [**?**] testing the algorithms on a much harder environment, namely an Atari version of Breakout available in the `gym` framework.

The report is structured in the following way. Section 2 introduces the basic theory behind reinforcement learning, briefly describing the Q-Learning and the SARSA algorithms, which have been used to train the agent in the experiments. Section 3 describes $\text{LTL}_f/\text{LDL}_f$ non-Markovian rewards and how they can be used to train a RL agent. Section 4 describes the framework `gym` from OpenAI, which provides a level of abstraction on Arcade Learning Environment (ALE), which in turn provides a huge amount of classical Atari games. Section 5 discusses about the main implementation of the project, comparing the PyGame version of the Breakout game of the paper with one of the Atari Breakout versions of `gym`. Atari wrappers are introduced and robot features extractor, goal features extractor and temporal goals used in the implementation are presented in details. Section 6 presents all the experiments performed during the development of the project, highlighting differences between the two environments discussed in the previous section and all the changes applied to both environments in order to better understand weaknesses from both parts. The report ends with the conclusion the summarizes the work done and discusses about possible future works.

2 Reinforcement Learning

Reinforcement learning [**?**] is an area of machine learning which aims at studying how to develop agents that can interact with their environment maximing a cumulative reward. The environment can be formally defined as a Markov Decision Process (MDP), which is a tuple $\langle S, A, \delta, R \rangle$ where S is a finite set of states that can represent the environment, A is a finite set of actions that can be perform by an agent in the environment, δ is a probability function modeling the transition from a state to another when performing a certain action and R is a reward function which models the reward received by the environment when performing a certain action which makes the agent move from a state to another.

An interesting property of the MDP is that it satisfies the Markov property, hence, future states that will be reached by the agent do not depend on the past interaction of the environment, but just on the current state. This makes it possible to define the transition and the reward function depending only on the current state (and of course the action and the future state of interest).

This section considers two common reinforcement learning algorithm, namely Q-Learning and SARSA, which have been used in our experiments in order to train an agent interacting with an Atari Breakout environment (section 6).

2.1 Q-Learning

Q-Learning is a temporal difference (TD) algorithm that directly approximates the optimal action-value function. This method guarantees to find an optimal behaviour under the assumption that all the state-action pairs are updated infinitely many times. It is defined [**?**] by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big] \qquad (1)$$

Let's briefly discuss the implementation used in our project by studying the Python implementation (Algorithm 1). The algorithm is defined by the class `QLearning` that extends the abstract class `TDBrain`. The constructor of the class (lines 2-4) simply calls its parent constructor that will initialize the parameters of the object, hence, the observation space and the action space (`gym` objects), the strategy used by the policy function, which is ε-greedy by default and the hyperparameters γ, α and λ of the upper class. The abstract method inherited from `TDBrain` is `update_Q`, which should be implemented in order to define how to update the action-state table. The method (lines 6-21) simply follows Eq. 1.

Algorithm 1: Q-Learning algorithm Python implementation.

```
1  class QLearning(TDBrain):
2      def __init__(self, observation_space:Discrete,
           ↪ action_space, policy:Policy=EGreedy(),
3                   gamma=0.99, alpha=None, lambda_=0):
4          super().__init__(observation_space, action_space,
               ↪ policy, gamma, alpha, lambda_)
5
6      def update_Q(self, obs:AgentObservation):
7          state, action, reward, state2 = obs.unpack()
8
9          action2 = self.choose_action(state2)
10         Qa = np.max(self.Q[state2])
11         actions_star = np.argwhere(self.Q[state2] == Qa).
               ↪ flatten().tolist()
12
13         delta = reward + self.gamma * Qa - self.Q[state][
               ↪ action]
14         for (s, a) in set(self.eligibility.traces.keys()):
15             self.Q[s][a] += self.alpha.get(s,a) * delta *
                   ↪ self.eligibility.get(s, a)
16             if action2 in actions_star:
17                 self.eligibility.update(s, a)
18             else:
19                 self.eligibility.to_zero(s, a)
20
21         return action2
```

2.2 SARSA

A similar TD algorithm is the SARSA algorithm, which name comes from the fact that at each timestep a quintuple $\langle S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}\rangle$ is considered. As before, SARSA converges to an optimal action-value function under the assumption that all state-action pairs are updated infinitely many times. It is defined [**?**] by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big] \tag{2}$$

Let's briefly discuss the implementation used in our project by studying the implementation (Algorithm 2), as done before with the Q-Learning algorithm. Again, the algorithm is defined by the class `Sarsa` that extends the abstract class `TDBrain`. The constructor of the class (lines 2-4) calls its parent constructor initializing the parameters of the upper class exactly in the same way as the class `QLearning`. The class implements the inherited method `update_Q` by following Eq. 2 (lines 6-17).

Algorithm 2: SARSA algorithm Python implementation.

```
class Sarsa(TDBrain):
    def __init__(self, observation_space:Discrete,
        ↪ action_space, policy:Policy=EGreedy(),
                 gamma=0.99, alpha=None, lambda_=0.0):
        super().__init__(observation_space, action_space,
            ↪ policy, gamma, alpha, lambda_)

    def update_Q(self, obs:AgentObservation):
        state, action, reward, state2 = obs.unpack()

        action2 = self.choose_action(state2)
        Qa = self.Q[state2][action2]

        delta = reward + self.gamma * Qa - self.Q[state][
            ↪ action]
        for (s, a) in set(self.eligibility.traces.keys()):
            self.Q[s][a] += self.alpha.get(s,a) * delta *
                ↪ self.eligibility.get(s, a)
            self.eligibility.update(s, a)

        return action2
```

3 $\text{LTL}_f/\text{LDL}_f$ Non-Markovian Rewards

Recently, non-Markovian reward decision processes (NMRDPs) have attracted interest in the scientific community because of the possibility of specifying them as MDPs with $\text{LTL}_f/\text{LDL}_f$ non-Markovian rewards [**?**]. In particular, it is possible to model the problem with two separate representations of the world, one for the agent (low-level) and one for the goal (expressed in terms of high-level fluents).

This section presents the approach used in [**?**], where an efficient method has been developed in order to work with NMRDPs. The theory behind the main idea is quickly described and an example on a theoretical Breakout environment is discussed in order to be used in the following sections easily.

3.1 Theoretical Background

Before describing the problem, let's give a formal definition of NMRDP. A non-Markovian reward decision process is a tuple $M = \langle S, A, \delta, \bar{R} \rangle$, with S finite set of states that can represent the environment, A is a finite set of actions that can be performed by an agent in the environment, δ is a probability function modeling the transition from a state to another when performing a certain action and $\bar{R} : (S \times A)^* \to \mathbb{R}$ is a function from finite state-action sequences (traces) to real-values that represents the reward given by the environment when performing a certain state-action sequence. Specifying a non-Markovian reward function explicitely is difficult even when considering a finite number of traces. Luckily, the $\text{LTL}_f/\text{LDL}_f$ formalism allows to specify $\bar{R}$ implicitely using a set of pairs $\{(\phi_i, r_i)\}_{i=1}^m$ with ϕ_i boolean proposition over the components of the state vector and r_i such that, given a current trace $\pi = \langle s, a_1, \ldots, s_{n-1}, a_n \rangle$, the agent receives at s_n a reward r_i if ϕ_i is satisfied by π, hence:

$$\bar{R}(\pi) = \begin{cases} r_i & \text{if } \pi \vDash \phi_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Since the NMRDP rewards are based on traces, instead of state-action pairs, typical learning algorithms like Q-learning or SARSA cannot be used. Nevertheless, it has been shown [**?**] that for any NMRDP $M = \langle S, A, \delta, \{(\phi_i, r_i)\}_{i=1}^m \rangle$ there exists an MDP $M' = \langle S', A', \delta' R' \rangle$ that is equivalent to M. The idea behind the proof consists in starting from an initial decision process $M_{ag}^{goal} = \langle S, A, R, \mathcal{L}, \delta_{ag}^g, \{(\phi_i, r_i)\}_{i=1}^m \rangle$ with $\text{LTL}_f/\text{LDL}_f$ goals (with $\mathcal{L}$ set of of configuration of the high-level features needed for expressing ϕ_i), transform it into a

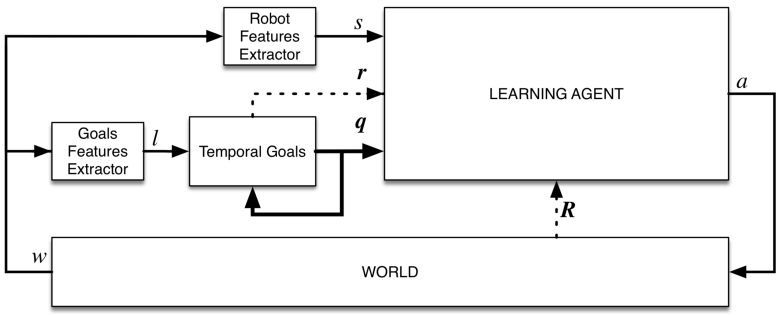


Figure 1: Pipeline describing how the agent is interacting with the world and how the robot features extractor and the goal features extractor are used in order to handle non-Markovian rewards.

NMRDP in order to further transform it into a MDP where it is possible to execute learning algorithms such as Q-learning. All the details are out of the scope of the project and are discussed in [**?**]. The set of states S is used to express low-level features of the agent.

The new MDP is extended by using automata needed to track the satisfiability of a $\text{LTL}_f/\text{LDL}_f$ formulas, as explained in [**?**]. Each formula is, hence, associated with an automaton that accepts exactly the traces satisfying the formula itself.

Once the MDP has been determined it is possible to train a model that will be able to manage non-Markovian rewards as well. Fig. 1 shows a pipeline of the agent interacting with the world. In particular, a robot feature extractor will analyze the state of the world w in order to pass a representation s to the agent, a goal feature extractor will extract higher-level features l of the world that can be used to determine if a temporal goal has been achieved or not. In positive case a reward r is given to the agent. The evaluation q of the formula that specifies the temporal goals is given to the learning agent as well, in order to construct an extended state of the MDP as specified by the theory above. Each time the learning agent performs an action a, the environment changes and it gives to the agent a reward R.

3.2 Examples

A typical example is breaking the bricks of a theoretical Breakout game in a certain order. Let's consider a 3×3 Breakout. A good idea is to use $\text{LTL}_f/\text{LDL}_f$ goals to help the agent break to bricks from top to bottom in order to make it create a hole on the left (or the right) and let the ball do all the job without making any movements when it reaches the top part of the environment. The

idea is, hence, to help the agent learn the most effective strategy, which should also decrease the training time. Let's consider a formula that checks weather a row has been broken or not:

$$\begin{aligned}\langle(\neg\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2)^*; (\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2); (\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2)^*; \\ (\varphi_0 \wedge \varphi_1 \wedge \neg\varphi_2); (\varphi_0 \wedge \varphi_1 \wedge \neg\varphi_2)*; (\varphi_0 \wedge \varphi_1 \wedge \varphi_2)\rangle tt\end{aligned} \tag{4}$$

which, given φ_0 first row from the top broken, φ_1 second row from the top broken and φ_2 third row from the top broken, can be interpreted as:

- $(\neg\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2)^*$: initially, and for an indefinite amout of time, none of the rows have been broken;
- $(\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2)$: then, the first row from the top has been broken but the remaining two are intact;
- $(\varphi_0 \wedge \neg\varphi_1 \wedge \neg\varphi_2)^*$: then, for an indefinite amount of time, the first row remains broken and the remaining two remain intact;
- $(\varphi_0 \wedge \varphi_1 \wedge \neg\varphi_2)$: then, the first two rows from the top have been broken and the remaining one at the bottom is intact;
- $(\varphi_0 \wedge \varphi_1 \wedge \neg\varphi_2)*$: then, for an indefinite amount of time, the first two rows from the top remain broken and the last one remain instact;
- $(\varphi_0 \wedge \varphi_1 \wedge \varphi_2)$: in the end, all the rows have been broken, hence, all the bricks have been broken.

In general, the formula $\langle\rho\rangle\phi$ states that, from the current step in the trace, there exists an execution satifying ρ such that in its last step ϕ is satisfied. In the example above, at the end of the execution all the rows have been broken.

4 OpenAI Gym

OpenAI `gym` [?] is a toolkit for developing and comparing reinforcement learning algorithms, without making assumptions about the structure of the agent interacting with the environment, in order to keep development flexible to updates on both sides.

4.1 Framework

The framework of `gym` allows to interact easily with an environment, giving to developers the tools they need to perform actions and to observe the state of the environment itself. In this way it is possible to focus more on the development of the agent without spending time on the structure of the world.

`gym` makes it possible to interact with multiple kinds of environments. Among these, the authors of the framework developed the support for Arcade Learning Environment [?], which includes all the classing Atari games, including Breakout, which has been used in this project.

4.2 Examples

Let's consider a simple example to understand how `gym` works and how the framework can be used to interact with an environment. The description will follow Algorithm 3.

Algorithm 3: Example of a random interaction with the `gym` environment `BreakoutNoFrameskip-v4`, used also in our experiments of subsection 6.

```
1  import gym
2
3  env = gym.make("BreakoutNoFrameskip-v4")
4  env.reset()
5
6  for _ in range(1000):
7    env.render()
8    action = env.action_space.sample() # takes random actions
9    observation, reward, done, info = env.step(action)
10   if done == True:
11     env.reset()
12
13 env.close()
```

Initially (line 1) the framework is imported. Then (line 3-4) an environment is created specifying its name and initializing it. The program makes a random agent interact randomly with the environment for 1000 timesteps (lines 6-11) before closing the environment. Line 7 renders the current observation of the

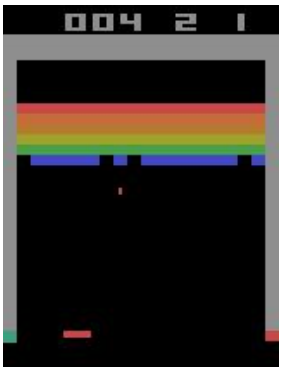


Figure 2: Observation of a frame of the environment `BreakoutNoFrameskip-v4`.

environment on screen, line 8-9 performs a random action between those available in this Brekout version, note that the method `step` return an `observation` (shown in Fig. 2), which is an array of pixels that represent the current state of the environment, a `reward`, which is a value return by the game after performing the specified action `action`, a boolean value `done`, which is `True` when the game is over, `False` otherwise, and `info` which contains extra information about the game. Lines 10-11 handles the case when the game is over, resetting the environment.

5 Atari Breakout

This section contains the main part of the project, describing in detail our starting point, how the program has been developed, the results achieved and the comparison with the original implementation, which uses a non-Atari version of the game Breakout [**?**] with the same number of bricks used in the Atari implementation of `gym`, already introduced in section 4.

Initially, the non-Atari version of Breakout (built on PyGame) is introduced, its implementation is discussed and the results on the training with a brick matrix of dimension 6×18 are presented. Then the `gym` environment `BreakoutNoFrameskip-v4` is presented and compared with the PyGame breakout. A detailed description of the implementation of the project is described, discussing in detail how the features have been extracted from the environment (both the robot and the goal features), how temporal goals have been used to evaluate the state of the bricks and how everything is connected together in order to make it work with `gym`.

5.1 PyGame Breakout

As introduced above, in [**?**] a PyGame version of the Breakout game has been used in order to test the algorithms introduced in the paper. The implementation of this Breakout easily allows to determine the state of the environment, saving the status of the bricks (either in the scene or broken), the position of the ball, the direction of the ball and the position of the paddle. This makes it possible to reduce a lot the computational time of the implementation of the agent since the data it receives are already preprocessed in order to have a complete overview of the environment, allowing to focus on higher-level reasoning tasks.

Originally, the paper focused on a Breakout environment with a brick matrix of dimension 4×5. Since the Atari version of Breakout is dealing with a brick matrix of dimension 6×18, a new test has been performed to make the two environments comparable, in order to better understand the potentiality of $\text{LTL}_f/\text{LDL}_f$, the use of non-Markovian rewards and how to approach complex `gym` environments in the future.

5.2 Arcade Learning Environment

This work aims at comparing the `gym` environment `BreakoutNoFrameskip-v4`, already introduced in section 4 with the non-Atari Breakout used in [**?**]. In last

years, the reinforcement learning community has grown a lot thanks to the introduction of `gym` and deep reinforcement learning algorithms [**?**] that managed to easily solve complex games that are considered difficult also for humans, often achieving better results than expert human gamers. More and more algorithms are introduced every year, exploiting GPU resources and managing to solve harder games like Montezuma Revenge [**?**]. The popularity of deep reinforcement learning begun with the introduction of Arcade Learning Environment (ALE) that includes most famous arcade Atari games [**?**].

The main characteristics of `gym` (or ALE) environments is that the world can be observed only from the pixels of the screen, putting the algorithms at the same level of the human, that can only observe the display while playing. This makes the game a lot more complex since a more abstract reasoning strategy is needed in order to solve the game. This hypothesis should make `gym` games a lot harder than the non-Atari version of Breakout that has been used to test algorithms that work with non-Markovian rewards.

5.2.1 Atari Wrappers

Before introducing the main part of the implementation it is important to discuss the use of Atari wrappers, introduced with the OpenAI baselines [**?**], that simplify the interaction with the environment, lightening the code and managing important aspects of the game. The wrappers used in the project, as it is possible to notice from Algorithm 4, are the following:

- `EpisodicLifeEnv`: make an "end of life" be the end of the episode resetting the environment only on the true game over, this helps in value estimation;
- `FireResetEnv`: use "Fire" as starting action in order to launch the ball;
- `MaxAndSkipEnv`: returns only the skip-th frame, this reduces the amount of frame the agent has to deal with (set to 4 in the main experiments).

Note that line 1 defines the `gym` environment used in the project, namely `BreakoutNoFrameskip-v4`. All the wrappers extend the class `gym.Wrapper`.

Algorithm 4: Initialization of the Atari Breakout environment and the use of Atari wrappers from OpenAI.

```
1 env = gym.make("BreakoutNoFrameskip-v4")
2 env = EpisodicLifeEnv(env)
3 env = FireResetEnv(env)
4 env = MaxAndSkipEnv(env, skip=4)
```

5.3 Implementation

Following the pipeline introduced in [**?**] and described in Fig. 1, the robot features extractor, the goal features extractor and the temporal goals are described. In particular, their implementation makes use of OpenCV [**?**] in order to deal with images easily and a Python implementation of FLLOAT [**?**] to deal with $\text{LTL}_f/\text{LDL}_f$ formulas. Note that the abstract class `TemporalEvaluator` in Algorithm 7 makes use of the libraries Pythomata [**?**] and RLTG [**?**] to build automata from the desired $\text{LTL}_f/\text{LDL}_f$ formulas in order to make it possible to receive non-Markovian rewards.

5.3.1 Robot Features Extractor

The implementation of the robot features extractor is shown in Algorithm 5. The class extends the abstract class `BreakoutRobotFeatureExtractor`, which has been developed just to keep a consistent structure among other implementation of other robot features extractors. The class implements two methods: the constructor `__init__`, which takes as input a `gym` object defining the space of the observation, and the method `_extract`, which takes as input the observation `input` coming from the environment and a dictionary `kwargs` containing other optional parameters.

The constructor `__init__` defines the robot features space (lines 4-7) with two `gym.spaces.Discrete` objects defined with values 287 and 157, respectively the maximum possible values (extreme excluded) that the position of the ball can have with respect to the paddle and the height of the ball. Then, internal representation of the position of the ball and of the paddle are initialized (lines 9-12). The boolean `self.still_image` is used to avoid repetitions of the same observation in the method `_extract` since the same observation is considered twice. In the end, the superconstructor is called (line 14) in order to finalize the construction of the object.

The method `_extract` checks weather the image has been already seen or not (lines 17-19) has explained above. Then, the position of the paddle is extracted from the observation (an image) on lines 20-35. Initially, only the bottom part of the image is extracted from the observation (line 21), it is then converted to a gray-scale image (line 22) so that it is possible to apply a threshold function in order to make the paddle white and the rest of the image black (line 23). In this way it is possible to find contours of the objects contained in that part of the image (there should be only one actually since only the bottom part is considered) and extract the centroid of the paddle, in this way the variable `paddleX` is updated. Similarly, the position of the ball is extracted (lines 37-

69) from the upper part of the image. Here, it is important to actually check that the centroid is part of the ball since there could be objects that are part of the bricks. Fortunately the ball has a unique RGB color $(200, 72, 72)$ which simplifies this step.

Finally, the internal representation of the object is updated (lines 71-73) and a tuple containing data specified in the constructor is returned (line 75).

Algorithm 5: Robot feature extractor Python implementation.

```
1  class BreakoutNRobotFeatureExtractor(
       ↪ BreakoutRobotFeatureExtractor):
2
3      def __init__(self, obs_space):
4          robot_feature_space = Tuple((
5              Discrete(287),
6              Discrete(157),
7          ))
8
9          self.prev_ballX = 0
10         self.prev_ballY = 0
11         self.prev_paddleX = 0
12         self.still_image = True
13
14         super().__init__(obs_space, robot_feature_space)
15
16     def _extract(self, input, **kwargs):
17         self.still_image = not self.still_image
18         if self.still_image:
19             return (self.prev_ballX-self.prev_paddleX+143,
                   ↪ self.prev_ballY)
20         # Extract position of the paddle:
21         paddle_img = input[189:193,8:152,:]
22         gray = cv2.cvtColor(paddle_img, cv2.COLOR_RGB2GRAY)
23         thresh = cv2.threshold(gray, 60, 255, cv2.
               ↪ THRESH_BINARY)[1]
24         cnts = cv2.findContours(thresh.copy(), cv2.
               ↪ RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
25         cnts = cnts[0] if imutils.is_cv2() else cnts[1]
26         min_distance = np.inf
27         paddleX = self.prev_paddleX
28         for c in cnts:
29             M = cv2.moments(c)
30             if M["m00"] == 0:
31                 continue
32             pX = int(M["m10"] / M["m00"])
33             if abs(self.prev_paddleX - pX) < min_distance:
34                 min_distance = abs(self.prev_paddleX - pX)
35                 paddleX = pX
36
37         # Extract position of the ball:
38         ballX = self.prev_ballX
39         ballY = self.prev_ballY
```

```
40          ballspace_img = input[32:189,8:152,:]
41          lower = np.array([200, 72, 72], dtype=np.uint8)
42          upper = np.array([200, 72, 72], dtype=np.uint8)
43          mask = cv2.inRange(ballspace_img, lower, upper)
44          cnts = cv2.findContours(mask.copy(), cv2.
                ↪ RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
45          cnts = cnts[0] if imutils.is_cv2() else cnts[1]
46          for c in cnts:
47              M = cv2.moments(c)
48              # Avoid to compute position of the ball if M["
                    ↪ m00"] is zero:
49              if M["m00"] == 0:
50                  continue
51              # Calculate the centroid
52              cX = int(M["m10"] / M["m00"])
53              cY = int(M["m01"] / M["m00"])
54              # Check that the centroid is actually part of
                    ↪ the ball:
55              left_black = False
56              right_black = False
57              if cX > 3:
58                  if ballspace_img[cY][cX-3][0] != 200 or \
59                      ballspace_img[cY][cX-3][1] != 72 or \
60                      ballspace_img[cY][cX-3][2] != 72:
61                      left_black = True
62              else:
63                   if ballspace_img[cY][cX+3][0] != 200 or \
64                       ballspace_img[cY][cX+3][1] != 72 or \
65                       ballspace_img[cY][cX+3][2] != 72:
66                       right_black = True
67              if left_black or right_black:
68                  ballX = cX
69                  ballY = cY
70
71          self.prev_ballX = ballX
72          self.prev_ballY = ballY
73          self.prev_paddleX = paddleX
74
75          return (self.prev_ballX - self.prev_paddleX + 143,
                ↪ self.prev_ballY)
```

5.3.2 Goal Features Extractor

The implementation of the goal features extractor is shown in Algorithm 6. The class extends the abstract class `FeatureExtractor`, which has been developed just to keep a consistent structure among other implementation of other goal features extractors. The class implements two methods: the constructor `__init__`, which takes as input a `gym` object defining the space of the observation and the number of rows and columns composing the bricks matrix, and the method `_extract`, which takes as input the observation `input` coming from the

environment and a dictionary `kwargs` containing other optional parameters.

The constructor `__init__` saves the number of rows and columns (lines 3-4) of the bricks matrix used in its internal representation and defines the space used by the method `_extract` to return objects (a simple representation of the bricks matrix). In the end, the superconstructor is called (line 6) in order to finalize the construction of the object.

The method `_extract` simply returns a numpy representation of the bricks matrix seen from the observation, which is in turn given by the environment each time the agent takes an action. The algorithm cycles on the pixels of the image representing the observation checking weather they are black or not. In particular, it is possible to determine this by checking the two pixels on the upper left and upper right part of the brick comparing their color (lines 10-20) with the background (black). If both of them are black, then the brick has already been destroyed, otherwise it is still present in the environment. Note that it is necessary to check both pixels because one of the two could be different due to the presence of the ball during the game. As mentioned above, the method return a 6×18 numpy matrix representing the status of the bricks (each element will have value 0 if the brick has been destroyed, 1 otherwise).

Algorithm 6: Goal feature extractor Python implementation.

```
1  class BreakoutGoalFeatureExtractor(FeatureExtractor):
2      def __init__(self, obs_space, bricks_rows=6,
           ↪ bricks_cols=18):
3          self.bricks_rows = bricks_rows
4          self.bricks_cols = bricks_cols
5          output_space = Box(low=0, high=1, shape=(
               ↪ bricks_cols, bricks_rows), dtype=np.uint8)
6          super().__init__(obs_space, output_space)
7
8      def _extract(self, input, **kwargs):
9          bricks_features = np.ones((self.bricks_cols, self.
               ↪ bricks_rows))
10         for row, col in itertools.product(range(self.
               ↪ bricks_rows), range(self.bricks_cols)):
11             # Pixel of the observation to check:
12             px_upper_left  = int( 8 + 8 * col)
13             py_upper_left  = int(57 + 6 * row)
14             px_upper_right = int(15 + 8 * col)
15             py_upper_right = int(57 + 6 * row)
16
17             # Checking max because the input has 3 channels
                   ↪ :
18             if max(input[py_upper_left][px_upper_left]) ==
                   ↪ 0 or \
19                 max(input[py_upper_right][px_upper_right])
                       ↪ == 0:
```

```
20                  bricks_features[col][row] = 0
21
22          return bricks_features
```

5.3.3 Temporal Goals

The implementation of temporal goals is shown in Algorithm 7. It is divided in three parts:

- `get_breakout_lines_formula`: a function that determines the $\text{LTL}_f/\text{LDL}_f$ formula as a string, in order to be later parsed by the FLLOAT parser;
- `BreakoutCompleteLinesTemporalEvaluator`: a temporal evaluator class that handles rows and columns as lines;
- `BreakoutCompleteRowsTemporalEvaluator`: a temporal evaluator class that extends `BreakoutCompleteLinesTemporalEvaluator` and work on rows, this is the main class used in the project regarding temporal goals.

In particular, `get_breakout_lines_formula` (lines 1-15) generates a $\text{LTL}_f/\text{LDL}_f$ formula as explained in Section 3, but extended to a general case of a Breakout consisting of a bricks matrix of size $n \times m$.

`BreakoutCompleteLinesTemporalEvaluator` extends the RLTG abstract class `TemporalEvaluator` which contains the abstract method `fromFeaturesToPropositional`. Its constructor `__init__` (lines 20-38) parses the $\text{LTL}_f/\text{LDL}_f$ specified by the function `get_breakout_lines_formula` (lines 22-30) in order to pass it to the superconstructor (lines 33-38) that will manage the construction of the automata using Pythomata and RLTG libraries. The method `fromFeaturesToPropositional`, inherited from `TemporalEvaluator`, maps the bricks matrix to a propositional formula in order to update the automata while training the agent.

`BreakoutCompleteRowsTemporalEvaluator` has the same structure of the previous class with the exception of the boolean `self.bottom_up` that specifies weather breaking the rows from top to bottom or viceversa. For all the experiments of the project this variable has been set to `False` in order to incite the agent to find a strategy that makes the ball go to the upper part of the environment in order to break the bricks without making any effort.

Algorithm 7: $\text{LTL}_f/\text{LDL}_f$ formulas Python implementation.

```
1  def get_breakout_lines_formula(lines_symbols):
2      # Generate the formula string
3      # E.g. for 3 line symbols:
4      # "<(!l0 & !l1 & !l2)*;(l0 & !l1 & !l2);(l0 & !l1 & !l2
           ↪ )*;(l0 & l1 & !l2); (l0 & l1 & !l2)*; l0 & l1 &
           ↪ l2>tt"
```

```
5       pos = list(map(str, lines_symbols))
6       neg = list(map(lambda x: "!" + str(x), lines_symbols))
7
8       s = "(%s)*" % " & ".join(neg)
9       for idx in range(len(lines_symbols)-1):
10          step = " & ".join(pos[:idx + 1]) + " & " + " & ".
                ↪ join(neg[idx + 1:])
11          s += ";({0});({0})*".format(step)
12      s += ";(%s)" % " & ".join(pos)
13      s = "<%s>tt" % s
14
15      return s
16
17  class BreakoutCompleteLinesTemporalEvaluator(
        ↪ TemporalEvaluator):
18      """Breakout temporal evaluator for delete columns from
            ↪ left to right"""
19
20      def __init__(self, input_space, bricks_cols=3,
            ↪ bricks_rows=3, lines_num=3, gamma=0.99,
            ↪ on_the_fly=False):
21          assert lines_num == bricks_cols or lines_num ==
                ↪ bricks_rows
22          self.line_symbols = [Symbol("l%s" % i) for i in
                ↪ range(lines_num)]
23          lines = self.line_symbols
24
25          parser = LDLfParser()
26
27
28          string_formula = get_breakout_lines_formula(lines)
29          print(string_formula)
30          f = parser(string_formula)
31          reward = 10000
32
33          super().__init__(BreakoutGoalFeatureExtractor(
                ↪ input_space, bricks_cols=bricks_cols,
                ↪ bricks_rows=bricks_rows),
34                           set(lines),
35                           f,
36                           reward,
37                           gamma=gamma,
38                           on_the_fly=on_the_fly)
39
40      @abstractmethod
41      def fromFeaturesToPropositional(self, features, action,
            ↪  *args, **kwargs):
42          """map the matrix bricks status to a propositional
                ↪ formula
43          first dimension: columns
44          second dimension: row
45          """
46          matrix = features
47          lines_status = np.all(matrix == 0.0, axis=kwargs["
```

```
            ↪ axis"])
48          result = set()
49          sorted_symbols = reversed(self.line_symbols) if
                ↪ kwargs["is_reversed"] else self.line_symbols
50          for rs, sym in zip(lines_status, sorted_symbols):
51              if rs:
52                  result.add(sym)
53
54          return frozenset(result)
55
56  class BreakoutCompleteRowsTemporalEvaluator(
        ↪ BreakoutCompleteLinesTemporalEvaluator):
57      """Temporal evaluator for complete rows in order"""
58
59      def __init__(self, input_space, bricks_cols=3,
            ↪ bricks_rows=3, bottom_up=True, gamma=0.99,
            ↪ on_the_fly=False):
60          super().__init__(input_space, bricks_cols=
                ↪ bricks_cols, bricks_rows=bricks_rows,
                ↪ lines_num=bricks_rows, gamma=gamma,
                ↪ on_the_fly=on_the_fly)
61          self.bottom_up = bottom_up
62
63      def fromFeaturesToPropositional(self, features, action,
            ↪  *args, **kwargs):
64          """complete rows from bottom-to-up or top-to-down,
                ↪ depending on self.bottom_up"""
65          return super().fromFeaturesToPropositional(features
                ↪ , action, axis=0, is_reversed=self.bottom_up
                ↪ )
```

6 Experiments

Although the algorithm applied to the PyGame version of Breakout gives excellent results, the same algorithm applied to the Atari version does not manage to solve the game correctly, without even satifying the first temporal goal. In this section we will analyze the results obtained, also showing some (not all) of the tests that were carried out in an attempt to find out what was the cause of the intractability of Atari's environment. In particular, we will first show the results on PyGame and Atari environments, comparing them. Then we will explain the changes made to the PyGame version during the subsequent test phases. Finally we will draw conclusions from these experiments, analyzing in detail the problems encountered, and we will show some possible changes to be used to address these problems.

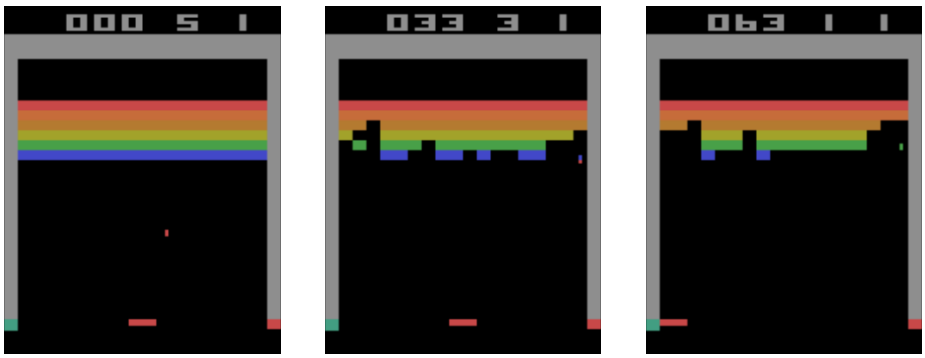


Figure 3: Sampled frames from the Atari Environment experiments

6.1 PyGame and Atari Results

The method applied to the environment of PyGame gives excellent results already from 50,000 training periods, while the results obtained by applying the same method to the Atari environment are lower, even after 200,000 epochs. These are the results of the method applied to the two environments:

As you can see, the results are much lower in the second case. In an attempt to improve the Atari results, changes in the code have been tested: we have reduced the size of the state space, either the ball space or the paddle space, or its movement space; we have tried to completely remove the information concerning the movement of the paddle; we tested a one-shot version of Atari, in order to make it similar to PyGame and finally we have tried combinations of the above ones. These are the results of these tests:

Given the number of tests to be performed, we performed them with a low

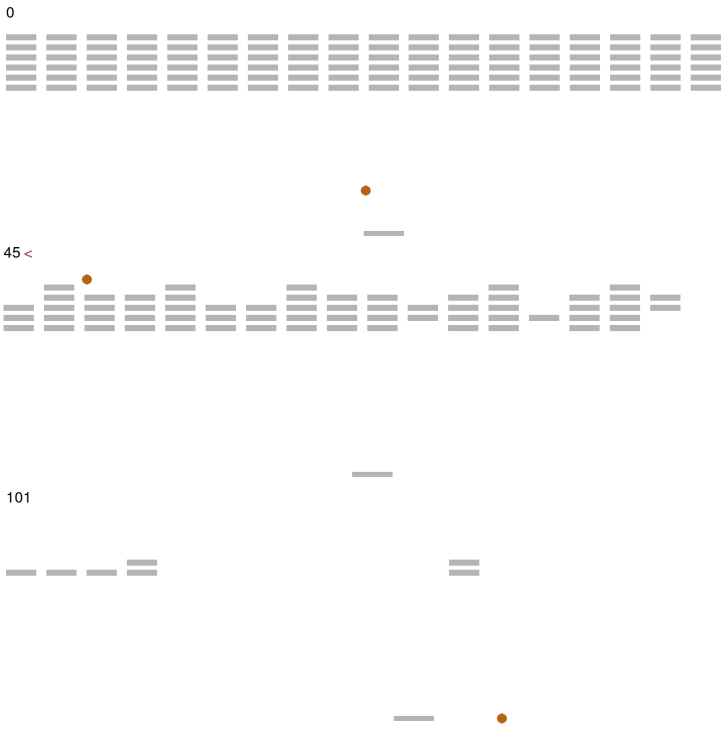
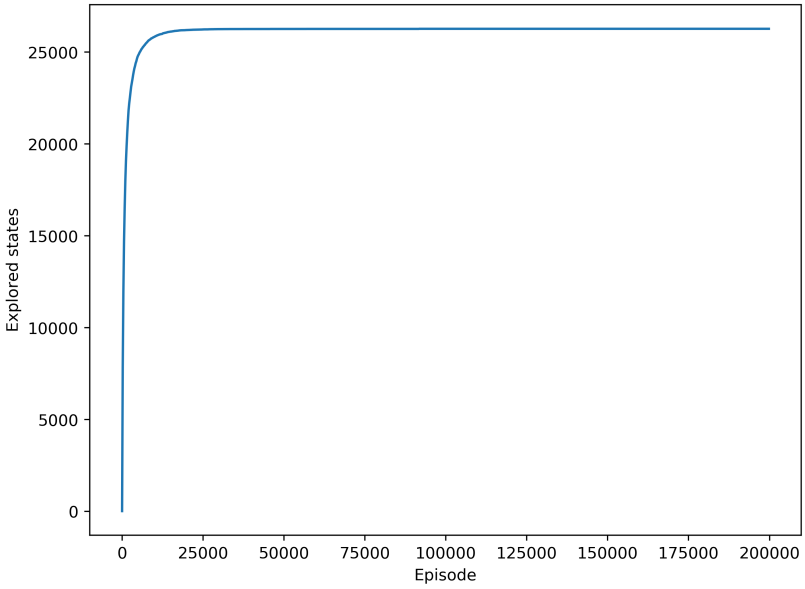


Figure 4: Sampled frames from the PyGame Environment experiments

number of training periods, 5000, for obvious reasons of time. So it is necessary to take this data with a grain of salt. But it is still possible to see a trend: the reduction of the state space does not significantly improve the result you get, while giving Atari environment the opportunity to play all his 5 lives worsens the score obtained at the end of the first life, but increases the total score. This is a consequence of the fact that there are no penalties in the reward associated with losing a life: there are only positive rewards given by the destruction of a brick or the achievement of a temporal goal. This makes possible scenarios in which, after losing a life, the paddle succeeds in destroying a brick in the next life, which leads to a reward for the sequence of state-action pairs that led to that result. All this makes the training of AI very random, because the life in which it will take the reward is random, with the risk of losing lives uselessly. We will draw further conclusions later in this section.

6.2 Modified PyGame

In the previous point we tried to identify strategies for managing the state space and lives to improve the score, but despite this the difference between the two environments is still large. It is true that PyGame Breakout has a very



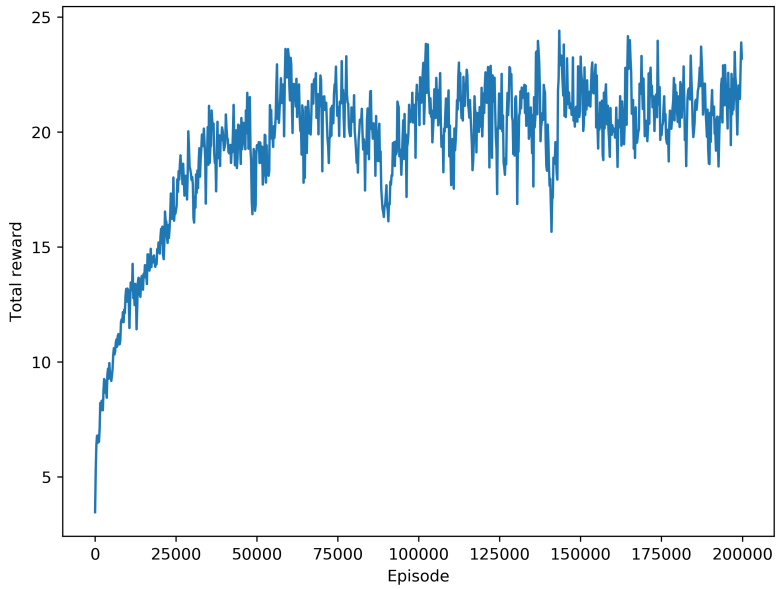
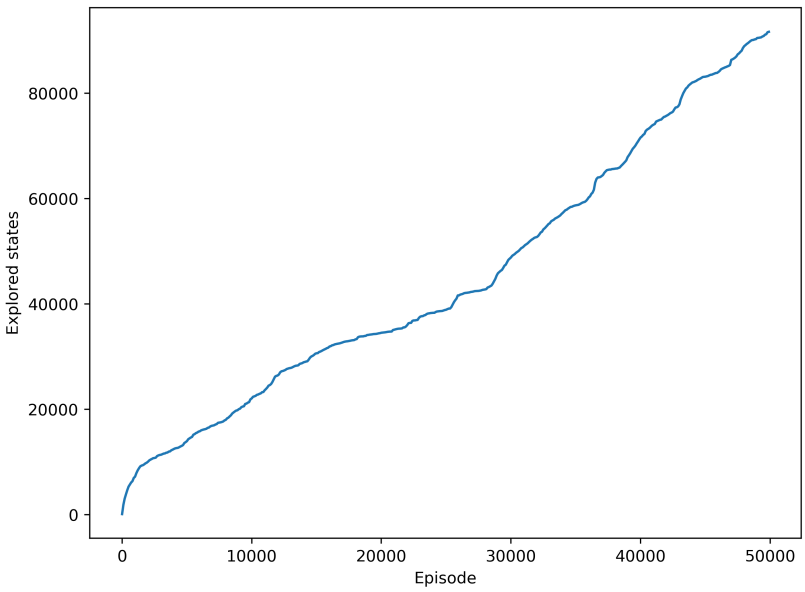


Figure 5: Plots from the Atari Environment experiments. The reinforcement learning algorithm used in this experiment is SARSA.



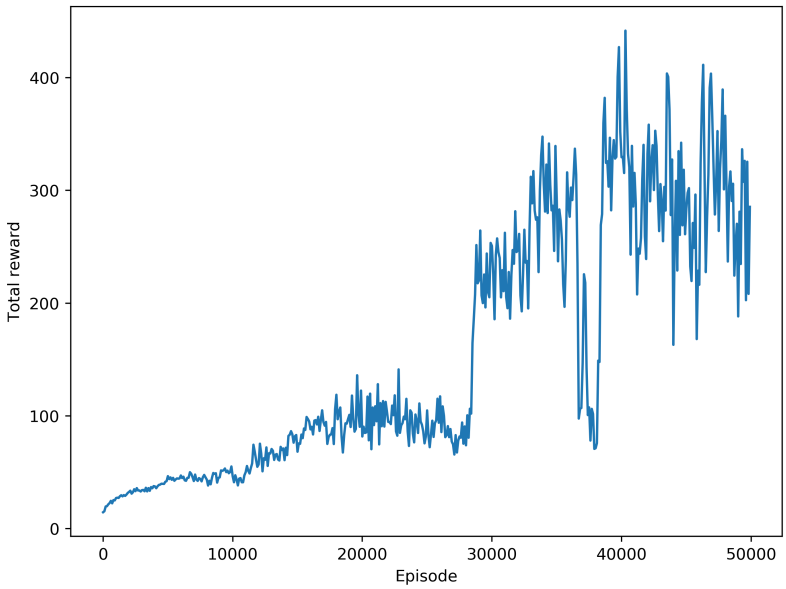


Figure 6: Plots from the PyGame Environment experiments. The reinforcement learning algorithm used in this experiment is SARSA.

Environment	Epochs	Explored States	Total Reward
Atari	100,000	26,291	46
Atari	200,000	26,271	63
PyGame	50,000	91,821	1010

Table 1: Comparison of the results obtained. All the tests shown in the table were performed with the SARSA algorithm. The tests of the Q-Learning algorithm application did not show significant changes in the final results, therefore they were not shown. It is important to note that the reward is assigned differently in the two environments: PyGame returns a reward of 10 for the destruction of any brick. Atari, on the other hand, returns a reward equal to 1, 4 or 7 for the destruction of a brick, depending on the column in which the brick is located: the bricks higher up give a higher reward.

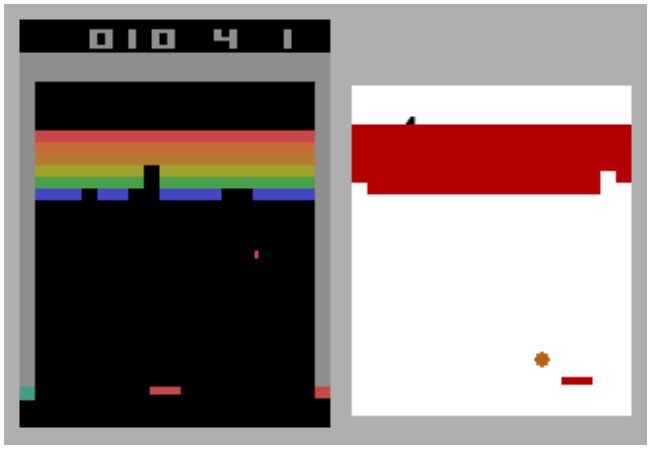


Figure 7: Comparison between the Atari environment and the modified version of the PyGame environment

small state space, but nevertheless decreasing the state space in Atari Breakout to make it similar to PyGame does not produce any improvements. So instead of making Atari similar to PyGame, let's try to do the opposite. There are indeed considerable structural differences between the two environments: the size of paddle, ball, bricks, window and placements of objects in the field. Moreover, the movement of the ball is much slower in PyGame than in Atari: the ball, to reach the first bricks from the paddle, takes 35 frames in PyGame, and only 15 in Atari. In 1500 frames of PyGame it can only destroy 10 bricks, in 1500 frames of Atari it destroys 29 bricks, even 47 with 2000 frames! Remind that a frame corresponds to a state in which you choose an action to move to the next state. In order to better compare the two environments, in an attempt to understand the problems inherent to Atari, we gradually approximated PyGame to Atari, to make the two environments identical.

To do this we changed the PyGame code at some points. In doing so it was highlighted a bug that was already present in PyGame, but that did not show

Ball X	Ball Y	Angle	Paddle	Move	Lives Count	Frame Skip	Score
144	157	10	144	49	1	4	6
144	157	10	144	49	5	4	4-20
8	18	10	8	49	1	4	5
144	157	10	144	10	1	4	7
144	157	10	144	0	1	4	7
144	157	10	144	0	5	4	2-16
8	18	10	8	10	1	4	6
8	18	10	8	0	1	4	2
8	18	10	8	0	5	4	5-15
144	157	10	144	13	5	14	19
144	157	10	144	13	5	10	24
144	157	10	144	13[1]	5	10	16
144	157	10	144	121	5	10	24

Table 2: The first five columns indicate the state space of the related variables, the sixth column indicates the number of lives per game, the sequential column indicates the number of game-frames for each state-frame (see point 6.3.2). The last column indicates the score at the end of the game, expressed as the number of bricks destroyed. In some cases (only those with more than one life) we have specified both the score at the end of the first life and the score at the end of the last life, separated by a dash. All tests were performed with 5000 training epochs. The results of these tests indicate that the score improves with the increase of the lives and the frame skip, with an optimal number of frame skip equal to 10 (not all the tests we performed have been included in this table), while the size of the state space has little influence on the score obtained. [1] This test was performed with the same hyperparameters as the previous test, but with a different calculation of the move variable: instead of distributing the 13 states of the move variable from -60 to +60 motion pixels, they were distributed over the range [-40 , +40], cropping excess values.

up in its original version:

Algorithm 8: A more sophisticated version of the ball rebound algorithm.

```
def hitDetect(self):
    ##COLLISION DETECTION

    # [...]

    for brick in self.bricks:
        if brick.rect.colliderect(ball_rect):
            if ((not self.se_brick is None)):
                self.se_brick.play()

            self.score = self.score + 1
            self.brick_hit_count += 1
            self.bricks.remove(brick)
            self.last_brikcsremoved.append(brick)
            self.bricksgrid[(brick.i, brick.j)] = 0

            # bug correction begin

```

```
            min_distance = math.fabs(self.ball_x - brick.x)
            edge_hit = 0
            tmp = math.fabs(self.ball_x - (brick.x +
                ↪ block_width))
            if tmp<min_distance:
                min_distance = tmp
                edge_hit = 1
            tmp = math.fabs(self.ball_y - brick.y)
            if tmp<min_distance:
                min_distance = tmp
                edge_hit = 2
            tmp = math.fabs(self.ball_y - (brick.y +
                ↪ block_height))
            if tmp<=min_distance:
                edge_hit = 3

            if edge_hit == 0: #left edge
                if self.ball_speed_x > 0:
                    self.ball_speed_x = -self.ball_speed_x
            elif edge_hit == 1: #right edge
                if self.ball_speed_x < 0:
                    self.ball_speed_x = -self.ball_speed_x
            elif edge_hit == 2: #top edge
                if self.ball_speed_y > 0:
                    self.ball_speed_y = -self.ball_speed_y
            else: #bottom edge
                if self.ball_speed_y < 0:
                    self.ball_speed_y = -self.ball_speed_y

            # bug correction end

            self.current_reward += self.STATES['Scores']
            self.paddle_hit_without_brick = 0
            break
```

We quickly solved it and compared the two environments that we obtained. Despite all these efforts, the difference between the results obtained by applying the method on one environment rather than another is still large: the modified PyGame still manages to produce good results, event with just 5000 epochs of training. It has been noted, however, that PyGame uses an internal reduction of the actually used states: when the ball is lower in the screen, the ball and paddle positions are calculated with greater resolution. We tried to use this resolution reduction algorithm in Atari, but the results were once again disappointing. We even found a worsening effect of the score.

Algorithm 9: The resolution reduction algorithm inside the get state function.

```
def getstate(self):
    resx = b.resolutionx  # highest resolution
    resy = b.resolutiony  # highest resolution
    if (self.ball_y < self.win_height // 3):  # upper part,
```

```
        ↪ lower resolution
        resx *= 3
        resy *= 3
    elif (self.ball_y < 2 * self.win_height // 3):  # lower
        ↪ part, medium resolution
        resx *= 2
        resy *= 2

    ball_x = int(self.ball_x) // resx
    ball_y = int(self.ball_y) // resy
    ball_dir = 0
    if self.ball_speed_y > 0:  # down
        ball_dir += 5
    if self.ball_speed_x < -2.5:  # quick-left
        ball_dir += 1
    elif self.ball_speed_x < 0:  # left
        ball_dir += 2
    elif self.ball_speed_x > 2.5:  # quick-right
        ball_dir += 3
    elif self.ball_speed_x > 0:  # right
        ball_dir += 4

    if self.simple_state:
        paddle_x = 0
    else:
        paddle_x = int(self.paddle_x) // resx

    diff_paddle_ball = int((self.ball_x - self.paddle_x +
        ↪ self.win_width) / b.resolutionx)

    return {
        "ball_x":   ball_x,
        "ball_y":   ball_y,
        "ball_dir": ball_dir,
        "paddle_x": paddle_x,
        "diff_paddle_ball": diff_paddle_ball,
        "bricks_matrix": self.bricksgrid
    }
```

After all these futile attempts we have finally managed to identify the problems of Atari, which are summarized in the next point.

6.3 The Atari Breakout Difficulties

In this point we will analyze the intrinsic problems in Atari that prevent the achievement of good results with the method used. One has already been addressed in point 6.1, which concerned the use of an environment with more lives. Here we will describe the problems of the state space, of the local maximum, of non-determinism and of the collision of states.

6.3.1 The Dimension Space Problem

The problem of the size of the state space was clear from the beginning. Before testing it, PyGame had only been used with a 3x3 brick matrix. Atari, on the other hand, has a fixed-size matrix equal to 6x18, which makes both the automaton defining the temporal goals and the size of the playing field larger, in proportion to the paddle size. As explained up to here, there have been many attempts at reducing the state space in Atari, to the extent that it made that dimension identical to that of PyGame, and all attempts proved to be unsuccessful. They did not produce significant improvements in the score, and sometimes even showed a worsening. In fact, it is possible to notice from the number of states visited at each epoch that the speed with which the AI explores new states falls after some threshold, despite maintaining a low score. The algorithm therefore has difficulty in exploring new states, which can have two explanations: there are too many states, and the algorithm get lost, or the problem is not about the number of states, is about the inability of the algorithm to learn, probably due to a collision of states (see below).

6.3.2 The Local Maximum Problem

The artificial intelligence algorithm that must learn to maximize the reward must face a problem of local maximum. In fact, there are many states from which it is difficult to escape. Take the case in which the paddle hits the ball in the lower left corner. If the ball tends to go to the right as a result of the action, there is the possibility that it will fall back into the opposite corner, lower right. If this is the case, the paddle will have to move multiple times, before it can get a new reward. In particular, we need a series of subsequent "go right" actions, which make it possible to continue the game. This is very unlikely, because, in the presence of new states, the choice of action to be taken is random. And there's a second problem: the difficulty introduced by a greater number of possible actions: while on PyGame there are only three actions ("go left", "go right", "stay still"), on Atari there is a fourth possible action, which is needed to spawn the ball at the start of the game, or when a life is lost. This action is automatically performed by the environment on the aforementioned occasions, but it has not been possible to prevent the execution of these actions during the normal course of the game. So the correct action to be taken ("go right") must be chosen randomly by a group of 4 actions (25% probability of choosing the correct action) instead of 3 (33%). And if, say, the number of times to perform the "go right" action is N, the probability of this happening is $25\%^N$, which falls below 1% already for N = 4. If we analyze the moments in

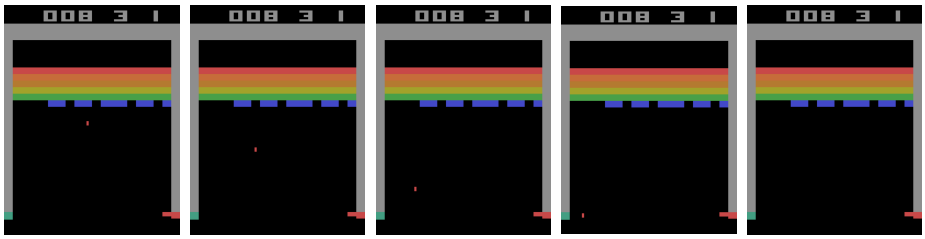


Figure 8: Sequence of sampled frames

which the paddle drops the ball in the training carried out, we note that most of them are just the cases described here, in which the paddle is in a corner of the screen and does not move, sign that no rewards have been discovered for any action, and the ball falls in the opposite corner.

But we have found a trick to alleviate this problem. By default, Atari returns an observation every 4 frames, and for each of those frames Atari performs the same action we told it to execute. In practice a state-frame corresponds to 4 game-frame: at each state-frame the AI chooses an action that will be performed for 4 game-frames, after which the environment will return a new observation, that is a new state-frame. Increasing this number increases the number of times the same action is performed, increasing the paddle movement from one state-frame to another. This allows you to choose the same action for fewer times in order to reach the opposite corner of the screen. That is, it reduces N, thereby increasing the probability of this happening ($25\%^N$). On the other hand, this makes it more difficult to hit the ball. Tests were performed and optimal intermediate values were found for the frame skip, between 10 and 14, which lead to a good increase in the score.

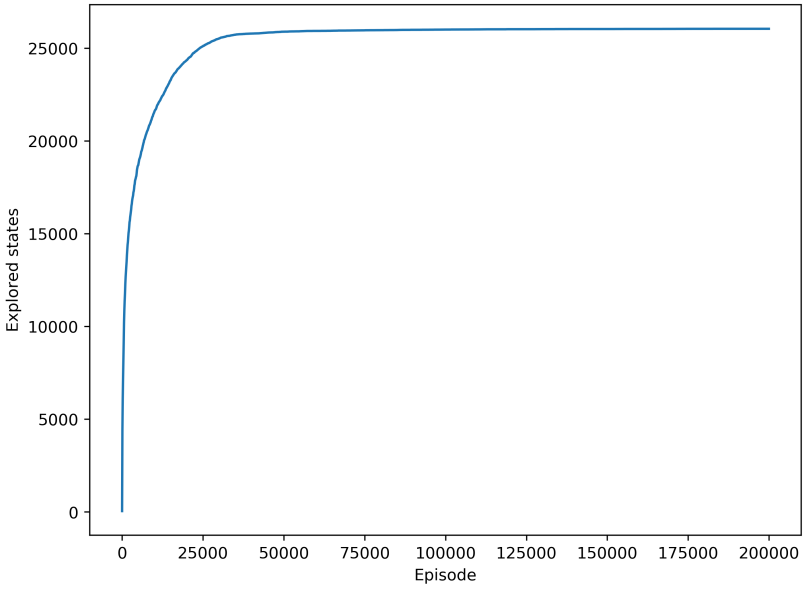
6.3.3 The Non-Determinism Problem

During the frame skip tests we noticed the problem that is the main difference between the two environments: non-determinism. When you perform an action in a state, it is not possible to know for sure what the next state will be like. This is not the case for PyGame. For Atari, this is because the amount of paddle movement is not fixed, but rather varies, partly randomly and partly according to the actions performed in previous states. It is even possible that, given a state and a movement action to be performed, the paddle will move into the next state in the opposite direction to that indicated, due to the inertia that the paddle has at that moment. In particular, without a frame skip, the paddle moves from -6 to +6 pixels. The skip of the frames, however, accentuates this problem, increasing the uncertainty of the movement from -6f to + 6f, with f

the number of skipped frames from one state to another. To solve this problem at least partially, we inserted another variable in the state: move. It indicates the amount of movement performed with the last action, in order to have an estimate of the inertia of the paddle, and therefore know at least approximately where the paddle will go afterwards. The results of these tests are in point 1 of this section.

6.3.4 The State Collision Problem

The most serious problem, however, is the one introduced by non-determinism: the collision of states, that is when two different states are represented in the same way by the AI. The introduction of the variable move serves precisely this, that is, to provide the algorithm of a sort of memory that indicates more or less what the previous state was and, consequently, help to predict the future state based on its own actions. However, this is not a definitive solution. Suppose in fact that move indicates that the paddle has moved an average amount between no movement (0) and maximum movement (6f). We do not know if the paddle was accelerating or braking. We do not know if the previous action was "stand still" and the paddle moved randomly by a lot, or if it was "go right" and the paddle randomly moved by a little bit. Suppose now that the AI has learned that in a certain state, by performing a certain action, it will get a reward. What happens now if, during the training phase, the AI meets again the same state encountered some frames before, but in a different circumstance? If the epsilon policy does not tell us to choose randomly, we will obviously do the same action as before, but the result may not be the same, we could lose the ball. Even worse is the case in which the epsilon policy tells us to perform a random action, different from the optimal one, but leads to a reward. In this case we are going to overwrite the information of the optimal action on that state and moreover, since this reward is more recent, it will have a higher value. So it will happen that, the next restart of the game, when we meet the first occurrence of the double state, the AI will perform the wrong action, losing not only the ball, but also all the rewards we would have obtained following that path of actions leading to the second occurrence of the state, the very same path that we had already explored at the cost of so many clock cycles. Furthermore, the greater the distance between the two occurrences of the conflicting state, the greater the loss of rewards. This explains the fluctuating trend of the score during the training phase, which is notably different from the PyGame case. So this is the main reason for the difficulty of applying the Q-Learning or SARSA algorithms on Atari Breakout.



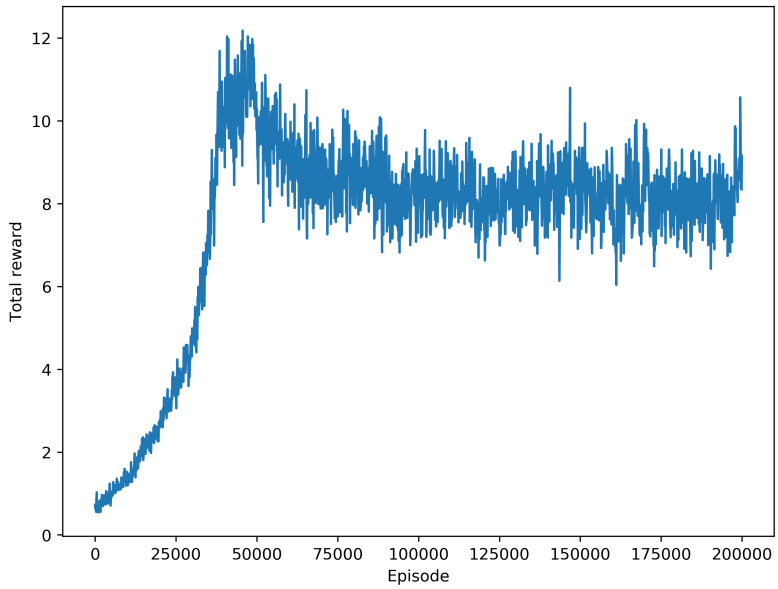


Figure 9: Plots from the Atari Environment experiments. The reinforcement learning algorithm used in this experiment is Q-Learning.

7 Conclusion

The project presented an extension of the work introduced in [**?**], making experiments on an Atari Breakout version available in the `gym` framework. All the theory behind the work has been briefly introduced and all the main parts of the code have been discussed in detail, in order to simplify the analysis of the two environments used in the project. The best performance did not manage to correctly solve the Breakout game, suggesting that more work needs to be done on the extractors, in order to make them more reliable and fast, and on the learning algorithm. In fact, all the successful recent work managed to solve difficult Atari games using deep neural networks, which are able to generalize on the observation making the training much faster. Moreover, the training phase uses GPUs which make it possible to parallelize the computation, making the training even faster. More recent work use asynchronous methods [**?**] which manage to obtain great results without using the GPU. Working in this direction could help the agent in working for more episode in less time, discovering much better strategies and managing to interact with the environment in the best possible way, allowing the agent to work also with non-Markovian rewards.