Concurrent Engineering Project

AI Gamer

Barbier De La Serre Nicolas
Nicoletti Francesca Paola
Raita Omar
Tchatat Njieyep Victoire

Professor: Giancarlo Ferrari Trecate

Assistants :
Fabbiani Emanuele
Galimberti Clara
Guo Baiwei
Turan Mustafa
Xu Liang

June 5, 2020

Contents

1 Introduction

1.1 Context and aim of the project

In a world that is turning more and more to Artificial Intelligence (AI) for problem solving, mechanical engineers need the tools to also tackle this particular field. Following that vision and adding some fun to it, our concurrent engineering project aimed at implementing reinforcement learning (RL) techniques such as Deep Learning (Pytorch) and Deep Q-Learning to Atari Games .

1.2 Reinforcement learning and Theory and fundamentals

Reinforcement learning involves a learning agent interacting with its environment in order to carry out actions that will lead it to yielding the most reward.

First, a reward is number than can be negative or positive. It is obtained periodically from the environment. The reward is thus just a scalar that is used to tell the agent how well it have behaved, and to indicate its recent performance. The notion of reinforcement is derived from the fact that the reward received by the agent is supposed to reinforce or enhance its behaviour in a positive or negative way. Generally , the agent receives a reward for every interaction with the environment, but it could also be every minute or once in its lifetime. Anyways, the most important thing to remember is that a reward is a feedback given to the agent regarding its recent achievements.

But what is really an agent? The agent is something/someone who constantly interacts with the environment by carrying out actions and observations and eventually perceiving rewards accordingly. In reality, the agent is the part of our software whose goal is to solve the problem addressed in our game as efficiently as possible.(1)

The environment finally represents everything outside of the agent. The latter interacts with the environment in three different ways. The agent receives rewards from the environment, performs actions in the environment and is able to observe from the information provided by the environment.Nevertheless, The agent or the learner is not told which actions to take but instead figures it out by trying itself.

Reinforcement learning can easily show itself useful in real life situations such as: neuroscience to model decision making process, traffic light control, robotics, chemistry and game improvement to name a few. Therefore , some specific algorithms are implemented and will be unfolded throughout the report.

Neural Networks can be seen as a more complex approach to solving these kinds of learning problems. They are actually a biologically-inspired programming paradigm which enables a computer to learn from observational data. In fact a neural network is structured in layers that resemble the network structure of the brain. It can learn from data and be trained to recognise patterns, classify data, and forecast future events.

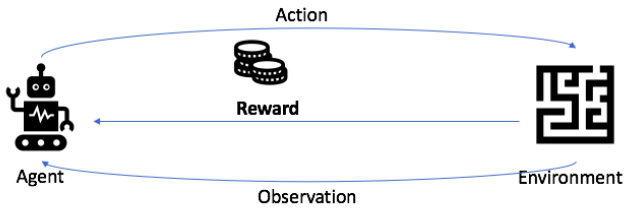

Figure 1: RL entities and interactions (1).

1.3 OpenAI environments

In the course of our project, we use the toolkit named OpenAI Gym for developing our reinforcement learning algorithms. It is tailored for reinforcement learning. Indeed, it provides an environment, an `action_space` and an `observation_space`. Its API allows interfacing with different games. For example the extension `gym[atari]` emulates Atari 2600 games. Other environments are more focused on control theory. This work essentially focused on some of them according to the purpose.

- First, to get started , the Classic control environments, which mainly complete small-scale tasks in the RL literature, have been tested. The games " CartPole-v1" and "MountainCar-v0" were especially investigated.
- Then, when the algorithm were efficient on these games, they were adapted to the Atari environment. The final results were obtained with "BreakoutDeterministic-v4", "PongNoFrameSkip-v4" "MsPacmanNoFrameSkip-v4".

Let us take you through the process...

2 Reinforcement Learning algorithms implemented

2.1 Markov property and Bellman equation

The learning process started with games such as "Maze" ,"GridWorld" and "MountainCar-v0". The objective here was to train the agent into taking actions yielding the most reward.

All tasks followed the Markov decision process (MDP) , meaning they satisfy the Markov property. A state signal that has the Markov property is defined as a state signal that summarizes past sensations compactly, then, in which all relevant information is retained. The main point of the Markov property is to make every observable state self-contained to describe the future of the system. Thanks to this property, a one-step dynamics enables to predict the next state and the expected next reward only given the current state (s) and action (a). Then, is possible to define $P\left(s'|s\right)$ the probability of transition to state s' from state s.

The agent tries to get the most expected sum of rewards from every state it lands in. In order to achieve that the optimal value function has to be found. Bellman equation will help to do so. In Markov decision processes, a Bellman equation is a recursion for expected rewards. The expected reward for being in a particular state s and following some fixed policy π (some set of rules that controls the agent's behavior) has the *Bellman equation* :

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P\left(s'|s, \pi(s)\right) V^{\pi}\left(s'\right) \tag{1}$$

This equation describes the expected reward for taking the action prescribed by some policy π .

By iterating this equation, the objective is to find the optimal policy $\pi*$ that respects the *Bellman optimality equation* :

$$V^{\pi*}(s) = \max_{a} \left\{ R(s,a) + \gamma \sum_{s'} P\left(s'|s,a\right) V^{\pi*}\left(s'\right) \right\} \tag{2}$$

This equation describes the reward for taking the action giving the highest expected return.

2.2 Q-learning algorithm

The first algorithm implemented is the Q-Learning algorithm. It uses the optimal action-value function q_*(s, a) = max_{π} q_{π}((s, (a)).Where q_{π} represents the expected return starting from state s, taking the action a, and finally following the policy π. A policy being a mapping from each state, $s \epsilon$ S, and action, $a \epsilon$ A(s), to the probability π(a|s) of taking action a when in state s. Following the *Bellman equation*, the algorithm predicts the most rewarding value q_*(s', a') that would be obtained if we knew all the possible values of *a'* (next action) by selecting the action that maximises the value of $r + \gamma$ *Q* (s',a')* (where r is the reward and γ is the discount factor) (2)

$$Q^{*}(s,a) = [r + \gamma \times maxQ^{*}(s', a')|a](maxovera') \tag{3}$$

This process is iterative (Diagram: 2) and, as the agent trains and explores the environment, it gradually updates the q-values first initialised to zero in the Q-tables. The q-values are updated at each time t considering the learning rate α and following this specific affectation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \times [r_{t+1} + \gamma \times \max_{a} Q(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{4}$$

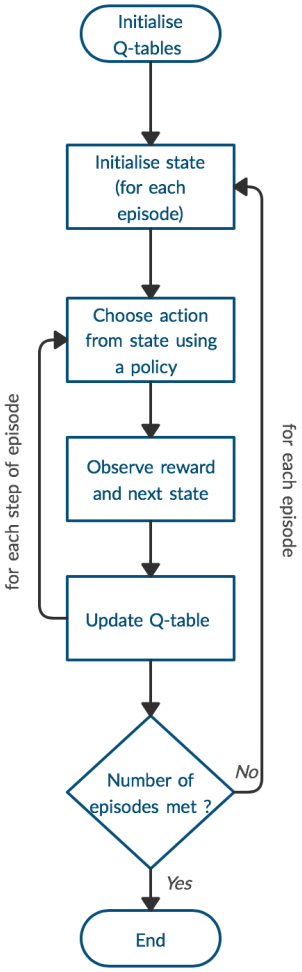

Figure 2: Q-learning algorithm (2).

Following a certain strategy, an agent uses values already updated that will lead him to earning the maximum reward. Note that the agent is not told how to take action. Therefore ,it has to learn through trial and error. It can either explore by improving its knowledge about each action or exploit its current action-value estimates. An example of strategy epsilon-greedy policy, where epsilon refers to the probability of choosing to explore.It is a simple method used to balance exploration and exploitation by randomly choosing between one of them.An implementation will be presented further.

Note that:
- S represents the set of possible states.
- A(s) represents the set of actions available in a particular state.

2.3 Policy iteration algorithm

The next section will present the implementation of all the algorithms on concrete games. Yet , after working with Q-learning we wanted to implement a new algorithm to see if we could improve our results. Therefore we went for the Policy iteration algorithm. The objective of this process is to determine the optimal policy for the model. It consists of three steps:

- The first one is the *Policy Evaluation* that estimates the value of the long-term reward function with the policy π obtained from the last Policy Improvement or, for the first iteration, the result of our Q-learning or a random policy π.
- The second one is the *Policy Improvement* that updates the policy with the action that maximizes the value function q_π (the expected return, meaning the value of state s under policy π) for each of the states.
- The final one is the policy iteration that repeats each of the previous steps till convergence is reached.

Policy iteration is thus based on exploration and can be considered to use model free dynamics.

All steps of policy iteration algorithm are illustrated in figure 3.

Policy Evaluation is based on two equations :

$V(s) = \sum_{s',r} p\left(s', r|s, \pi(s)\right)\left[r + \gamma V\left(s'\right)\right]$
$\Delta = \max(\Delta, |V'(s) - V(s)|)$

Policy Improvement is based on one main computation :

$\pi(s) = \arg\max_a \sum_{s',r} p\left(s', r|s, \pi(s)\right)\left[r + \gamma V\left(s'\right)\right]$

The variables of these equations are :

- V the value function
- π the policy
- s the state
- s' the next action
- γ the discount factor
- $p\left(s', r|s, \pi(s)\right)$ the probability of transition to state s' with reward r from state s and policy $\pi(s)$

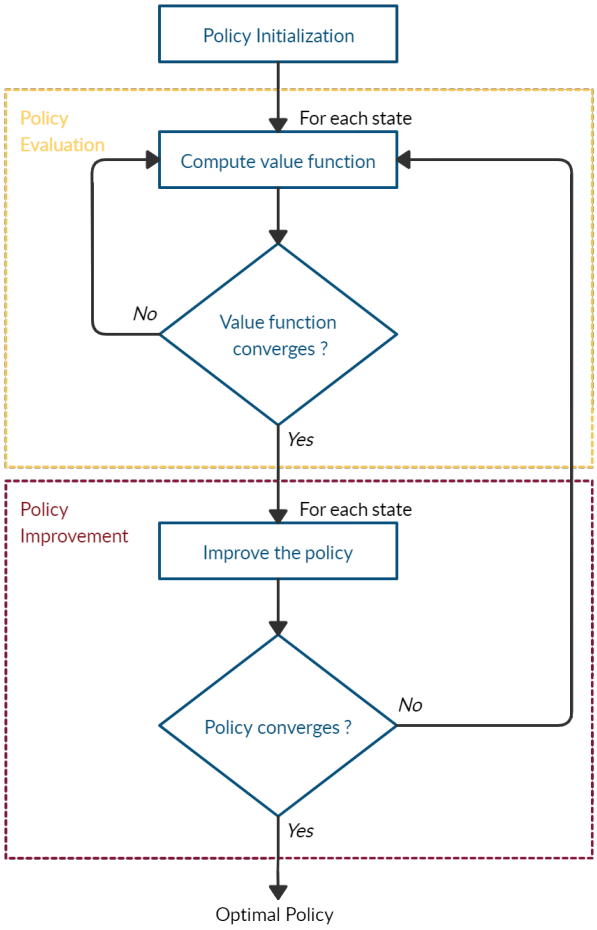

Figure 3: Policy iteration algorithm

2.4 SARSA algorithm

The last algorithm we implemented is the SARSA (State Action Reward State Action) Algorithm. It is a variation of the Q-learning algorithm. The difference is that it uses an on-policy method: the agent learns the value function according to the current action derived from the policy currently being used not from another policy. This is opposed to an off-policy method where the learning agent learns the value function according to the action derived from another policy.

All the steps of SARSA algorithm are illustrated in figure 4 where the Q-function is updated using the following equation :

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma Q\left(s',a'\right) - Q(s,a)\right] \tag{5}$$

with Q the Q-function, s the state, s' the new state, a the action, a' the new action, r the reward, α the learning rate, γ the discount factor.

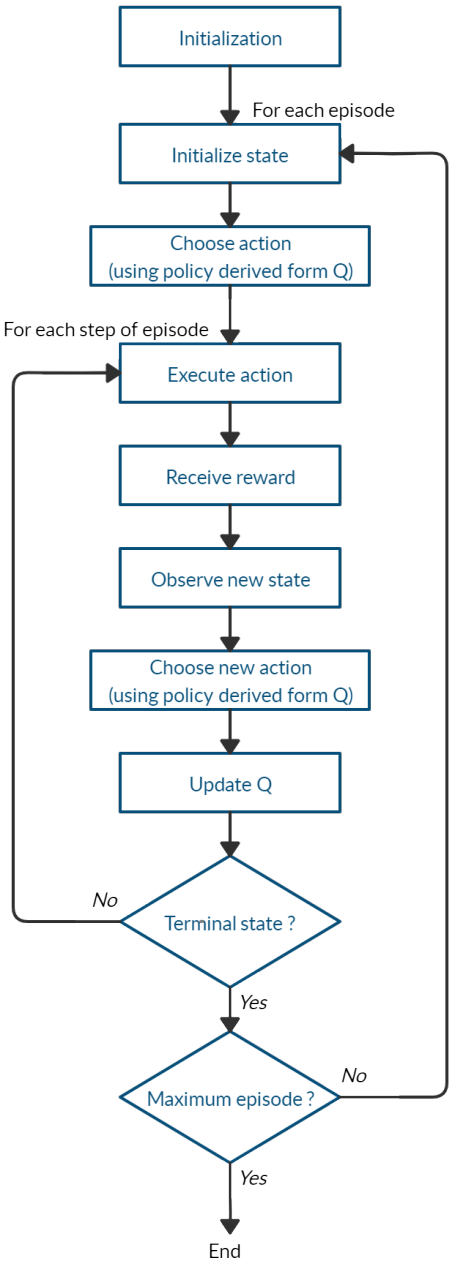

Figure 4: Policy iteration algorithm

3 Reinforcement Learning on games

3.1 First insights into reinforcement learning with discrete problems: Maze

3.1.1 Problem

The maze game from Gym is a 2D environment made of cells, where the agent has to find its way from the top left corner to the bottom right corner where the goal is. The objective is to find the shortest path to this goal. The agent is able to teleport if he steps into a colored cell. The teleportation can only be done between two cells with the same color. The observation space is the (x,y) coordinate of the agent. The latter can only choose to go right, left , up and down. A reward of 1 is earned when the agent reaches the goal. However, for any step in the maze, the agent receives a reward of -0.1/(number of cells). Finally , the end condition is met when the agent reaches the goal.

3.1.2 Q-learning for Maze

This first game served as a study case in the project. The learning process began with an example of reinforcement learning through an existing code from GitHub. First a q-table initialised to zero is created based on the number of discrete states and action. The q-table is then updated using an iteration of the Bellman equation:

```
# Update the Q based on the result
best_q = np.amax(q_table[state])
q_table[state_0 + (action,)] += learning_rate * (reward + discount_factor * (best_q) - q_table[
    state_0 + (action,)])
print(q_table)
# Setting up for the next iteration
state_0 = state
```

Listing 1: Updating (Q-tables- Maze code from GitHub)

In order to gain more intuition about the functioning of the q-learning algorithm, a plot has been made to better visualize the optimal q-values for each state-action pair (Fig.6). The green cells are the ones corresponding to the optimal q-value for a given state. It is possible to think backwards and find the optimal path found by the agent. For instance, the only table with a green cell in the position (2,2) is the table "Action South". This means that it is the optimal action for that state. When looking at the position (2,2) in the maze, one can see that the agent will choose to step on the colored cell so that he can be teleported to another cell closer to the final goal and thus perform less steps. The optimal path has been reconstructed from these tables (Fig.5).

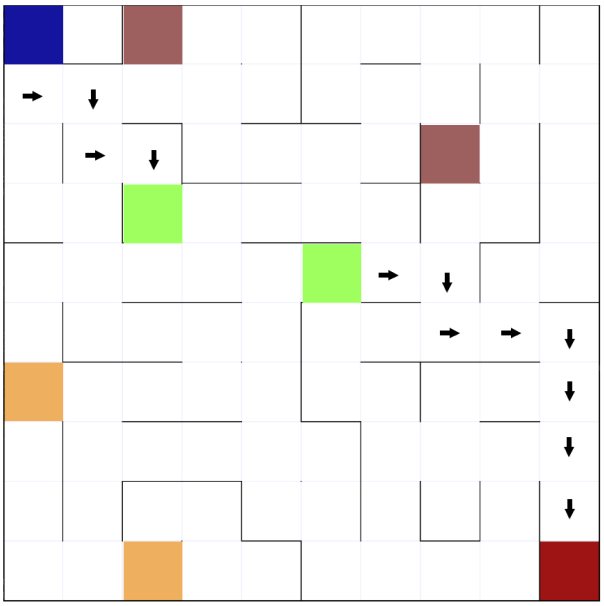

Figure 5: Example of optimal path

Figure 6: Optimal q-values for each action

3.2 First insights into reinforcement learning with discrete problems : Grid world

Grid world is a simple problem that helps us develop intuitions in reinforcement learning. One can better understand the meaning of Q-values by observing its Q-tables. It is also interesting to visualise the propagation of the value function among the discrete space after each iteration. A first implementation of Q-learning is already provided with the environment(3).
The improvement that has been made was the implementation of the **Policy iteration** algorithm. This algorithm is very interesting for discrete environments such as grid world. In fact, since it uses iterations over all the states, the algorithm might be well adapted for such problems.
The environment used is a 3 by 4 simple grid. Let x be the coordinate corresponding to rows and y be the one that corresponds to columns. For a starting position at (0, 0), a win position at (2, 3) and a lose position at (1, 1), the reward associated with each step is -0.01. If the agent falls into the sink it receives a -10 reward. If the agent achieves the final position the environment returns +10 as a reward (Fig.7).

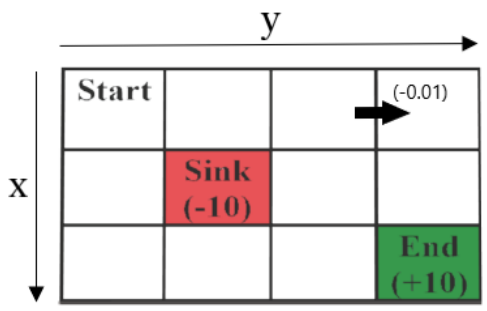

Figure 7: Illustration of the grid world problem

For an initialisation with a random policy, the whole algorithm performs only 3 iterations before it finds the optimal policy. This is coherent with the grid world example's results provided in the literature (2).

down	left	down	up
up	down	up	left
up	up	right	down

Table 1: Example of initialisation with random policy

right	right	down	down
down	down	down	down
right	right	right	down

Table 2: Optimal policy obtained from the policy iteration algorithm

5.495	6.217	7.019	7.91
6.217	-10.0	7.91	8.9
7.019	7.91	8.9	10.0

Table 3: Final value function evaluated at each state

It is important to note that, for the same hyper-parameters, the same value table is obtained for different runs with a random policy.

See the implementation of the algorithm below.

```
def policy_evaluation(self,theta,V,V_old):
    while True:
         delta=0
         for x in range(0, BOARD_ROWS):
             for y in range(0, BOARD_COLS):
                V_old[(x,y)]=V[(x,y)] # store the old V-table, an iteration is necessary to not
    copy the same object
         for x in range(0, BOARD_ROWS):
             for y in range(0, BOARD_COLS):
                 self.State.state = (x, y)
                 nxtState = self.State.nxtPosition(policy[(x, y)])
                 if (x,y)== WIN_STATE or (x,y)== LOSE_STATE:
                     continue # no need to update values of Win and Lose states
                 nxtreward = self.State.giveNxtReward(nxtState)
                 reward=self.State.giveReward()
                 v = V[(x, y)]
                 V[(x, y)] = reward + discount_factor * V_old[nxtState] # update values
                 delta = max(delta, abs(v - V[(x, y)])) # variation of values

         if delta<theta: #Theta is the convergence threshold, arbitrarily set to 0.01
             return V
             break
```

Listing 2: Policy evaluation

```
def policy_improvement(self,V):
    policy_stable = True
    for x in range(0, BOARD_ROWS):
        for y in range(0, BOARD_COLS):
            self.State.state = (x, y)
            a = policy[(x, y)]
            v = V[self.State.nxtPosition(policy[(x, y)])]
            a_star = a
            v_star = v
            for action in ['u', 'd', 'l', 'r']:
                v_prime = self.State.giveReward() + discount_factor * [self.State.nxtPosition(
    action)]
                if v_prime > v_star:
                    a_star = action # best action among all possible actions, for each state
                    v_star = v_prime
                    policy_stable = False
                policy[(x, y)] = a_star # update the policy
    return policy_stable
```

Listing 3: Policy improvement

3.3 Mountain car problem

3.3.1 Problem

Here is the original description of the game , presented by **Andrew Moore** in his PhD thesis:

"A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Here, the reward is greater if you spend less energy to reach the goal."

Some key information on the environment included the possible parameters which are :

- position: varies between -1.2 (min) and 0.6 (max)
- velocity : varies between -0.07 (min) and 0.07 (max)

More so, there are three predefined actions: push left, no push and push right. The associated reward is -1 for each time step until the goal position of 0.5 is reached. In the original code, the episode ends when you reach 0.5 position or if 200 iteration are performed.

3.3.2 Q-learning for Mountain car and polynomial approximation of Q-functions

The goal here is to train our car to reach the mountain top in as less episodes as possible.The Q-learning algorithm is also implemented with an epsilon-greedy strategy. Remember that epsilon refers to the probability of choosing to explore. Nevertheless, the strategy exploits most of the time with a smaller chance of exploring.In fact, for a random probability p the ϵ-greedy policy will choose the best current action for $p > \epsilon$ and a random action in the other case.

```
while not done:
  if np.random.random() > epsilon:
    action = np.argmax(q_table[discrete_state]) # action depuis Q table
  else:
    action = np.random.randint(0, env.action_space.n) #random
```

Listing 4: Epsilon-greedy for Mountain Car

For every iteration, the q-values update a q-table which takes the form of a 3D matrix. The q-values can be plotted (Figure : 8) as a function of the two parameters of the problem which are state and velocity, for each action of the agent (0=left, 1=nothing, 2=right).

```
new_state, reward, done, _ = env.step(action)
new_discrete_state = get_discrete_state(new_state)
if not done:
  max_future_q = np.max(q_table[new_discrete_state]) #max Q value in step
  current_q = q_table[discrete_state + (action, )] #current Q value
    new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE * (reward + DISCOUNT * max_future_q)
  q_table[discrete_state+(action, )] = new_q #updatee Q table
```

Listing 5: Q-table update for Mountain Car

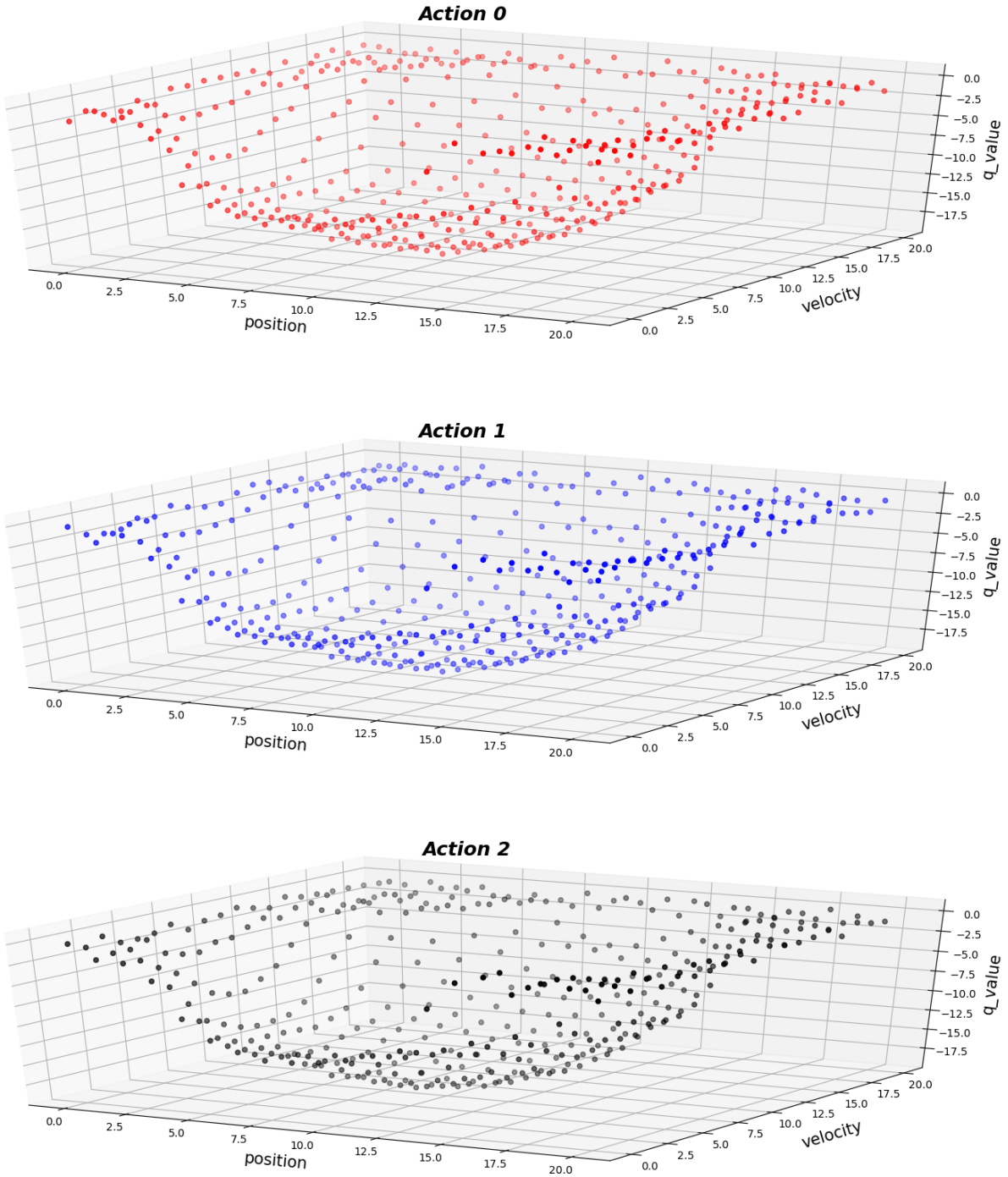

Figure 8: Q-values computed after 980 episodes for mountain car

For every action, the array of q-values can also be approximated by a second-order polynomial function. It can be noticed from Figure 9 that the state (7,:) (for (position,velocity)) corresponds to the starting position and the state (18,:) to the winning one. At first the q-values are random between 0 and -2 and when the game is won the max is 0 as predicted.
The objective is now to take the polynomial approximation of the first winning episode and implement it in our algorithm. Rather than choosing an action based on the best q-value for a given state, the agent is now choosing the action associated with the max value of the polynomial function evaluated at the position-velocity tuple. We observe that in this configuration the agent wins a lot of times compared to the first episode. Also, when it does not win, the agent is often very close to doing so. This might be caused by the approximations of the interpolated function.

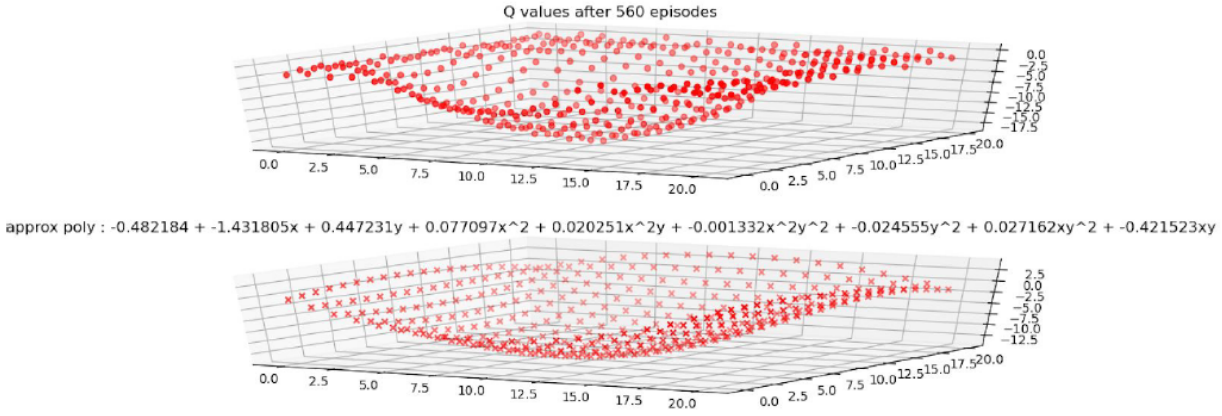

Figure 9: Q-values and polynomial approximation of the Q-function for Mountain car

3.3.3 Policy iteration for Mountain Car

The policy iteration algorithm has shown good performance with discrete problems (section 3.2). One way to further explore its potential is to try it on continuous problems such us Mountain Car. However, the implementation of this algorithm is not straightforward. As it is described in the pseudo-code 76, the agent must be able to access all the states in order to perform a policy evaluation and improvement. Unlike the grid world problem, the agent can not access an adjacent state by performing an action from the action space. For example, if the car is at the position (10,10) on the state space, taking the action "right" doesn't necessary mean that the next state of the car will be (10, 11) or (11, 10). It might lead to a completely random state, depending on the car's position on the hill. In this context, it is important to highlight the major difference between policy iteration and SARSA algorithms. Unlike SARSA, policy iteration algorithm is about accessing the states iteratively. Which mean that the number of states that are visited is fixed. However, for SARSA, the agent keeps performing steps until it reaches the terminal states.
Thus, the problem described in the previous paragraph can not be dodged by performing actions that lead to a random state until the car reaches the terminal state, otherwise it won't be a policy iteration anymore but rather a SARSA implementation.
For this reason, a function was taken from the source code of the mountain car environment and reused to simulate the game dynamics without explicitly playing with the car. This function is the one that computes the next state, given a state action pair.

```python
def nxtState(state, action): # Here, the input state is discrete e.g (10,10)
    position = state[0] * (max_position - min_position) / discretisation - 1.2
    velocity = state[1] * (max_speed + max_speed) / discretisation - 0.07 #  state[0] and state[1] are discrete integers going from 0 to 19.
    velocity += (action - 1) * force + math.cos(3 * position) * (-gravity)
    position += velocity
    position = np.clip(position, min_position, max_position)
    velocity = np.clip(velocity, -max_speed, max_speed)
    if (position == min_position and velocity < 0): velocity = 0
    done = bool(position >= goal_position and velocity >= goal_velocity)
    reward = -1.0
    position, velocity = get_discrete_state((position, velocity))

    return position, velocity, reward, done, {}
```

Listing 6: Function used to compute the next state from a discrete state

Using the function defined above and following the pseudo-code 76, the implementation of policy evaluation becomes straightforward.

```python
def policy_evaluation(theta, V):
    while True:
        delta = 0
        for x in range(0, discretisation):
            for y in range(0, discretisation):
                V_old[(x, y)] = V[(x, y)]  # store the old V-table, an iteration is necessary to avoid copying the same object
        for x in range(0, discretisation):
            for y in range(0, discretisation):
                state = (x, y)
                position, velocity, _, done, _ = nxtState(state, policy[state])
                nextstate = (position, velocity)
                if state == WIN_STATE:
                    continue # not apdating values for Win state
                reward = giveReward(state)
                v = V[state]
                V[state] = reward + DISCOUNT * V[nextstate]
                delta = max(delta, abs(v - V[state])) # variation of values
        if delta < theta: #Theta is the convergence threshold, arbitrarily set to 0.001
            return V
            break
```

Listing 7: Policy evaluation implementation for mountain car

- Implementation of policy improvement:

```python
def policy_improvement(V, StateAccessible):
    policy_stable = True
    for x in range(0, discretisation):
        for y in range(0, discretisation):
            state = (x, y)
            position, velocity, _, done, _ = nxtState(state, policy[state])
            nextstate = (position, velocity)
            a = policy[state]
            v = V[nextstate]
            a_star = a
            v_star = v
            for action in [0, 1, 2]:
```

```
                positionn, velocityy, _, _, _ = nxtState(state, action)
                exploreState = (positionn, velocityy)
                StateAccessible[(exploreState[0], exploreState[1])] = 1
                v_prime = giveReward(state) + DISCOUNT * V[exploreState]
                if v_prime > v_star:
                    a_star = action # best action among all possible actions
                    if policy[(x, y)] != a_star:
                        policy_stable = False
                    v_star = v_prime
                policy[(x, y)] = a_star # update the policy
    return policy_stable
```

Listing 8: Policy improvement implementation for mountain car

-Results:
The algorithm converges after more or less 12 iterations, with 25 maximum iterations in the policy evaluation. Note these two numbers change, from an iteration to another, due to the random initialisation of the policy. The number of iterations is greater than the one seen in the grid world is due to the fact that, this time, the algorithm deals with a 20x20 state space.

In order to see whether the policy that was found after 12 iterations solves the mountain car problem or not. The game was rendered with an agent that uses the same stable policy.
The following observations were made:

- The car couldn't reach the winning flag but its movements seem to be coherent. In fact, it performs two back and forth moves, allowing it to reach the extreme left but not the flag at the extreme right.
- The exact same policy and value table are found for each run, even if the initial policy is random.
- The whole value table is updated.

It is important to note that this is not an optimal policy for the real problem. This might not be due to an implementation error since the same code is working well for the grid world problem. Furthermore, the algorithm converges to a certain policy that makes the car perform a smooth movement allowing it to reach the top left of the hill.

The problem might be due to the discretization. It is true that the same discretization (20x20) is used for other algorithms but the main difference here is that, for policy iteration algorithm, the discretization is performed on the states that are used to compute the next states with respect to a certain action. In fact, the first two lines of the `nxtState()` function claim that the car only moves in a discrete way. Although the real position and velocity are calculated using the dynamics of the environment, these two remain discrete.

The following diagram gives a comparison between the discretization used for policy iteration and other algorithms:

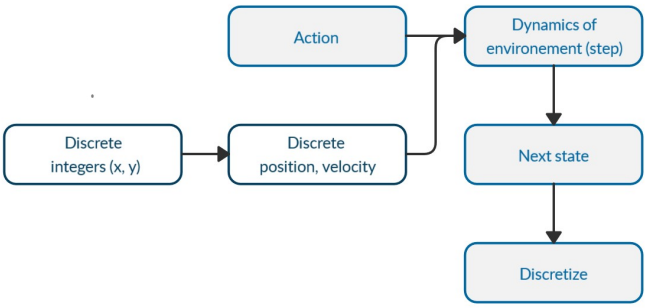

Figure 10: Discretization used in policy iteration algorithm

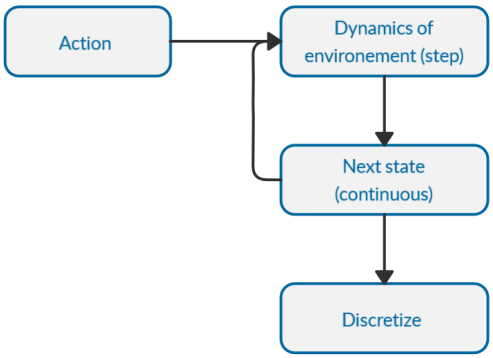

Figure 11: Discretization in other algorithms

Since the car's movement is modelled in a discrete way, it is possible that not all points in the state space are reachable from a discrete space state. Thus, the value function might not propagate properly. The following illustration gives a quick intuition of the problem.

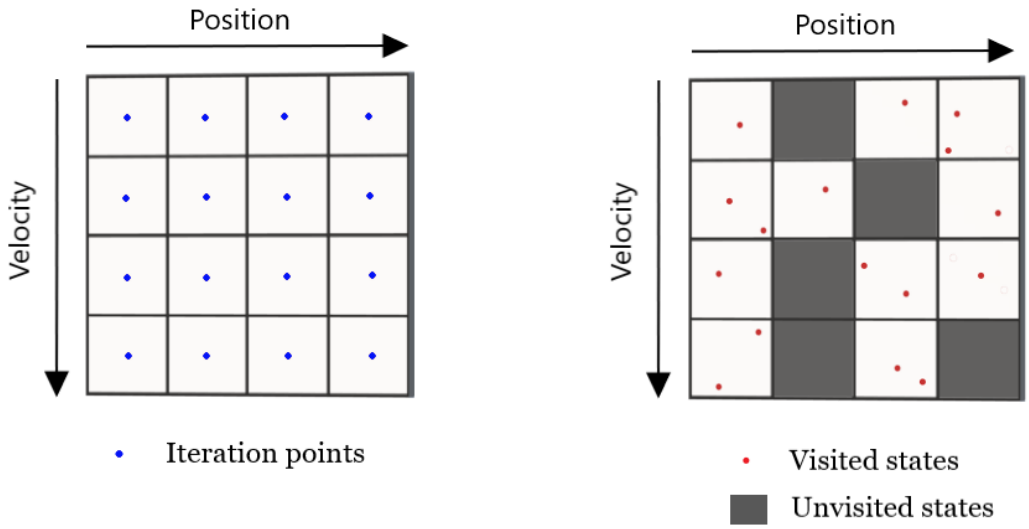

Figure 12: Illustration of the potential problem with the policy iteration applied to mountain car

Another reason why the policy iteration algorithm is not adapted for this type of problems is that it uses a huge assumption which is that the agent is able to learn the value function from all the states in an iterative way, which means that the agent already has information on the environment.

3.3.4 SARSA for mountain car

The results obtained from policy iteration algorithm were not very convincing. In fact, the algorithm has some limitations such the initial discretization of the state space and the known environment assumption. For this reason, the next step is trying to find an on-policy method that is able to solve the mountain car problem. This was the main motivation behind the implementation of Sarsa. This algorithm doesn't require any previous knowledge about the environment.
A first implementation was done using the same hyper-parameters that were used for the Q-learning.

```
episodes = 30000
alpha = 0.1
gamma = 0.99
a = 1.0 # a is the starting value of epsilon
eps = a
```

Listing 9: Training loop

One of the convergence conditions is that the policy must converge in the limit to the greedy policy (2). Therefore, the first function that was used to define the epsilon decay rate is the following:

```
if eps > 0:
    eps -= a/episodes
else:
    eps = 0
```

Listing 10: Training loop

The first results that were obtained can be seen in the figure 13

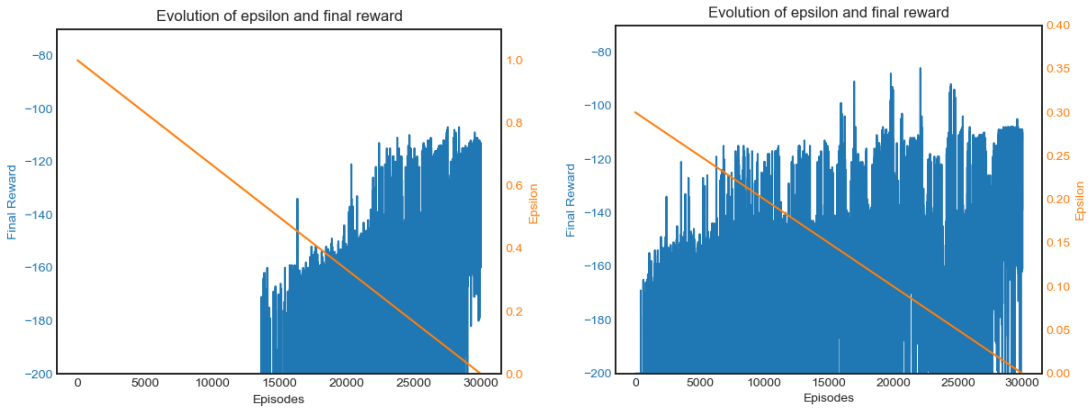

Figure 13: Total reward over each episode for epsilon starting at 1.0 and 0.3

One can notice the reward evolution in both graphs but there is no proof of convergence.

When looking at the implementation used in the windy Gridworld example of Sutton's book (2), the initial value of epsilon was chosen to be 0.1.
Therefore, another try was done with an epsilon starting from this new value. For the same number of episodes improvements can be seen in Fig.14.

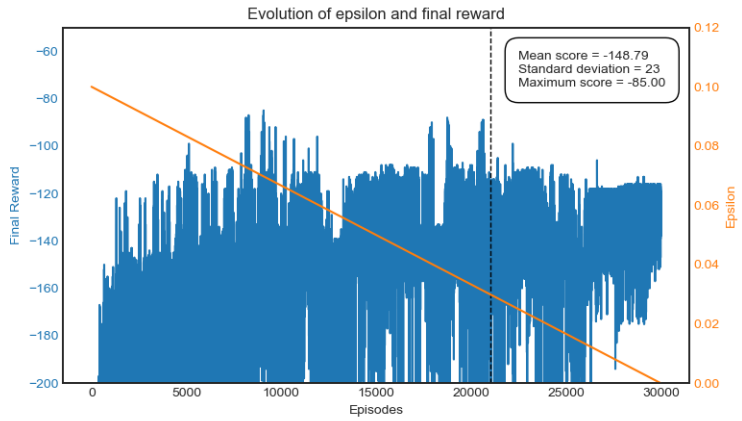

Figure 14: Total reward over each episode for epsilon starting at 0.1

- **Example of results**
episode 7100 score -148.0 eps 0.07633666666670869
episode 7200 score -151.0 eps 0.07600333333337594
episode 7300 score -147.0 eps 0.0756700000000432
Won 101 times in a row at episode 7327 with position= 0.5165260120808546 and velocity= 0.037971648756194874

The algorithm performs a significant number of wins but it is difficult to tell whether it really converges or not to an optimal policy. The hypothesis is that the algorithm doesn't have enough settling time since the epsilon achieves 0 until the final episode. One way to deal with this is to impose an intersection of the epsilon function with x-axis before the final episode.
However, if this intersection takes place in early episodes, the algorithm will not explore enough paths to find better policies. Therefore, one should take into account this trade-off while defining the epsilon decay function.

In order to maximise the exploration in the first episodes and perform enough exploitation to be able to see the convergence. A new epsilon decay function was defined. It is a second order polynomial that has two parameters. The parameter a defines the starting value of the function. While the parameter b defines the intersection with the x-axis. Few runs were made to select the parameter b that shows the best convergence.

```
alpha = 0.1
gamma = 0.99
a = 0.1          # to define epsilon function's offset
b = 7/10         # epsilon function crosses x-axis at b*episodes
eps = a
discretisation = 20
```

Listing 11: Final hyper-parameters

```
if eps > 0:
  eps = -a*(1/((b*episodes)*(b*episodes)))*((i*i)-((b*episodes)*(b*episodes)))
else:
  eps = 0
```

Listing 12: Final epsilon decay function definition

Figure 15 shows the evolution of the total reward for $a = 0.1$ and $b = 7/10$.

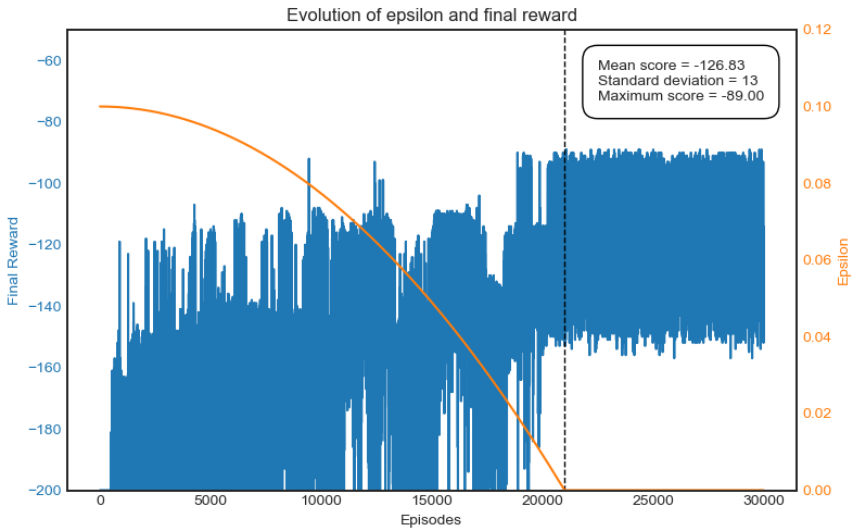

Figure 15: Total reward for 30000 episodes

The mean score has significantly improved from -149.79 to -126.83, the standard deviation has decreased. However, the maximum score also decresed from -85 to -89. This might be due to the high decrease rate of the 2nd order epsilon function around the episode 15000.

-Implementation:

```
while not done:
    observation_, reward, done, info = env.step(action)
    state_ = get_discrete_state(observation_)
    action_ = max_action(Q, state_) if np.random.random() > eps else env.action_space.sample()
    score += reward
    Q[state, action] = Q[state, action] + alpha * (reward + gamma * Q[state_, action_] - Q[state,
    action]) # Updating the Q-values
    state = state_
    action = action_
    steps += 1
    if observation_[0] >= 0.5 and observation_[1] >= 0:  # If state is terminal
        Q[state, action] = 0
```

Listing 13: SARSA implementation

Epsilon values comparison:

To have a better look at the algorithm's improvement and to further investigate the impact of hyper-parameters on the results, some modifications were to be added: Setting maximum steps to 1000 instead of 200 to better visualize the evolution of the total reward with respect to different values of epsilon. This choice has been made to be able to see the evolution for the same number of episodes even if epsilon is starting at high values. Note that the evolution couldn't be seen for high values of epsilon with 200 maximum steps (Fig.13).
For each episode, the algorithm stops once the agent achieves the win state or do 1000 steps. This means that every time the agent gets a reward greater than -1000 means that the car achieved the final position. Thus, the main challenge is to find the optimal path. Figures below show the results for epsilon starting at 0.1 and 0.5.

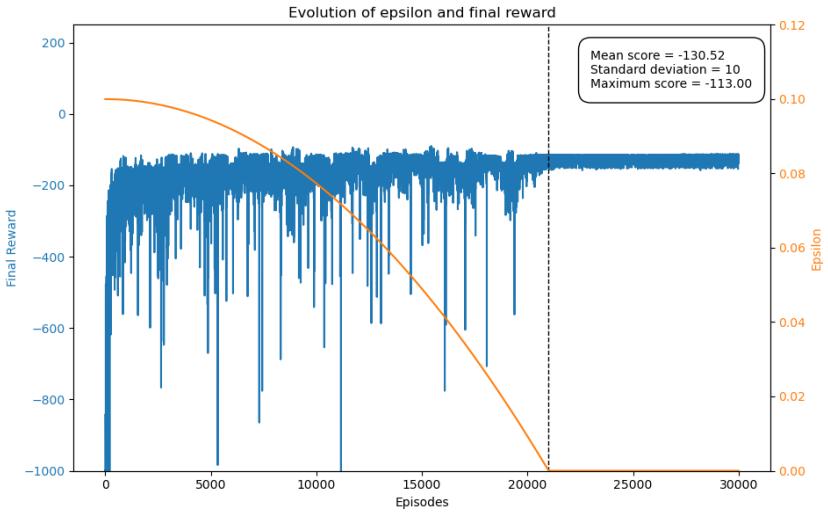

Figure 16: Performance of SARSA algorithm applied to the mountain car problem with epsilon starting at 0.1

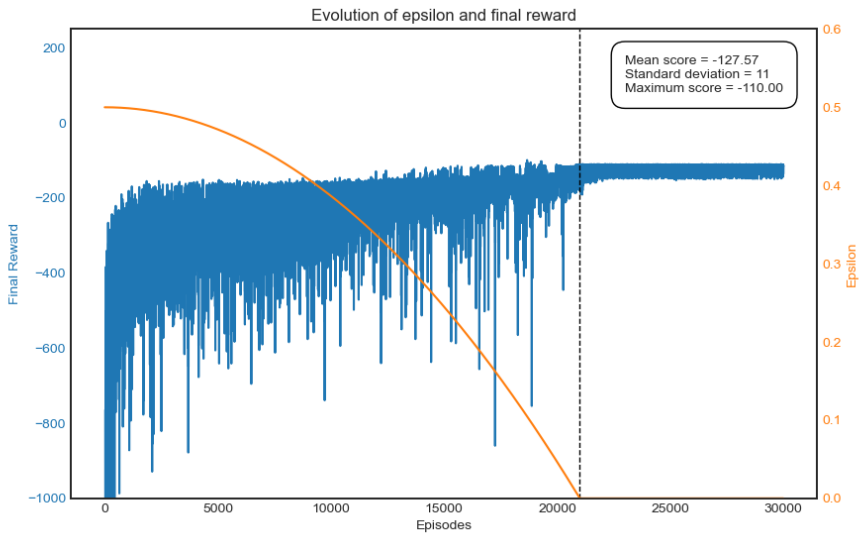

Figure 17: Performance of SARSA algorithm applied to the mountain car problem with epsilon starting at 0.5

The algorithm seem to find a better path when epsilon starts at 0.5, the maximum score that is achieved is greater than the one achieved with a starting value of epsilon at 0.1. The mean score is higher as well. This is coherent since the algorithm does more exploration when the epsilon is higher.

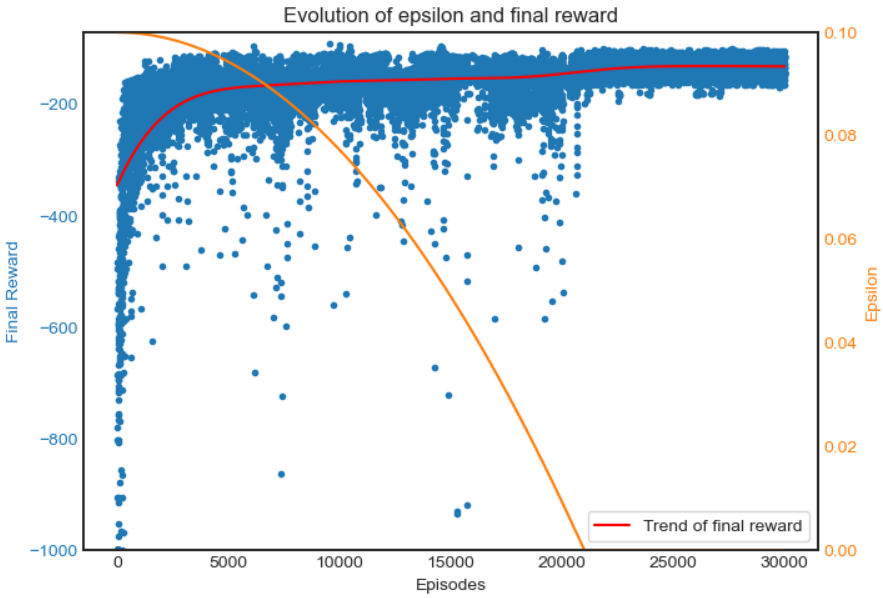

Figure 18: Trend visualization using LOESS (4)

In order to make a more rigorous comparison, one might need to plot trends of final rewards for different starting values of epsilon. This can be done using a locally weighted linear regression (Fig.18).

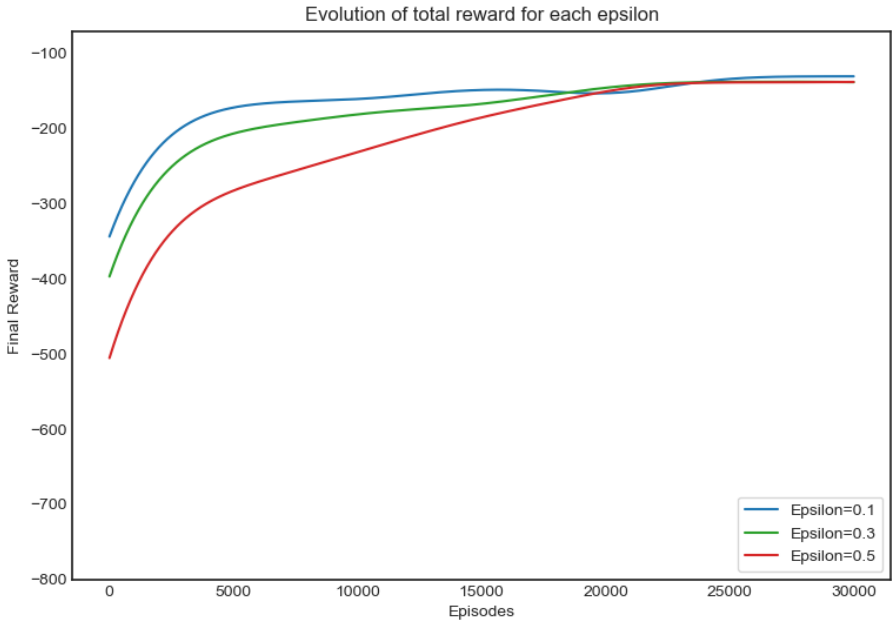

Figure 19: Trend of final reward for each starting value of epsilon

All the values of epsilon seem to converge to the same final total reward. In the first episodes the total reward is greater for smaller epsilons. Thus, smaller epsilon lead to faster convergence. This gives a better explanation on why the convergence has been obtained when the initial value of epsilon was set to 0.1 (Fig.14).
Yet, no conclusions can be made on the final performance. This might be due to the fact that the exploration time is still not enough.

3.3.5 Regression Methods

Another way of approximating the coefficients of the Q-function is using regression. For this method we update a parametric approximation instead of updating Q-tables.

ALGORITHM 3.2 Least-squares fitted Q-iteration with parametric approximation.

Input: discount factor γ,
approximation mapping F, samples $\{(x_{l_s}, u_{l_s}, x'_{l_s}, r_{l_s}) \mid l_s = 1, \ldots, n_s\}$

1: initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
2: **repeat** at every iteration $\ell = 0, 1, 2, \ldots$
3: **for** $l_s = 1, \ldots, n_s$ **do**
4: $Q^{\ddagger}_{\ell+1,l_s} \leftarrow r_{l_s} + \gamma \max_{u'} [F(\theta_\ell)](x'_{l_s}, u')$
5: **end for**
6: $\theta_{\ell+1} \leftarrow \theta^{\ddagger}$, where $\theta^{\ddagger} \subset \arg\min_\theta \sum_{l_s=1}^{n_s} \left(Q^{\ddagger}_{\ell+1,l_s} \quad [F(\theta)](x_{l_s}, u_{l_s}) \right)^2$
7: **until** $\theta_{\ell+1}$ is satisfactory

Output: $\widehat{\theta}^* = \theta_{\ell+1}$

Figure 20: Least-squares fitted Q-iteration algorithm (5)

The goal, starting with some states and target values is to find a function that can map the states to the target values.
The procedure to be implemented is the following :

- Initialize three Q polynomial functions, one for each action.
- Calculate the corresponding N Q-targets for N states. (N arbitrary, here N=1000)
- Update the coefficients of the Q-functions by regression where d, e and f store polynomial coefficients for actions 0, 1 and 2 respectively.
- Iterate the last two points.

Least square regression
This method has been chosen to use the *lstsq* function of numpy.linalg python package. It returns the coefficients of the least-squares solution to a linear matrix equation, solving the equation ax=b by computing a vector x that minimises the Euclidean 2-norm $||b–ax||^2$.
In this case, a matrix A is calculated depending on the action, the state and the degree of the wanted polynomial approximation. The linear matrix equation that represents the Q-function is then composed of the previously calculated target and the A matrix.
Also, it is important to consider that each target value is linked to the corresponding state (where it was evaluated) and therefore, we stocked both the states and target values in 3 lists, one for each action. Hence we can compute the regression for a given action and have a “match” between the states and the targets.

All this has been translated into code to get the new coefficients thanks to the following regression function.

```
def regression(action,states,T):
    X1 = (np.array(states[action]).T)[0]
    X2 = (np.array(states[action]).T)[1]
    A = np.array([X1*0+1, X1, X2, X1*X2, X1**2, X2**2]).T
    coeffs, r, rank, s = np.linalg.lstsq(A, T[action], rcond=None)
```

```
    return coeffs
```

Listing 14: Regression function

Polynomial approximation
Let X_1 be the position, X_2 the velocity and a_i with $i \in [1, 5]$ be the coefficients of the polynomial. The Q-function has been approximated by the following second order polynomial : $a_1X_1 + a_2X_2 + a_3(X_1X_2) + a_4X_1^2 + a_5X_2$
To get this polynomial the follow getpoly function is used :

```
def getpoly(X, d, e, f):
    Q0= d[0] + d[1]*X[0] + d[2]*X[1] + d[3]*X[0]*X[1] + d[4]*X[0]**2 + d[5]*X[1]**2
    Q1= e[0] + e[1]*X[0] + e[2]*X[1] + e[3]*X[0]*X[1] + e[4]*X[0]**2 + e[5]*X[1]**2
    Q2= f[0] + f[1]*X[0] + f[2]*X[1] + f[3]*X[0]*X[1] + f[4]*X[0]**2 + f[5]*X[1]**2
    Qarray = Q0, Q1, Q2
    return Qarray
```

Listing 15: Getpoly function

It reconstructs the Q function for given coefficients. It returns Qarrays which contains the three Q values at a point.

Target formulas
Two target formulas have been tested.

a) *Target function*
The target values try to match with the function, such as : $Q_{target} = reward + discount \cdot maxQ(s', a')$

These formulas are directly implemented into the main function.

```
Qarray = getpoly(np.array(state2), d, e, f)
Qtarget = reward + discount_factor * np.max(Qarray)
T[action].append(Qtarget)
```

Listing 16: Target function

b) *Alternative one*
In Busoniu's book [5], the Q values are often iterated with a weighted average of the current Q value and the target value, such as : $Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot Q_{target}$

Again, this has been implemented into the main of the alternative algorithm :

```
Qarray = getpoly(np.array(state2), d, e, f)
Qtarget = reward + discount_factor * np.max(Qarray)
label = (1 - learning_rate) * Qarray[action] + learning_rate * Qtarget
T[action].append(label)
```

Listing 17: Alternative function

Main
The main loop consists in concatenating states and targets in lists linked to the corresponding action. The target and states lists are initialised empty :

```
states = [[], [], []]
T = [[], [], []]
```

Listing 18: Target and state initialisation

The coefficients of the polynomials approximation are initialised to one :

```
d = np.ones((6,), dtype=int)
e = np.ones((6,), dtype=int)
f = np.ones((6,), dtype=int)
```

Listing 19: Coefficients initialisation

The other variables are also initialised.

```
Qarray = getpoly(np.array(state), d, e, f)
learning_rate = 0.5
EPISODES = 3500
discount_factor = 0.95
epsilon = 0.5 #hasard entre 0 et 1, a quel point il explore
START_EPSILON_DECAYING = 1
END_EPSILON_DECAYING = EPISODES // 2 #get an integer
epsilon_decay_value = epsilon/(END_EPSILON_DECAYING - START_EPSILON_DECAYING)
```

Listing 20: Variables initialisation

Then, the Q-values are computed for a given state thanks to getpoly function.

The action to play for each state is found with an epsilon greedy policy that allows the agent to explore its environment, the target value is computed and both states and target lists are filled. Note that the conditions to exit the while loop are either that the game is won or that 1000 actions have been tested.

```
for episode in range(EPISODES):
    state = env.reset()
    score = 0
    done = False
    steps = 0
    while not done:
        compteur+=1
        if np.random.random() > epsilon:
          action = np.argmax(Qarray) # action depuis Q table
        else:
          action = np.random.randint(0, env.action_space.n) #random
        states[action].append(state)
        state2, reward, done, _ = env.step(action)
        done = False
        Qarray = getpoly(np.array(state2), d, e, f)
        Qtarget = reward + discount_factor * np.max(Qarray)
        T[action].append(Qtarget)
        state = state2
        score += reward
```

Listing 21: Epsilon greedy policy

Then, the regression is computed on a batch of values :

```
if compteur % 20000 == 0:
    d = regression(0, states, T)
    e = regression(1, states, T)
    f = regression(2, states, T)
    states = [[], [], []]
    T = [[], [], []]
```

Listing 22: Regression step

The epsilon is decremented progressively :

```
if END_EPSILON_DECAYING >= episode >= START_EPSILON_DECAYING:
    epsilon -= epsilon_decay_value
```

Listing 23: Epsilon decrement

Results

a) *Target function*

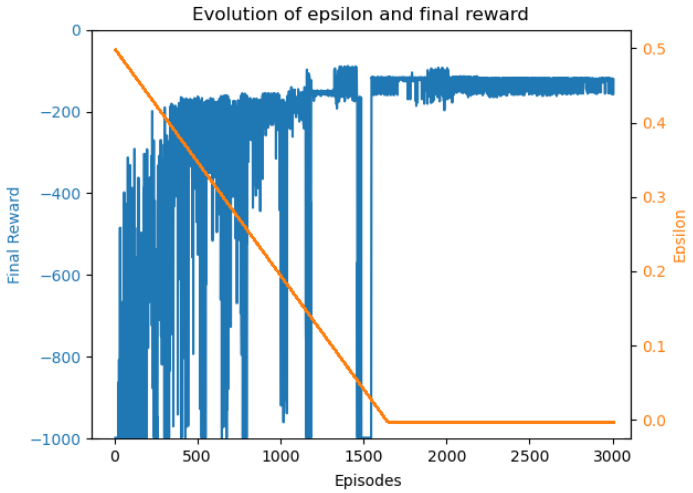

Figure 21: Evolution of epsilon and final reward

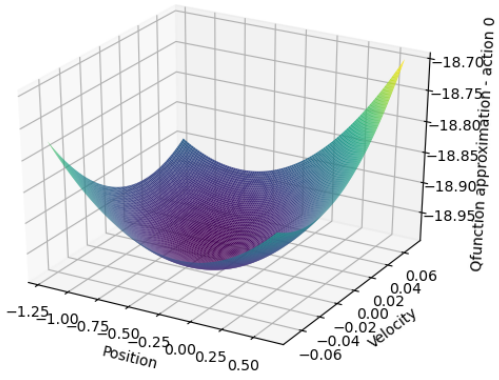

Figure 22: Q-function polynomial approximation of action 0

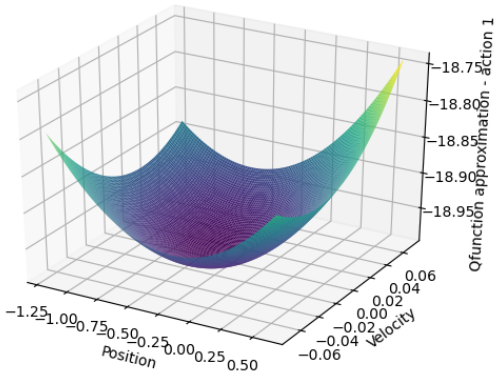

Figure 23: Q-function polynomial approximation of action 1

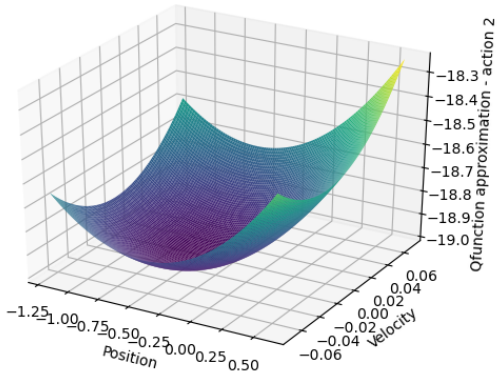

Figure 24: Q-function polynomial approximation of action 2

b) *Alternative one*

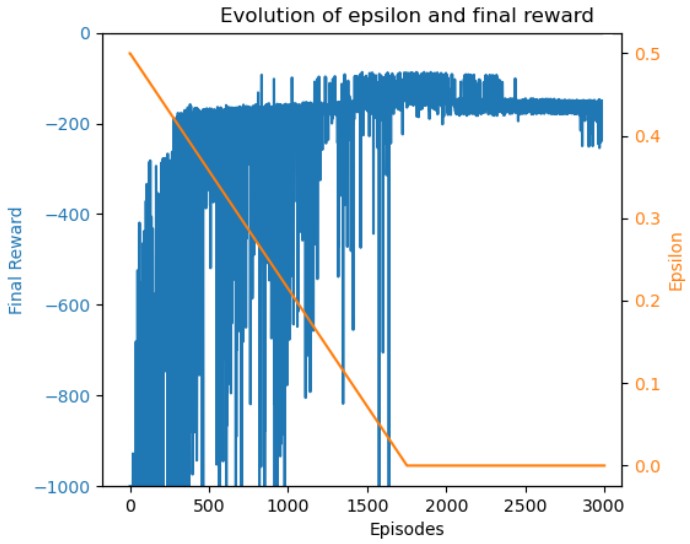

Figure 25: Evolution of epsilon and final reward

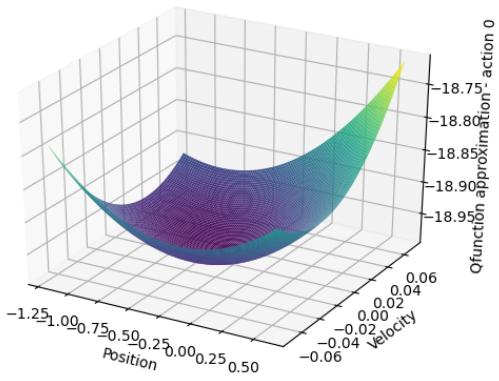

Figure 26: Q-function polynomial approximation of action 0

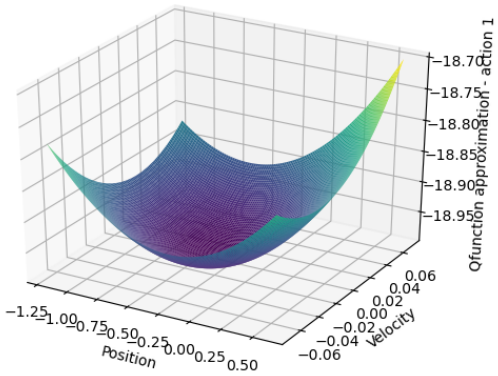

Figure 27: Q-function polynomial approximation of action 1

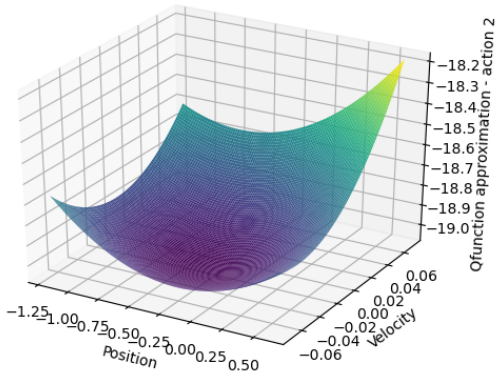

Figure 28: Q-function polynomial approximation of action 2

The Q-functions approximation looks quite the same for all actions and both methods. The shapes of the Q-functions approximations have the expected shape given the results obtained with the previously implemented methods and the graphs of the total reward show us that the two methods stabilize after about 1600 episodes.
Both methods converge towards the same value. They show a maximum total reward equal to -85 : the results

obtained seem now physically and theoretically correct.
To be sure, the renderings of the games have been plotted and it has been observed that the car follows the path which seems optimal to reach the top of the hill: climb on the left hill to reach a fairly high speed and manage to reach the top of the right hill (winning state) thanks to that speed.

Results comparison between Q-values, Polynomial approximation of Q-values and Regression

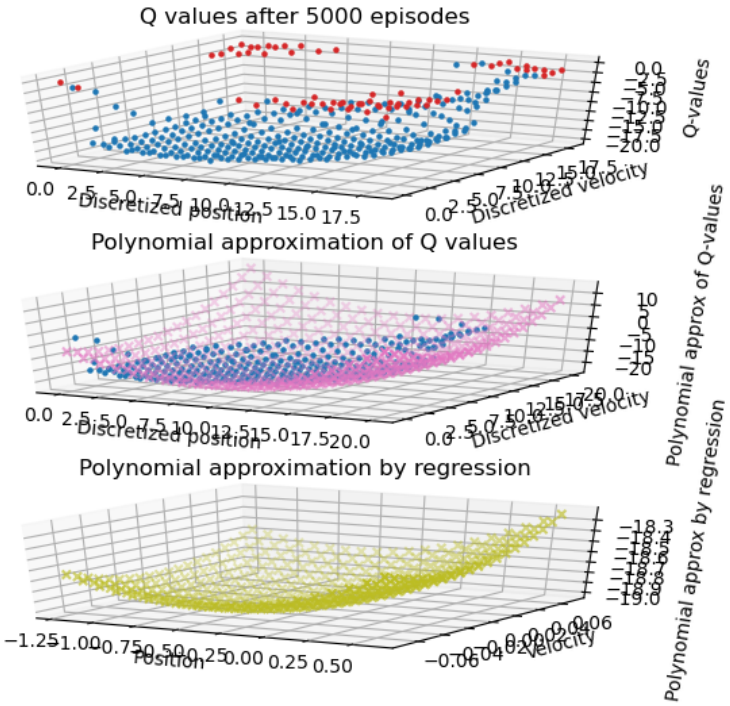

Figure 29: Plot comparison of three methods

The top graph in figure 29 shows Q-values corresponding to explored states (blue points) and the ones corresponding to non explored states (red points). These Q values has been calculated for a very large number of episodes (50000). Non explored Q-values stay at 0 (initialisation value). Explored Q-values gradually decrease until reaching a plateau at -20. However, some of them do not reach this plateau : these are in particular the Q-values which correspond to the states after the flag, therefore where the game is already won. Concerning the polynomial approximations, the shape is the same for both surfaces in all cases and the minimum reached is always around -20 (as Q-values). Nevertheless, the top values are considerably different between the two polynomial approximations : using regression they are much lower than the ones of the approximation of Q values. This can be explained as follow. The polynomial approximation obtained by regression adapts its shape only depending on the states it has gone through whereas the one based on Q values take into account also the non explored states (red points). However, the polynomial approximation of Q-values fits well also for the explored states (blue points) then the model should be correct.

4 Neural networks and Deep Reinforcement Learning

4.1 Motivation

Until this point the principle behind each algorithm was the same : explore a maximum number of states to approximate a value function that would give a good policy to the agent. This method was working pretty well. However, all the games on which these algorithms were implemented were showing a common point : all the possible states could be explored easily. Recalling the example of the maze, this one was perfect to implement a tabular learning algorithm as even in a case of a 10x10 maze the number of states was only 100 and with a set of 4 actions, the number of state-actions pairs -or Q-values- was no more than 400. Therefore, the agent could learn easily a good policy just by exploring all the possible combinations. The example of Mountain car is slightly more complex as the state is determined by a couple of positions and velocity. Also, unlike the maze, the range of positions is continuous. However, a discretization of the environment in twenty parts was enough for the Q-learning algorithm to be successful. In addition, the state is explicitly given the environment as a tuple of two numbers, which makes the task of determining Q-values particularly easy for the algorithm.

Yet, this is not always the case especially for more complex games such as Atari games. At the same time, Atari games present a great interest for reinforcement learning. Indeed, they present a limited number of actions (no more than 9) and integrate a system of rewards : the goal of each game is to maximise its score. Also, they can be easily emulated with openAI Gym and all have the same screen size. In these games, the state is no longer returned as a tuple indicating the position or velocity but rather as a three dimensional array of values indicating pixels. Indeed, a typical Atari game screen is a 210x160 image with RGB channels. Hence, each pixel takes a value between 0 and 255. Thus, we can have $255^{210*160} = 255^{33600} \approx 10^{70000}$ states. We can easily imagine that this environment is over complicated for a tabular leaning algorithm. Here comes the powerful convolutional neural network or CNN.
After introducing some algorithms doing a regression over the Q-values to approximate a polynomial Q-function, it is possible to succeed at Mountain Car with a second order polynomial function. In fact, neural networks are powerful and customizable function approximators. Neural network are able to fit high order polynomial. For this reason, it is also important to take care of not over-fitting the targeted functions. This adaptability to a wide range of functions makes the neural networks particularly suitable to approximate linear mapping both a state and an action onto a value. In addition, this can be made of different types of layers, we will use dense layers and convolution layers which are particularly efficient at dealing with raw pixels input as it will be discuss later.

4.2 What is a neural network?

Neural networks can be considered as a black box which links an input to an output with often a lot of non-linearity during the process. There are different types of networks such as feed-forward networks or convolutional networks which will be useful later when dealing with pixels.
Neural networks take the form of interconnected neurons also called perceptrons. A perceptron takes a given number of weighted inputs. The weights represent the importance given to each input. Then the perceptron integrates these different signals to produce an output. The associated transfer function is a linear transformation of the form :

$$G = X * w + b$$

where X is the input, w the weight and b the bias.

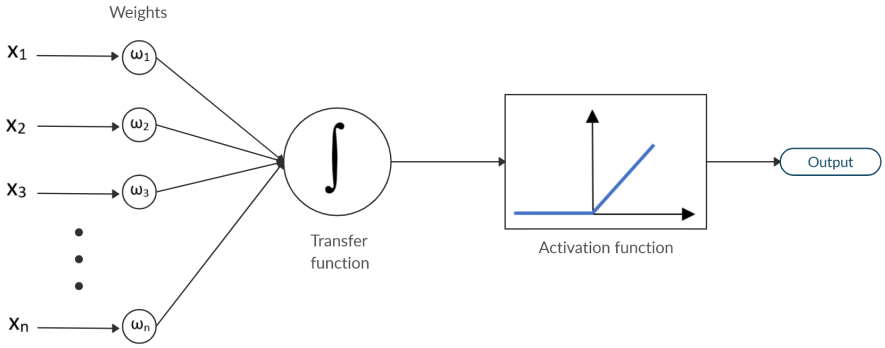

Figure 30: Scheme of a neuron

To introduce some non-linearity which is crucial to model complex problems we use an "activation function". In this project we used a popular activation function called the "rectified linear function" or "ReLu". We have

$$f(G) = max(0, G)$$

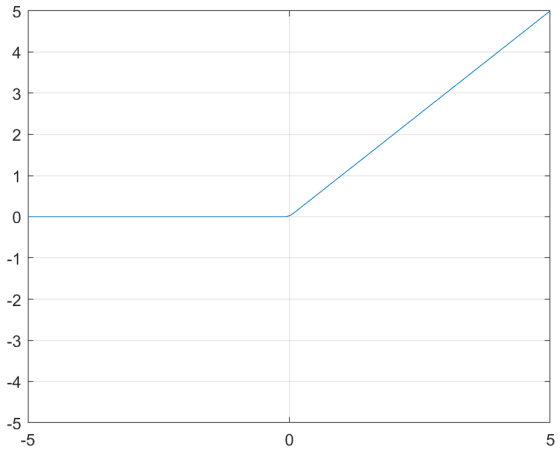

Figure 31: ReLu function

Hence, the output takes the value of the input only if it is positive. In fact, most of the time, this output is not necessarily the ouput of the network but may become on his turn the input of a new layer of neurons. Most neural networks present multiple hidden layers. The bigger number of neurons a neural network has, the bigger number of parameters (weights and biases) there are to tune. In this way, it is possible to imagine how flexible a neural network is.

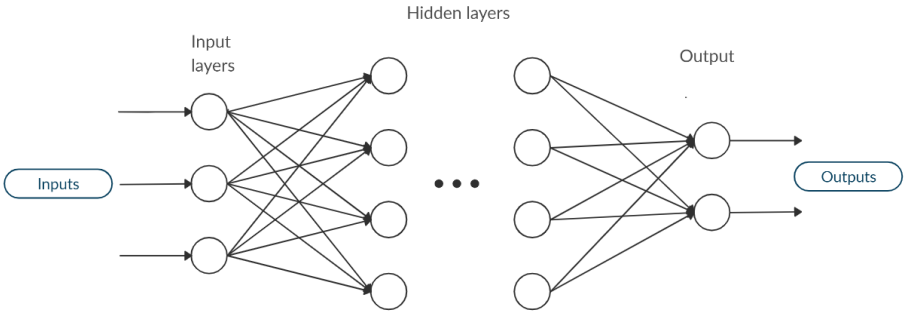

Figure 32: Illustration of a neural network

Determining these weights is then a key part of the performance of a neural network. At first, these weights are initialised either randomly or by following a specified distribution. Then, the network is fed forward from the input to the output by passing data through each layer at a time. Keeping in mind that the final goal of a neural network is to fit a target function with the maximum precision, the problem becomes an optimization problem. In this way, we have to compute a "loss" between the output of the network and the target function. The goal is to minimise this loss.

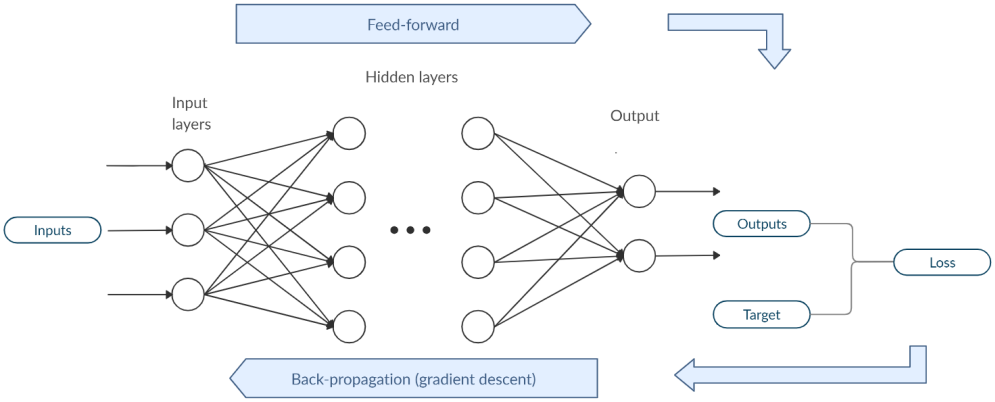

Figure 33: Illustration of how the parameters are updated

There exists several loss functions, two of them will be used here : the most popular one which is the the mean square error (MSE) and another one called the Huber loss. The MSE function is the squared error between the output of the network and the target value. It is averaged among each sample.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = 1$$

where y_i stands for the true value,$\hat{y}$ for the value predicted by the network and n is the number of sample points. Hence, the large errors are much more penalized than the smaller ones.

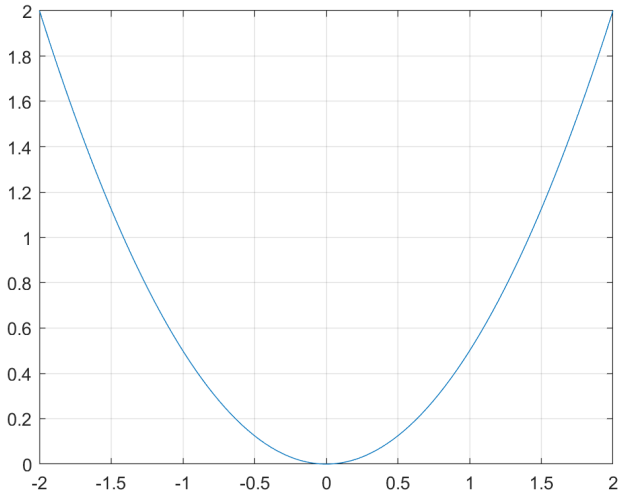

Figure 34: Mean Square Error

The Huber loss function is quite similar to the MSE. However, after a given threshold the function takes the absolute value of the error and no more its squared value:

$$H_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta\left(|a| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases}$$

This can be useful to avoid exploding gradients.

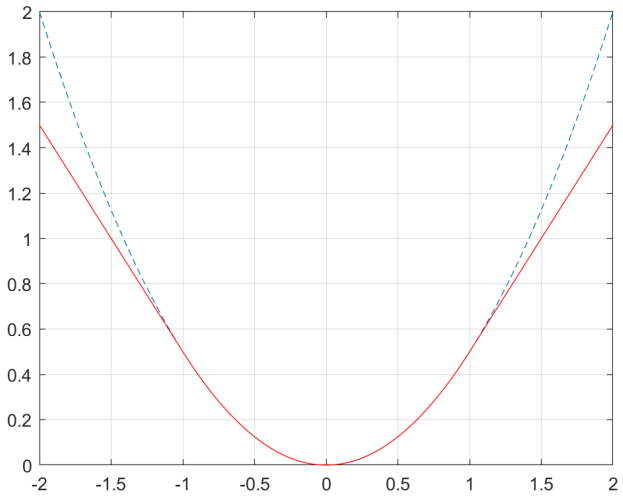

Figure 35: Huber loss (in red) vs MSE (in dashed blue)

The final step of the updating process is the backpropagation. The optimization problem consists in finding a global minima where the loss value is the smallest possible. Therefore, some steps from an initial point to reach this minima. The gradient descent algorithm allows us to find these steps. It uses the partial derivatives of the loss function with respect to the weights and biases. The value of the respective gradients multiplied by a learning rate is then subtracted to the values of the weight and biases in order to reach a new point where the gradient values are closer to zero.

$$w' = w - lr * \delta w$$
$$b' = b - lr * \delta b$$

where w' and b' are respectively the updated values of the weight and bias, w and b the current value, lr is the learning rate, δw and δb are the gradient's value with respect to w and b

The learning rate is actually a really important hyperparameter to have an efficient network. Indeed, if the learning rate is too big, the descent steps might be too large to find the minima. On the other hand if it is too small, the steps may converge to a local minima which is not recommended.

4.3 Use of Pytorch

To deal with neural networks, we needed a deep learning tool. Pytorch is one of them and happens to be really adapted to our problem. Pytorch is an open source machine learning framework. The following parts will explain how it can be used.

4.3.1 Tensors

Pytorch is built on what is called tensors. The use of tensors in Pytorch is really similar to the use of arrays in Numpy. They allow all the matrix computations on which lie the neural networks. They can be either one or multi dimensional (dimensions are referred as their rank).

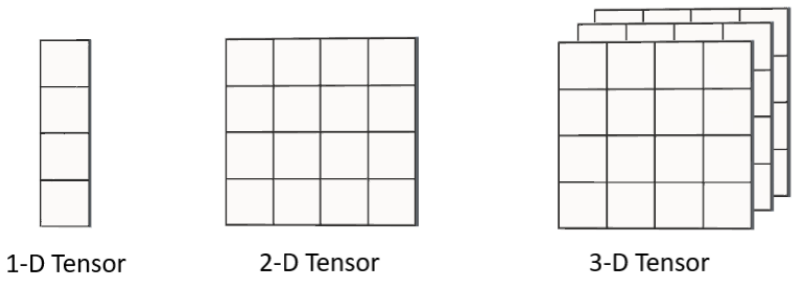

Figure 36: Different shapes of tensors

However, what makes Pytorch's tensors much more suitable for deep learning than numpy arrays is the fact that they are particularly adapted to run on GPU. It will appear later that, when the complexity rises, the use of a GPU becomes a necessity. The `torch.device()` command can be used to select the type of execution ('cpu' or 'cuda'for GPU).

4.3.2 Tools to build a neural network

The neural networks can be built with the `nn.Module` class. The `super().__init__` method can be used to create a customized neural netwok. For example the `nn.Linear` class allows to build a fully connected feed-forward layer. This class takes the input and the output dimensions as arguments and allows to set biases or not. The `nn.functional` class contains the activation functions such as ReLu. Hence, we can define our first neural network:

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):  # nn model  inputs --> 64layer ---> 64layer ----> outputs
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 64, bias=False)
        self.fc2 = nn.Linear(64, 64, bias=False)
        self.fc3 = nn.Linear(64, output_dim, bias=True)
        self.device = torch.device("cpu")

    def forward(self, x):  # forward function
        inputs = torch.Tensor(x).to(self.device)
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        outputs = self.fc3(x)
        return outputs
```

Listing 24: Simple DQN network

As a rule of thumb, the number of neurons is chosen as a power of 2. This network contains a single hidden layer with 64 neurons and no bias. Now that the feed-forward has been done, we need tools to compute the loss and do the backpropagation. We can also find the wanted loss function in the `nn.functional` class. For example `functional.mse_loss` gives the MSE loss function and `functional.smooth_l1_loss` gives the Huber loss. As described above, these functions take the predicted values and targeted values as inputs in order to compute loss values.

Finally, we need a tool to operate our gradient descent. This is called an optimizer. The optimizer uses the `autograd` library of pytorch. This package provides automatic differentiation which becomes very useful in the context of gradient descent. The `.backward()` function computes the gradients. The optimizer determines the steps performed during each gradient descent. It is defined in the `torch.optim` package. We use an optimizer called `Adam`. It takes the `.parameters()` (weights and biases) and learning rate as input. It shows outstanding performances for a large amount of problems. It introduces some stochasticity in the gradient descent process, which has been proven to bring a better performance.

With these tools it is possible to perform the whole training process:

```
dqn = DQN(input_dim, output_dim)
optimizer = optim.Adam(dqn.parameters(), lr=0.005)
def train(self):
    y_hat = dqn(input)
    self.optimizer.zero_grad()  # optimizer has to be emptied at each step
    loss = F.mse_loss(target, y_hat)  # loss between target and predicted value
    loss.backward()  # backward propagation
    self.optimizer.step()  # step optimizer -----> improves network
```

Listing 25: Q-function fitting network

The first task we can do to check if our networks actually work is to try to approximate a Q-function that we obtained by regression in the previous algorithms. The Q-function is fitted at 100 different sample states that have been generated randomly in the allowed ranges of position and velocity. This approach can bring us many useful

information. Indeed if our network is able to fit a MountainCar's Q-function then it will certainly be able to deal with the whole game. In addition we can use this algorithm to tune the hyperparameters that we will use later to play MountainCar. The algorithm is the following :

```
import numpy as np
import torch
import random
from torch import nn
import torch.optim as optim
import torch.nn.functional as F

class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):  # nn model 2 states --> 64layer ---> 64layer ----> 1 Q-value
        super().__init__()
        self.fc1 = nn.Linear(state_dim, 120, bias=True)  # 2d state space
        #self.fc2 = nn.Linear(64, 64, bias=False)
        self.fc3 = nn.Linear(120, action_dim, bias=True)  # one output per action
        self.device = torch.device("cpu")

    def forward(self, x):  # forward function
        state = torch.Tensor(x).to(self.device)
        x = F.relu(self.fc1(state))
        #x = F.relu(self.fc2(x))
        actions = self.fc3(x)
        return actions

def f1(x, y): # Q-function
  return -17.853867 + 0.2989069*x -0.81406932*y -1.74610622*x*y +0.27098633*x**2 +33.39477236*y**2

dqn = DQN(2,1)

state = [(random.uniform(-1.2, 0.6), random.uniform(-0.07, 0.07)) for i in range(100)]  #creating 100 random states tuples
value = []
for i in range (100):  #evaluating their value through Q-functiom
  value0 = f1(state[i][0], state[i][1])
  value.append(value0)

optimizer = optim.Adam(dqn.parameters(), lr=0.005)

for t in range(500):
  y_hat = dqn(state)
  optimizer.zero_grad()
  #loss = F.mse_loss(y_hat, torch.FloatTensor(value))  # mse loss
  loss = F.smooth_l1_loss(y_hat, torch.FloatTensor(value)) # huber loss
  print(f'\rstep = {t},loss = {loss}')
  loss.backward()  # backward
  optimizer.step()  # step optimizer -----> improves network
```

Listing 26: Training loop

By printing the loss it can be observed how fast it is decreasing. Here 500 gradient descent steps are performed. The result is the following:

```
step = 0,loss = 17.386871337890625
step = 1,loss = 17.264118194580078
step = 2,loss = 17.14182472229004
step = 3,loss = 17.0196533203125
```

```
step = 4,loss = 16.8973445892334
step = 5,loss = 16.774616241455078
[...]
step = 495,loss = 0.010644471272826195
step = 496,loss = 0.010643969289958477
step = 497,loss = 0.010643470101058483
step = 498,loss = 0.010642987675964832
step = 499,loss = 0.010642506182193756
```

After Looking at the loss values while modifying some parameters of the networks, some modifications were made. Indeed, it was observed that two layers (instead of three previously) of neurons were enough to properly approximate the Q-function. Although, each layer has to contain a bigger number of neurons (see the algorithm above). This procedure was also conducted to determine a satisfying learning rate for the optimizer (0.05 in this case). Finally, by plotting the predicted function and comparing it to the targeted one, it was observed that the Huber loss function was a little more accurate than the MSE one.

4.4 Deep Q-learning

4.4.1 Interacting with the environment

With this working network, it is now possible to create a Deep Q-learner. In the last example the values of the states that were feeding the network were completely random. Now the goal is to interact with the environment in order to sample some actual states of the games to feed the network. However, the states are not enough to train the network, they have to be linked to the action that provoked the transition between one state and the corresponding next state. Also, to compute the target values that the network has to match, it is necessary to know the reward associated to one action as well as if it resulted in ending the game (the game is 'done'). Therefore, a tuple of (state, action, reward, next state, done) has to be collected at each step. The target function follows the Bellman's equation (see part on Q-learning) and is computed as follows:

$$Q_{target} = \begin{cases} r + \gamma \max_{a' \in A} Q_{s',a'} & \text{if not done} \\ r & \text{otherwise} \end{cases}$$

Where r is the reward, γ the learning rate, s and a respectively the state and the action, s' and a' next state and next action. It follows that the MSE loss is :

$$L_{MSE} = \begin{cases} [Q(s,a) - (r + \gamma \max_{a' \in A} Q_{s',a'})]^2 & \text{if not done} \\ \big(Q(s,a) - r\big)^2 & \text{otherwise} \end{cases}$$

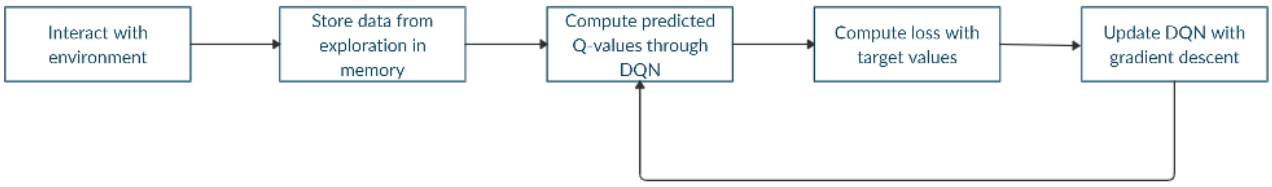

Figure 37: Interaction of the DQN with its environment

This approach is still lacking some tools to be efficient. There are multiple reasons to that. First, feeding the network with all the data stored in memory at each step would become really demanding in terms of computational power and would slow down the process as things progress. It is therefore necessary to feed the network with a batch of fixed six. A batch size that performs well can be for example 32 as it is the size that will be used in further developments. Please note that bigger batches can be used if enough computational power is available.

Yet, if the batch takes in memory samples in the order as the come in the game it won't be an optimal way of feeding the network either. In fact, the neighbouring states are strongly correlated. This is problematic in the context of a Markov decision process. Indeed, a Markov process implies that the training data has to be independent and identically distributed (i.i.d.). In this way, if the states are too close from each other it won't benefit to the network. In order to make the training data more independent it is possible to store the collected data in a big memory (it can be 100k or even 1M samples) and at each step to randomly create a batch from this data. For example, if 32 samples (state, action, reward, next state, done) are randomly picked out from this big memory of 100k elements,

these samples have much more chances to be independent. This big memory can be, at first initialised with a random policy and then, when the number of elements exceed its capacity, the entering elements can push the oldest one out of the memory. In this way the data stored in memory can remain relatively fresh and relevant.

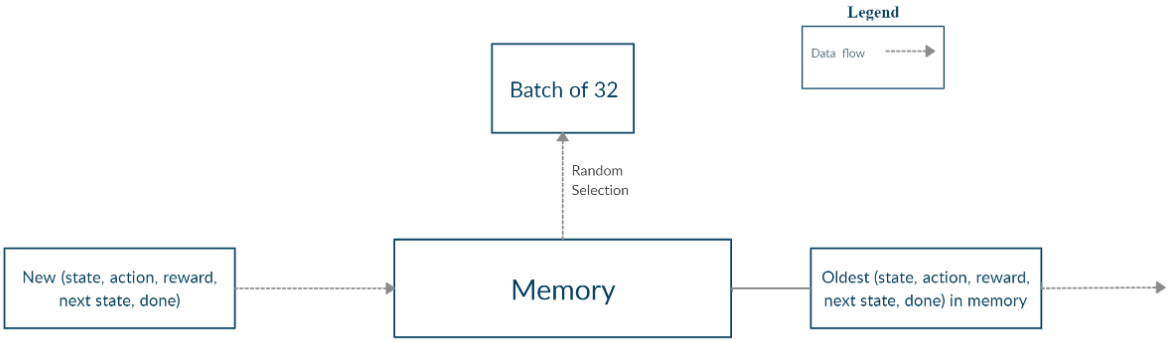

Figure 38: Evolution of memory at each step of the game

This can be implemented in the code as shown below:

```
if step < 100000: # 100k elements memory
    agent.memory.append(None)
    agent.memory[step % 100000] = (state, action, reward, next_state, done) #after 100k steps,
    overwrite the oldest data

if step >= 10000: #training begin after 10k steps
    sample = random.sample(agent.memory, BATCH_SIZE)
    agent.update(sample, step)
    agent.train()
```

Listing 27: Programmation of the memory (main loop)

The `update()` and `train()` functions will be detailed later on.

This use of the memory helps to avoid too much correlation in the inputs of the network. However, the training process is still disturbed by some correlation. Indeed, until this point the target function is evaluated at every step. Consequently both the predicted Q-function and to Q-target depend of the network parameters at each steps. Therefore, the network is trying to fit a target that is relatively unstable and moving. Indeed, the states s and s' are very similar and updating $Q(s, a)$ is likely to have impacts on Q(s', a'). The modifications on $Q(s', a')$ can alter the next prediction of $Q(s, a)$ and so on... This interference is not beneficial for the learning process. It is therefore recommended to use a target network that directly copies the parameters every x steps. Here x has to be large (in the order of 1k or even 10k). In this way the target network is frozen for a given number of steps and allows the network to fit a more stable target. Updating the target network every 1000 steps, for example, allows to maintain a relevant target that is relatively up-to-date.

Finally we can draw a comprehensive diagram for the learning process:

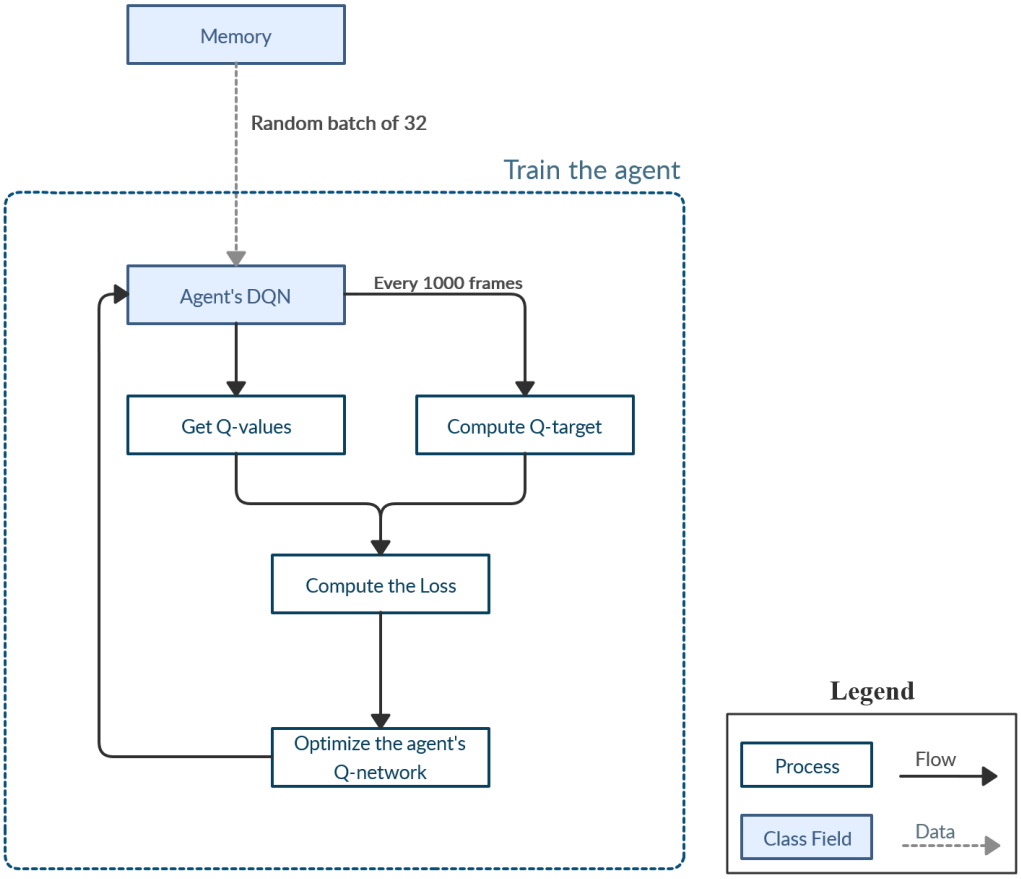

Figure 39: Diagram describing the whole training process

This can be translated into code as:

```
class Agent:
    def __init__(self, state_dim, action_dim):
        self.network = DQN(state_dim, action_dim).to(device)  # creates network from nn model
        self.target_network = copy.deepcopy(self.network).to(device)  # create target_network from
    network
        self.optimizer = optim.Adam(self.network.parameters(), lr=0.005)  # optimizer, parameters
        self.memory = []  #initialize empty memory
        self.states_memory = []
        self.actions_memory = []
        self.rewards_memory = []
        self.next_states_memory = []
        self.dones_memory = []

    def update(self, batch, update_dqn_target):
        states, actions, rewards, next_states, dones = zip(*batch)  # extracts data from batch
        self.states_memory = torch.from_numpy(np.array(states)).float().to(device)  # collect
    states
        self.actions_memory = torch.from_numpy(np.array(actions)).to(device).unsqueeze(1)  #
    collect actions
```

```
        self.rewards_memory = torch.from_numpy(np.array(rewards)).float().to(device).unsqueeze(1)
    # collect rewards
        self.next_states_memory = torch.from_numpy(np.array(next_states)).float().to(device)  # etc
        self.dones_memory = torch.from_numpy(np.array(dones)).to(device).unsqueeze(1)
        if update_dqn_target % UPDATE_TARGET == 0: #update target network
            agent.target_network = copy.deepcopy(agent.network)

    def train(self):
        target = self.rewards_memory + (gamma * self.target_network(self.next_states_memory).detach
    ().max(1)[0].unsqueeze(1)) * (~self.dones_memory)  # q-target
        Q_current = self.network(self.states_memory).gather(-1, self.actions_memory)  # Q_current,
     gather will index the rows of the q-values
        self.optimizer.zero_grad()  # optimizer
        loss = F.smooth_l1_loss(target, Q_current)  # Huber loss
        loss.backward()  # backward propagation
        self.optimizer.step()  # step optimizer -----> improves network
```

Listing 28: Agent class

This Pytorch implementation of our training process may still contain some unclear parts as it includes new tools and has to be explained. For example the `copy.deepcopy()` function allows to accurately copy all the parameters from the DQN into the target network. Once the data has been collected from the batch it is transformed into tensors from numpy in order to be treated by the neural network (see previous part on Pytorch's tensors). The `.detach()` to avoid automatic differentiation on this particular tensor, when no gradient descent has to be done. In the computation of the target function the command : `~self.dones_memory` allows to multiply the previous expression by 0 if the game is done and 1 otherwise, to respect the expression of the Q-target function (see above). Finally, the `.gather()` is a really useful tool. Indeed, it produces a one dimensional tensor of Q-values as shown:

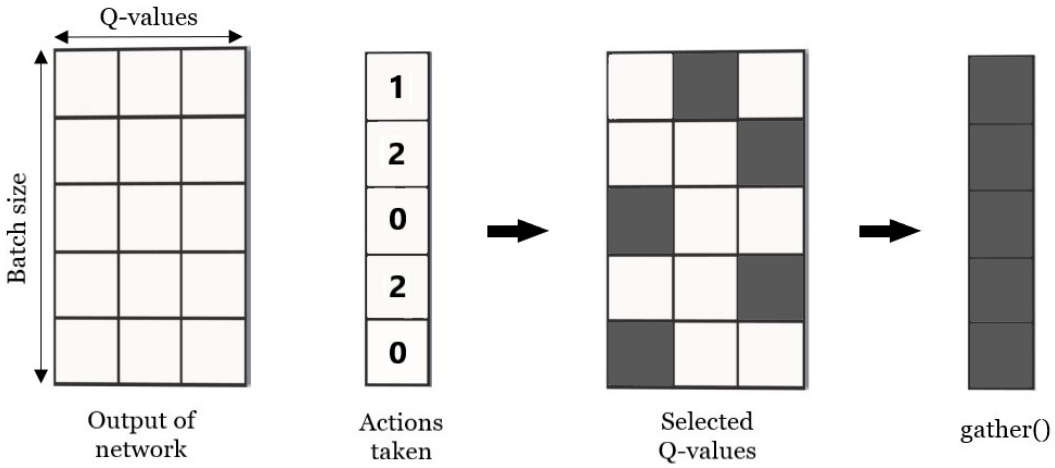

Figure 40: Use of the gather() function

4.4.2 Acting in the environment

Now that the DQN is linked to the environment, the agent has to take some actions in order to move in the environment. At a first stage, the agent do not have any specific policy and therefore has to explore its environment to find the states and actions associated to a good reward. To obtain this behaviour an epsilon greedy policy is used

as it has been done in the previous part. The agent then chooses an action according to the following strategy:

$$action = \begin{cases} \text{random with probability } \epsilon \\ \text{best action according to policy with probability (1-}\epsilon\text{)} \end{cases}$$

ϵ is initialised at 1 and decay for a given number of step. The decay value of ϵ is actually a key hyperparameter to tune. Indeed, if ϵ decreases too quickly the agent might stick to the first strategies it discovers. If ϵ decreases too slowly, the agent do not exploit what it has learnt. ϵ can be decreased as a geometrical sequence (`eps *= eps_coeff`) or as an arithmetic sequence (`eps -= decay_value`). After reaching a certain threshold (such as 0.02) ϵ is kept constant in order to keep exploring for better strategies.

Translated into code we have:

```
class Agent:
    def __init__(self, state_dim, action_dim):
        [...]

    def update(self, batch, update_dqn_target):
        [...]

    def train(self):
        [...]

    def act(self, env, state, eps):
        if random.random() < eps:
            return env.action_space.sample() #random action
        state = torch.tensor(state).to(device).float()
        Q_values = self.network(state.unsqueeze(0)).detach() #Output of the network for the state
    in argument
        return np.argmax(Q_values.cpu().data.numpy())  # choose action with argmax
```

Listing 29: Complete agent class

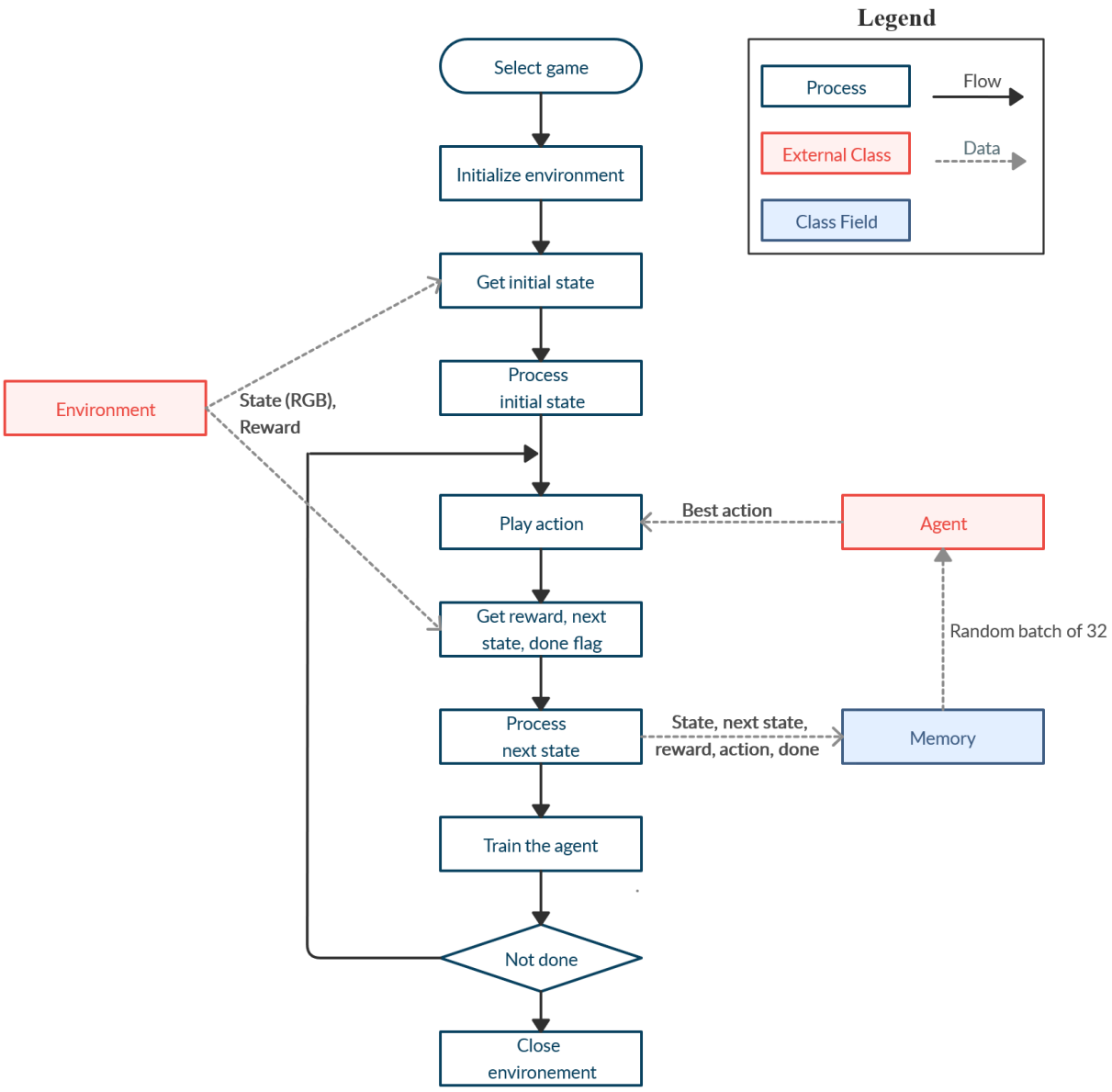

Figure 41: Implementation of the deep Q-learning algorithm

4.5 Mountain Car

4.6 Adaptation of the DQN to the game

Now that a deep learning algorithm has been created, it is time to apply it to games. The first game that is going to be played is MountainCar. In the previous parts other algorithms were implemented on this game in order to have some references. The comparison between the different methods can also be interesting.

As a reminder, in the MountainCar game, the agent has a set of three actions and the environment returns the state as a tuple of two values (position and velocity). Therefore, the network takes two inputs and must outputs three Q-values. The middle part of the network can be tuned by adjusting the number of hidden layers or the number of neurons. In the part 4.3.2, an actual Q-function of the MountainCar problem was fitted. It gave the opportunity to fine tune the different parameters. It was observed that two fully connected layers were enough to approximate the Q-value. Therefore, the following DQN class was implemented :

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):  # nn model 2 states --> 64layer ---> 64layer ----> 3 Q-action pairs
        super().__init__()
        self.fc1 = nn.Linear(state_dim, 512, bias=False)  # 2d state space
        self.fc2 = nn.Linear(512, action_dim, bias=True)  # one Q-value per action
        self.device = torch.device("cpu")

    def forward(self, x):  # forward function
        state = torch.Tensor(x).to(self.device)
        x = F.relu(self.fc1(state))
        q_values = self.fc2(x)
        return q_values
```

Listing 30: DQN class for MountainCar

There is no ReLu activation for the last layer during the forward pass as the Q_values can take negatives values. The Q_values in MountainCar are actually all negatives.

4.6.1 Hyperparameters

We call hyperparameters all the parameters that can be tuned. They are often assigned a fixed value. For every game, finding the optimalset of hyperparameters is a key task. A lot of time has been spent comparing the results issued from different hyperparameters. A difficulty that has been faced is the fact that these parameters are not necessarily independent from each others. Indeed, changing the value of a single hyperparameter may have influence on the other ones. For example, modifying the learning rate may have some repercussions on how often the target function has to be updated. In this way the hyperparameters' value can only be obtained empirically with many trials. They were chosen as follows :

```
learning_rate = 0.005
batch_size = 64
target_update = 150
gamma = 0.99
epsilon = 1 #at the beginning
decay_value = 0.995
```

A geometric decay has been used for `epsilon` (see part 4.4.2). It has been observed that the algorithm performance is very sensible to the update frequency of the target network, if the `target_update` parameter, the target seems to be rapidly outdated. Also, the learning rate has to be chosen with cautiousness. Indeed the learning process is really sensible to it: For example a learning rate of `0.005` shows a good learning behaviour whereas a learning rate

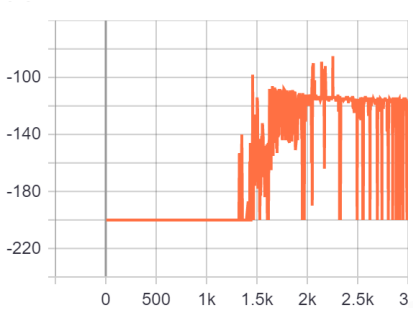

Figure 42: Learning rate = 0.001

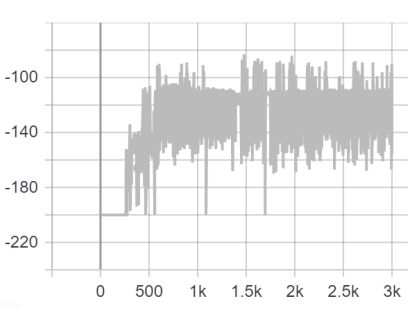

Figure 43: Learning rate = 0.005

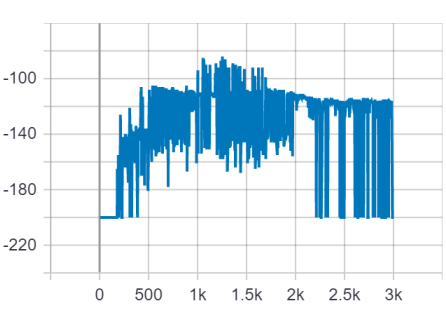

Figure 44: Learning rate = 0.01

of `0.001` penalises the agent. During the first implementations of the DQN to MountainCar, it was often observed that after a while, the agent was getting worse. This behaviour can be observed in figure 25 and figure 27. The only difference between these three results is the learning rate although the difference between them is relatively small. These plots happen to be really meaningful. The first thing that can be observed on these three plots is the fact that the smaller is the learning rate, the faster the agent wins. This is in accordance with part 4.2. Indeed, with a large learning rate, the gradient descent steps are bigger and the and the agent learns consequently faster. Also, it is observed that when the learning rate is either larger or bigger than `0.005` the policy drops after having been successful for some time. Indeed, as mentioned in part 4.2, if the descent steps are too small, the optimizer discover a local minimum which is not the best policy. On the other hand if the steps are too big, the optimizer might have trouble to stabilise in a global minimum (in fact it may oscillate between non-optimal regions).

Concerning the number of neurons in the network, which can be considered as an hyperparameter as well, some optimization has also been done. In fact, a bigger number of neurons is not necessarily associated with a better precision. The reason behind this is the fact that with a higher number of neurons, a network is able to fit a higher order function. If the order of the approximated function is not so high, the network can overfit the target function.

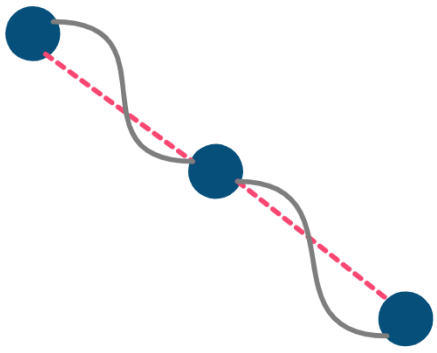

Figure 45: Representation of an overfitted approximation (in black) compared with the target function (in dashed red)

To illustrate this, on the figure above, the target function is a linear function (first order) and the approximation creates some error due to a too high order.

Final performance of the algorithm: with the right set of hyperparameters, the agent had a mean reward of about -130 over the first 3K episodes and an accuracy of 2700/3000 (number of won games). Finally, it is interesting to note that this algorithm initially written for MountainCar can be directly implemented to other Classic control environment such as "CartPole" and is performing relatively well.

4.7 From game's states to pixels

The previous example was showing good performance. However, in order to move on to more complex games, the DQN has to deal with a bigger number of inputs. Also the state is not always explicitely returned by the environment as it is the case in MountainCar. The most universal way to deal with a video game is to take pixel as input just like a human player does. Therefore, the DQN has to learn to deal with pixels as inputs.

4.7.1 Convolutional layers

Convolutional neural networks (or CNN) are particularly good at dealing with images. Flattening the pixels of an image to make a 1D array in order to use it as input to the network would imply a very large number of inputs (an Atari screen has 210x160 pixels). In addition, flattening the image would lose the relations that the nearby pixels have between them and therefore alter the meaning of the image. Thus, a method that does not alter the spatial dependencies of the image is needed. Convolutional neural networks fulfil this task as they divide the image in small matrices in order to process them. It takes the form of a window moving across the image. Each extracted group of pixels (the size of the window) is assigned some parameters (weight and bias) as it is the case with fully connected layers. This window has often a square shape, it is called a kernel. In the DQN, it is possible to attribute a size to the kernel. For example with `kernel_size=8` creates a 8x8 window. It will therefore analyse 64 pixels at time. An equation gives the dimensions of the output. If an image has the dimension (height, width, depth) the output's dimension will be :

$$
\begin{aligned}
height_{output} &= height - kernelsize + 1 \\
width_{output} &= width - kernelsize + 1 \\
depth_{output} &= 1
\end{aligned}
$$

For example a typical Atari screen frame of dimensions (210x160x3) passing through a 8x8 window would be processed as :

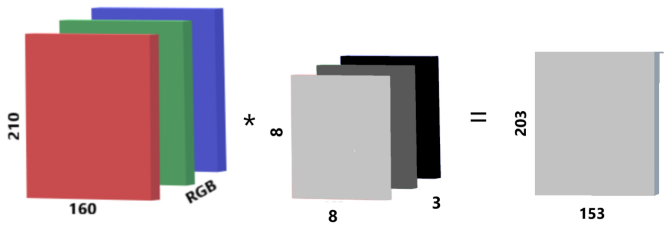

Figure 46: Output of an RGB image after the convolution through a single window

In this case the output is a 2-D matrix. However the image can be filtered by more than one window. The number of windows used gives the depth of the output matrix. In Pytorch, the number of filters can be set with the argument `output_channel` (the term channel designate the depth of the tensor). The number of `output_channel` typically increased in each convolutional layer while the `kernel_size` decreases.

Until this point, the windows process every subsection of the image with steps of one pixel at each time. This method produces an important overlap between the different chunks of image and is also very slow. For this reason, a `stride` is defined. The value of the `stride` is the step, in pixels, between two processing. A good trade-off is to choose a stride in the order of the half of the kernel size. In this way, the overlap is only 50%. Using a stride different than 1 changes the shape of the output:

$$height_{output} = \frac{height - kernelsize}{stride} + 1$$
$$width_{output} = \frac{width - kernelsize}{stride} + 1$$

The reason behind computing these values, is the fact that in order to output the Q-values the DQN has to be ended by fully connected linear layers. Thus, knowing the ouput dimensions of the CNN allows to make the transition with the last part of the DQN.

4.7.2 New DQN

With the elements discussed in the previous part, it is possible to build a DQN able to deal with image inputs. To do the regression task, three convolutional layers followed by a single fully connected layer will be used. This is a rather common shape for a DQN. At first, the number of filters in the convolutional layers will be relatively small and increased if necessary. As before a ReLu activation is used for the forward pass.

```
class DQN(nn.Module):

    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(state_dim[0], 16, kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=2)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=2)

        def conv2d_out(size, kernel_size = 5, stride = 2): #comput the dimension of the output
            return (size - kernel_size) // stride  + 1
        convh = conv2d_out(conv2d_out(conv2d_out(state_dim[1]))) #height of the output after the 3
    conv layers
        convw = conv2d_out(conv2d_out(conv2d_out(state_dim[2]))) #width of the output
        linear_input_size = convw * convh * 32  #flatten the output of CNN

        self.fc1 = nn.Linear(linear_input_size, action_dim)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        q_values = self.fc1(x.view(x.size(0), -1))
        return q_values
```

Listing 31: DQN class for MountainCar

Similarly to the choice of the number of neurons in a fully connected layer, the number of filters is chosen as a power of 2. An element that has to be taken into account is the fact that the input of convolutional layers in Pytorch needs have to shape (channel, height, width) instead of (height, width, channel) as images are often presented. Therefore,

`state_dim[0]` designs the number of channels.

Another tool that hasn't been used but has to be mentionned is called `nn.BatchNorm2d()`. It can be used to normalise the data in order to facilitate the process. As the convolutional layers are processing a large amount of pixels it is important to optimize the input data. A well pre-processed input optimizes the runtime without decreasing the quality of the output.

Please note that, when the network uses convolutional layers, it becomes a necessity to run on the GPU, `torch.device()="cuda` is used.

4.7.3 Producing an observable state from pixels in MountainCar

In MountainCar, the screen can be obtained with `screen = env.render(mode = 'rgb_array')`. However, this has a major drawback : using the `screen = env.render()` functions inevitably display the environment. Consequently one episode of training lasts at least the actual time of an episode (200 episodes at 30 fps, which means about 7s). By adding the processing of the image at each time step the training becomes very slow. For this reason it wasn't possible to do many trials on MountainCar in order to optimise it and very good results couldn't be achieved. In order to optimise the training process, different methods of image processing have been developed.

First, it has to be observed that using a single image as a state is not enough in the context of a Markov decision process. Indeed, to be considered as a Markovian process, each state must be fully observable. In other words each state must correspond to a unique situation. For example, in MountainCar, it is possible to take two screenshots looking exactly the same although on first one the car is going downhill while it is going uphill on the second one. Therefore, the state has to give information about the direction, the velocity and eventually the acceleration. Stacking 4 frames on top of each other and considering them as a state is a solution to this problem. More concretely, at each step the next state is stacked with the 3 previous screens. In this way, the network has 4 `input_channel` (see later how to reduce the depth of the image to 1). However, doing this procedure at each step wouldn't be efficient. That would cause unnecessary computations and could even be counterproductive. Indeed, as evoked before, the states separated by only one step in the game are too close to each other and the transition is so small that it doesn't give a good information about the dynamic of the game. It is therefore much more productive to interact with the environment every four frames of the game. By doing this, the agent takes an action every four steps and its also contributes to make a much more natural render. Translated into code:

```
for i in range(episodes):
    env.reset()
    state = (env.render(mode = 'rgb_array'))#get screen from render

    stacked_frames = np.stack((state, state, state, state), axis=2) #the image has shape (h, w, c)
    -->stack over dime 2
    stacked_frames = np.reshape([stacked_frames], (4, 400, 600)) #reshape according to pytorch
    convention --> (c, h, w)
    state = stacked_frames

    done = False
    total_reward = 0
    while not done:
        step += 1
        action = agent.act(env, state, eps)
        _, reward, done, _ = env.step(action)
        total_reward += reward

        if step % 4 == 0:
            next_state = env.render(mode = 'rgb_array')
```

```
            next_state = np.reshape([next_state], (1, 400, 600))##reshape according to pytorch convention
            next_sf = np.append(next_state, stacked_frames[:3, :, :], axis=0) #the image has shape (c, h, w) --> stack over dim 0
        state = next_sf
```

Listing 32: Producing a state from stack images

However by looking at this code, it appears clearly that the screen size and therefore the number of pixels to be processed is too big. Some strategies have to be find to reduce the number of input pixels.

4.7.4 Cropping the image

The most obvious thing to do in order to reduce the number of pixels in the input is to crop the image in order to get rid of unnecessary information. However in MountainCar, the car can be in a wide range of positions in directions x and y of the image. Thus, there is not much to crop if the objective is to keep a rectangular image (better for convolutions). At this point, an idea was to follow the position of the car with a window of 24x24 pixels.

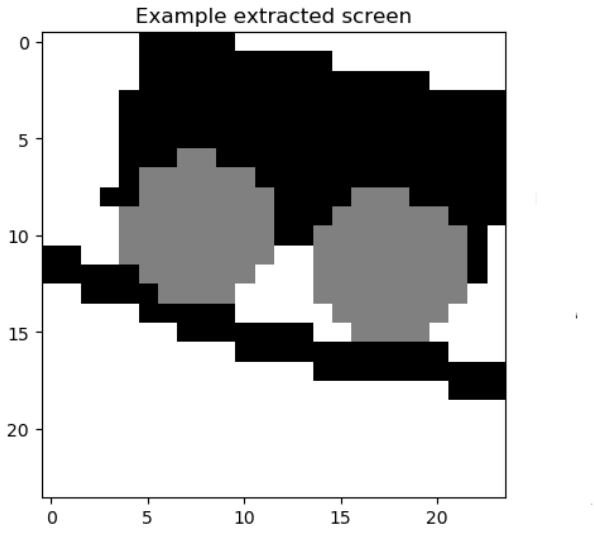

Figure 47: Croping the environment around the car

The window is following the car. To do this the position of the car is asked to the environment. Knowing the function that produces the hills, it is possible to know the position of the car on x and y. Then a square `[x-12:x+12,y-12:y+12]` is produced. This procedure can work relatively well, however it presents a major drawback. Indeed, by asking the position of the car to the environment, the task of learning from pixels only is not respected. Another method has to be found.

4.7.5 Processing of the image

Reducing the number of pixels in an image doesn't necessarily mean cropping the image, it can also be achieved by lowering the resolution of the image. This process is called downsampling. For example, a 400x600 image (the size of the MountainCar screen) downsampled 10 times gives a 40x60 image. This process has to be applied cautiously

to avoid losing too much information. Here a factor of 10 appears to be a good compromise.
See the illustration of the whole image processing in appendix B.

```
def downsample(img):
    return img[::10, ::10]
```

Listing 33: Downsampling

Moreover, as discussed before, an image is a 3-D matrix composed of pixels which are a combination of red, green and blue. A solution to decrease the dimension of the image to a 2D matrix is to operate a mean over the three colors' values. An image in gray scale is obtained . From a 40x60x3 image it becomes a 40x60x1 image.

```
def rgbtogray(img):
    return np.mean(img, axis=2).astype(np.uint8)
```

Listing 34: From rgb to gray

As the image has been reduced to a one-dimensional matrix of shape 40x60x1 it has be reshaped to 1x40x60 for Pytorch compliance.

Now it could be interesting to reduce the range of values in the matrix. A solution could be to divide all values by 255 in order to have only values between 0 and 1 (this can benefit to the network). A solution that appeared to be even better consists in having a binary matrix with only 0 and 1. It is observed that the pixels of the wheels of the car have a unique value (128). Therefore, it is possible to set this value to 1 and every other values to 0. A black background is obtained with only the wheels appearing in white.

```
def get_screen():
  screen = env.render(mode = 'rgb_array')
  processed_state = np.uint8(np.reshape(rgbtogray(downsample(screen)), (1, 40, 60))) #pytorch
    compatible shape
  processed_state[processed_state != 128] = 0
  processed_state[processed_state == 128] = 1
  return processed_state
```

Listing 35: Function responsible of processing the image.

Please note that the values of the image are assigned the data type `uint8` as it is the data type which demands the less amount of memory to be stored (8bits).

The whole process is summarized by the following diagram:

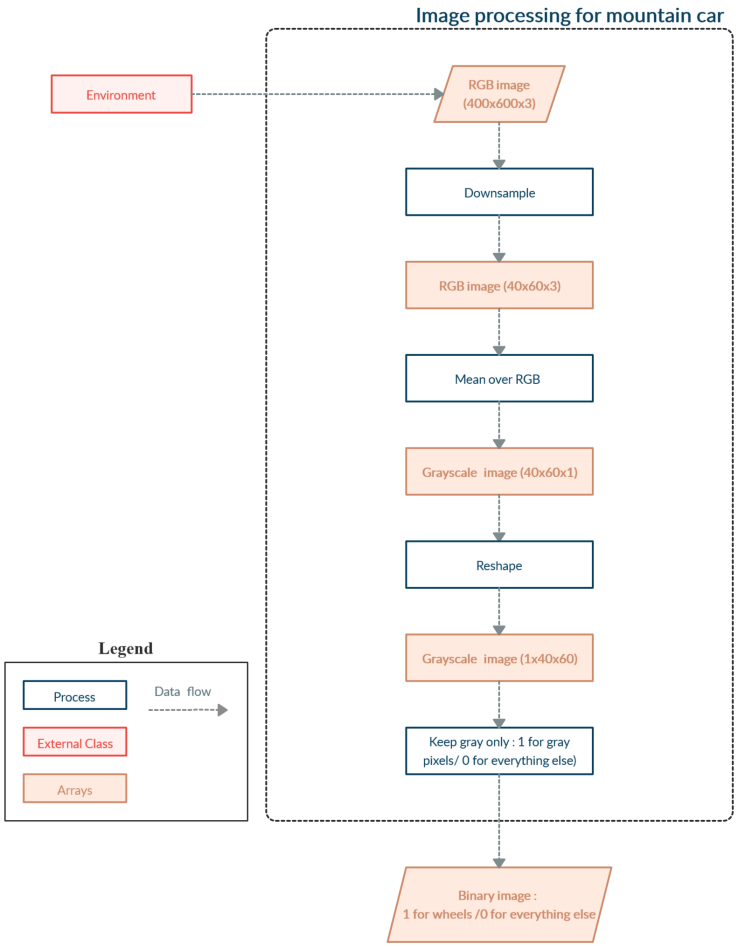

Figure 48: Image processing of the mountain car problem: method 2

Even if the running time improved a lot with the processing of the image, it remained really slow and it was difficult to see some results. Something completely different had to be tried.

4.7.6 subtracting the frames

In this part, instead of stacking frames, it is decided to subtract them two by two. The processing remains exactly the same as before. By subtracting the frames we make the strong assumptions that the result will be unique and will correspond to a specific combination of position and velocity. In other words, given four completely different states, subtracting one to another would give unique combination. It is done in the following manner:

```
for i in range(episodes):
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen #initilization


    done = False
    total_reward = 0
    while not done:
        step += 1
        action = agent.act(env, state, eps)
        _, reward, done, _ = env.step(action)
        total_reward += reward

        if step% 4 == 0:  #every 4 frames
            last_screen = current_screen
            current_screen = get_screen()

            if not done:
                next_state = current_screen + -1*last_screen
                if np.amax(next_state) == 0 : #avoid black screen
                    next_state = current_screen
            else:
                next_state = current_screen - current_screen
state = next_state
```

Listing 36: Subtracting images

This implementation is running much faster and shows some results. These results are far from a perfect policy however, it shows improvements. Concerning the hyperparameters, a learning rate of `5e-4` was used. It is interesting to observe that it is much smaller than one used previously without convolution layers. The other hyperparameters were as follows:

```
batch_size = 64
target_update = 100
gamma = 0.99
epsilon = 1 #at the beginning
decay_value = 0.99
```

The optimization of this algorithm might reside in parameters tuning, thing that couldn't been achieved due to a lack of time. The results are shown below:

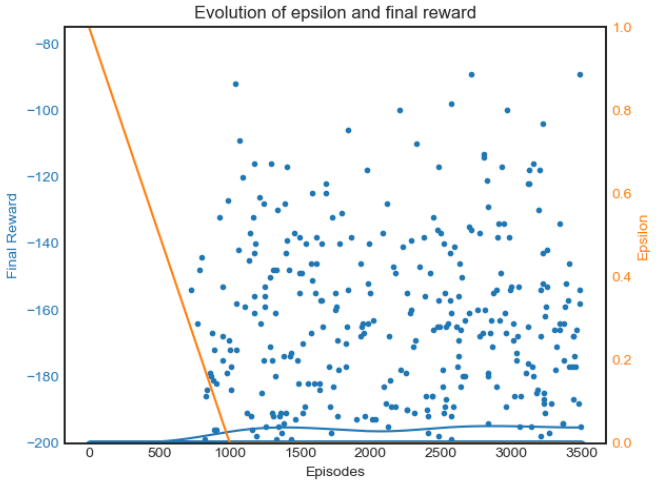

Figure 49: Performance of DQN on mountain car with pixels: method of the subtraction

4.8 Moving to Atari games

The final step of this project concerns the implementation of the DQN algorithm to the Atari environment. Especially it would be particularly interesting to write an algorithm that is not specific to a specific to a single game, therefore able to handle various games. Hence, the algorithm's performances will be tested on three different game : Pong, Breakout and MsPacman.

4.8.1 Atari's image processing

The Atari environment offers a display that quite similar between games. First of all, the screen size is always the same (210x160) with RGB colors. Also, the screen contains most of the time a section dedicated to the score and/or lives. Also the screen of an Atari contains borders, all the screen is not playable. These common points can be used to define a generic way to process an Atari game's screen.

Reminding that the convolutional layers are basically a square filter scanning the image with a predefined step and processing these subsections(see part 4.7.1), it can be deduced that convolutional layers are particularly efficient over square inputs. In addition, the execution with a GPU is optimized for square shapes. For these reasons, the image will be cropped in order to make a square. Therefore, the height of the screen images will be cut. Luckily the score is often located at the top or at the bottom. However, all the games cannot be cut in the same manner. Nevertheless, a square window of 160x160 works far a good amount of games, it could be considered as a default cropping size. In this case a strip is cut at the top and at the bottom of the image where there are often less useful information. In Pong and Breakout the parts of the image that can be removed are 34 pixels high. In this way, the croping part takes the form of : `frame = frame[34 :34 + 160, :160]` . Having a square screen of dimensions 160x160 makes things easy for the following as it can be downsampled by a factor of 2 (`frame = frame[::2 , ::2]`)to obtain a 80x80 image that can be feed the DQN.

However, if the goal is to write a generic deep Q-learner for Atari games, the image processing must be flexible to encompass different styles of environment. For example, the MsPacman's screen is not organised in the same way than Pong and Breakout. Indeed, the playble part of the screen begins really high compared to other games.

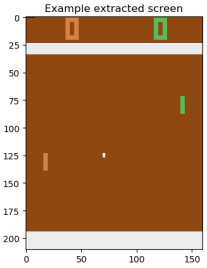

Figure 50: Pong's screen

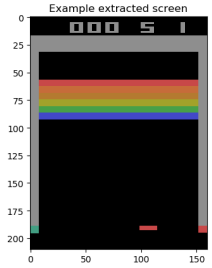

Figure 51: Breakout's screen

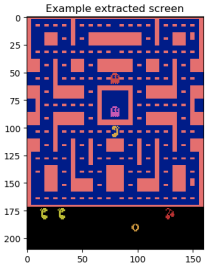

Figure 52: MsPacman's screen

By looking at pacman's screen, the image has to be cropped in the following way `frame = frame[2 :170, :160]` . The image's shape is therefore 168x160. It is not recommended to downsample more than twice the resolution of the image to avoid losing information. Thus after the downsampling the image must have a 84x84 shape. This downsampling is not as straightforward as the previous ones. The tool `cv2.resize()` allows to choose the shape of the output of the downsampling process.

To have the desired output we use `cv2.resize(frame, (84, 84), interpolation=cv2.INTER_AREA)`.
To make the algorithm adaptable for the different games, this command will be used for each of them. Indeed, downsampling to 84x84 instead of 80x80 won't be problematic.

As it has been done for MountainCar, the dimension of the image is also decreased to 1 by computing the mean value of the pixels over their RGB values. A grayscale image is obtained.

With the same logic as before, the data is normalised before being pass through the network. All the pixels value are divided by the max value of the matrix. Also they are transformed to data type np.float32 as it seems to be the data type the neural networks are the most efficient with. Into code : `np.array(frame).astype(np.float32) / np.amax(frame)`.

Finally, as usual the shape is transformed to respect Pytorch's convention. Into code :

```
1
2 def process(self, frame):
3     frame = frame[x:160 + x, :160] #x has to be chosen according to the game
4     frame = frame.mean(2)
5     frame = cv2.resize(frame, (self.width, self.height), interpolation=cv2.INTER_AREA)
6     frame = np.reshape(frame, [1, 84, 84])
7     return np.array(frame).astype(np.float32) / np.amax(frame)
```

Listing 37: Image processing of Atari games

Then the frames are also stacked by groupes of 4 as before.

It can be mentioned that other ways of processing the image were tested such as making all pixels black and white as it has been done in MountainCar.

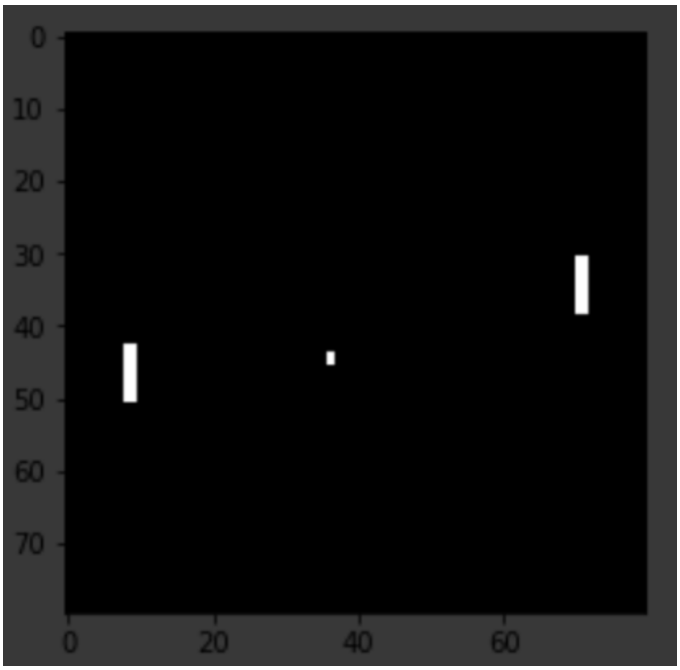

Figure 53: Pong processed black and white

However, this way of processing presupposes a knowledge of the pixel values and is not adaptable to other games.

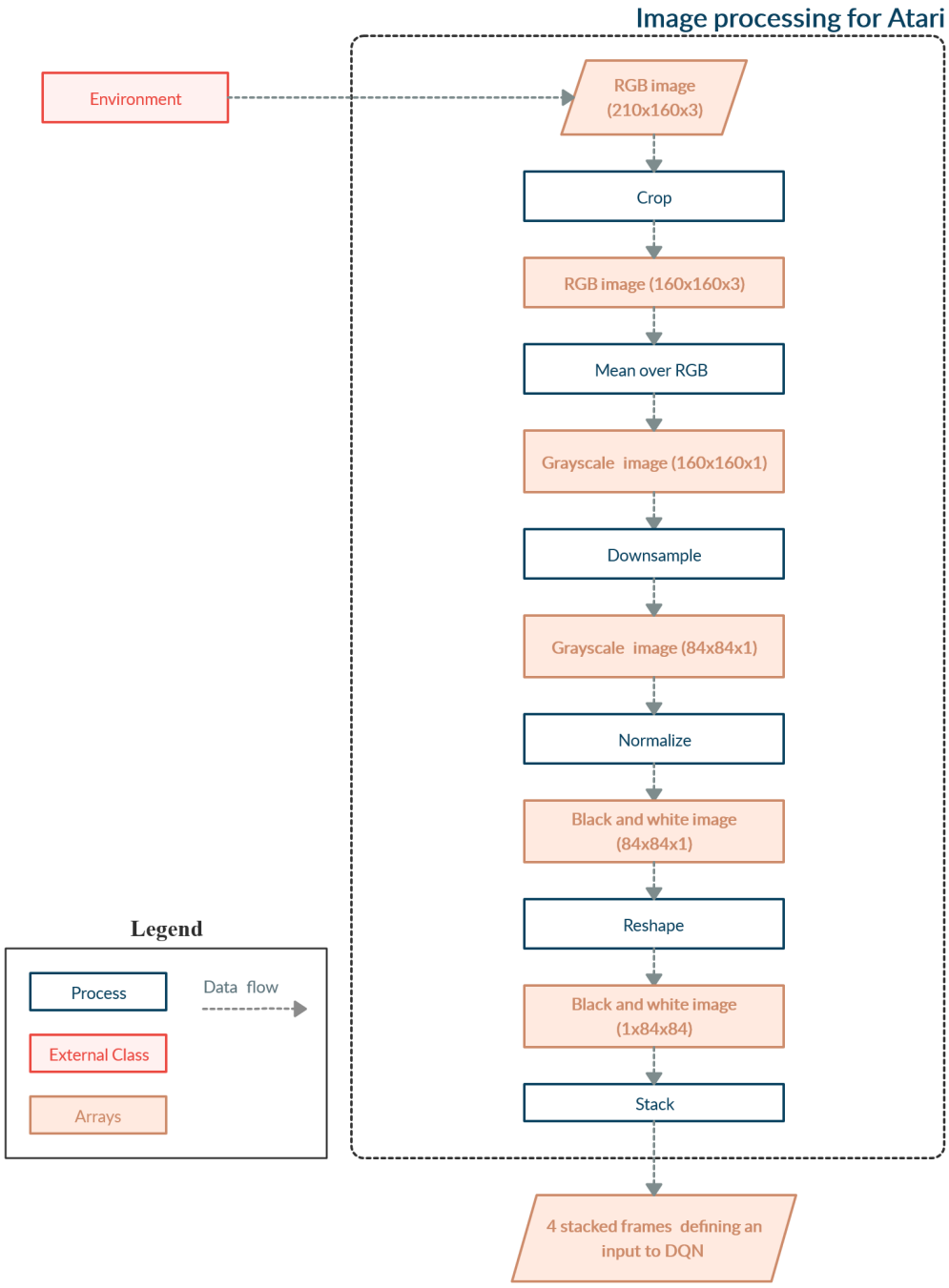

Figure 54: Image processing for breakout and pong

4.8.2 Modifications made on the DQN algorithm

Some modifications have been made on the DQN algorithm in order to adapt to the Atari environment. The Q-function for an Atari game has to be of higher order than the Q-function for MountainCar. To deal with this increasing complexity, the DQN has to contain more filters and also more neurons. Consequently, a new fully connected layer has to be added to make the transition to Q-values. The new DQN is the following :

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(state_dim[0], 32, kernel_size=8, stride=4, bias=False)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)

        self.fc1 = nn.Linear(3136, 512)
        self.fc2 = nn.Linear(512, action_dim)
        self.device = 'cuda'

    def forward (self, x):
        state = x.to(self.device).float()
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.fc1(x.view(x.size(0), -1)))
        q_values = self.fc2(x)
        return q_values
```

Listing 38: New DQN

The value `3136` was obtained with the equations of part 4.7.1.

Another modification has been in the training process, to change the computation of the Q-target. Indeed, double Q-learning was implemented. This method has been shown to improve the training process(6). Double Q-learning consists of evaluating the best action a' in a given `next_state` with the network and then approximating its Q-value with the target function. The reason behind this is that both networks have a noise . Here the action is chosen with the noise of the main network and the Q-value associated with this action is evaluated with the the target network which has its own noise. These two noises have a tendency to cancel each other. More mathematically, according to Van Hasselt (7): "the expectation of a maximum is greater than or equal to the maximum of an expectation".

Atari's wrappers Gym provides a very convenient framework that is called the `Wrapper` class. It directly inherits from the `Env` class. It can be found at github (8). They can also be found in the Annex. It allows modifying the way the agent interacts with the environment in a generic way. Some functions can be written in this class and are divided in three types, they can belong to: `ObservationWrapper` , `RewardWrapper` , `ActionsWrapper` which respectively interact with the `obs` , `rew` and `act` methods. For example, a function coding the epsilon-greedy policy could override the `act` method and the agent would take some random actions despite a predefined policy.

It is common practice to properly wrap the environment before the training process. Some function in the `atari_wrappers.py` file of OpenAI Baselines (see link above) are very interesting. As most of them will be used, this part will present them. Also, the image processing function will be implemented in it.

4.8.3 Random no-operations reset

The agent often start a game in the same initial state. Therefore, if it always performs the same first action at the same time, it will expose itself to a lower range of different game situations which won't benefit to the learning.

The class `NoopResetEnv` allows to randomise the initial behaviour of the agent by no starting to play for a random number of states (it is sampled between 0 to 30). This way the agent does not get used to a specific set of observations and is able to learn faster. This function is for example really useful for Breakout.

4.8.4 Fire on reset

Sometimes the agent has to take action "Fire" to begin the game. It correspond to the red button on Atari game pad. It is the case in Breakout where the action "Fire" throws the ball in the direction of the blocks. The agent could learn this behaviour, however it would slow the training process and it is much more convenient to use the class `FireResetEnv` to set the first action as "Fire" when it is in the `env.action_space`

4.8.5 Episodic Life

The use of the class `EpisodicLifeEnv` really improved the results of the algorithm. Indeed, it produces a `done` flag when a life is lost. This allows the rewards to better back propagate as the agent knows that losing a life is something bad. Even though, this class produces a `done` flag, it does not reset the environment which would be problematic as the agent would not be able to deploy strategies on multiple lives.

4.8.6 Max and skip env

The classical gym environments such as `Ponv-v0` or `Breakout-v0`, in other words if it has not "`NoFrameskip`" or "`Deterministic`" in its name, the action is repeatedly performed during n frames with n sampled from 2, 3, 4. To facilitate the learning process, the environment will be used with "`NoFrameskip`". Therefore, this class allows to skip frames at a uniform rate (the skip rate will be set as 4).

4.8.7 Warp frame

This class has been modified from the original one in order to implement the processing function that was written in part 4.8.1. It is done as follows :

```
class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env):
        """Warp frames to 84x84 as done in the Nature paper and later work."""
        gym.ObservationWrapper.__init__(self, env)
        self.width = 84
        self.height = 84
        self.observation_space = spaces.Box(low=0, high=255,
            shape=(1, self.height, self.width), dtype=np.uint8)

    def observation(self, frame):
        frame = frame[34: 194, :160]
        frame = frame.mean(2)
        frame = cv2.resize(frame, (self.width, self.height), interpolation=cv2.INTER_AREA)
        frame = np.reshape(frame, [1, 84, 84])
        return frame
```

Listing 39: New DQN

Clipping the rewards Clipping the reward results in imposing a value of either -1, 0, 1. In some environments such as MsPacman, the reward can take really high values and clipping and this has a great impact on the training process. In order to have a more stable and unified training process between games (which allows to keep relatively similar hyperparameters) the reward is clipped.

4.8.8 The Monitor class

The Monitor class is not in the atari-wrapper file. However it allowed to record the renders of the games as video and happened to be a useful tool.

4.8.9 Results

The implementation of the DQN showed improvements on each environment with minor modifications between them. The environment used were : `PongNoFrameskip-v4` , `BreakoutDeterministic-v4` and `MsPacmanNoFrameskip-v4` . The results will be analysed game by game but first, a few things can be noted. First the hyperparamters can be tuned for each game to obtain faster results. It is for example observed that Pong needs less exploration (an epsilon decaying over 30k frames is enough whereas it has to decay during 1M frames for MsPacman),than the two others to show good results. Also the learning rate can be bigger than for the two others. Overall training Pong requires less training time (about a million frames to have good results for pong while it can be of the order of 20 millions for the other). At the same time. MsPacman don't necessit to stack frames and the DQN is therefore able to process the images more quickly.

As MsPacman and Breakout require a lot of frames to be trained, and online GPUs were used, the training was often stopped. It was difficult to obtained a unified and continuous training time. To face this problem, the parameters were saved during each run and reloaded to continue training.

4.8.10 Pong

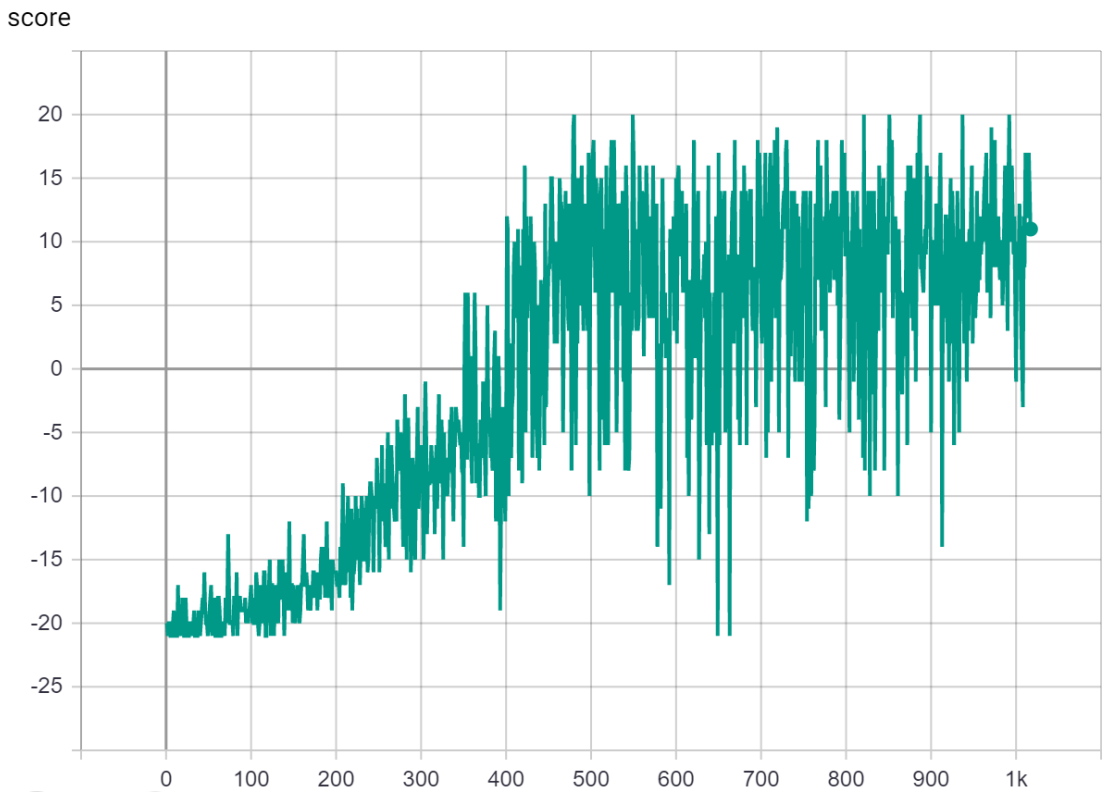

Figure 55: First Successful implementation of Pong

In pong a game when the reward is above 0. A game lasts 21 points, which means that a reward of 21 is equivalent to a game won 21-0. On the graph above the x axis is the number of episode. Reaching a good reward (more than 10) took approximately 1.5M frames and 3 hours of training.

To give an idea, a human average was made with the members of the group and it was about -12.

Improving the hyperparameters and optimizing the wrappers class, a much better performance could be achieved:

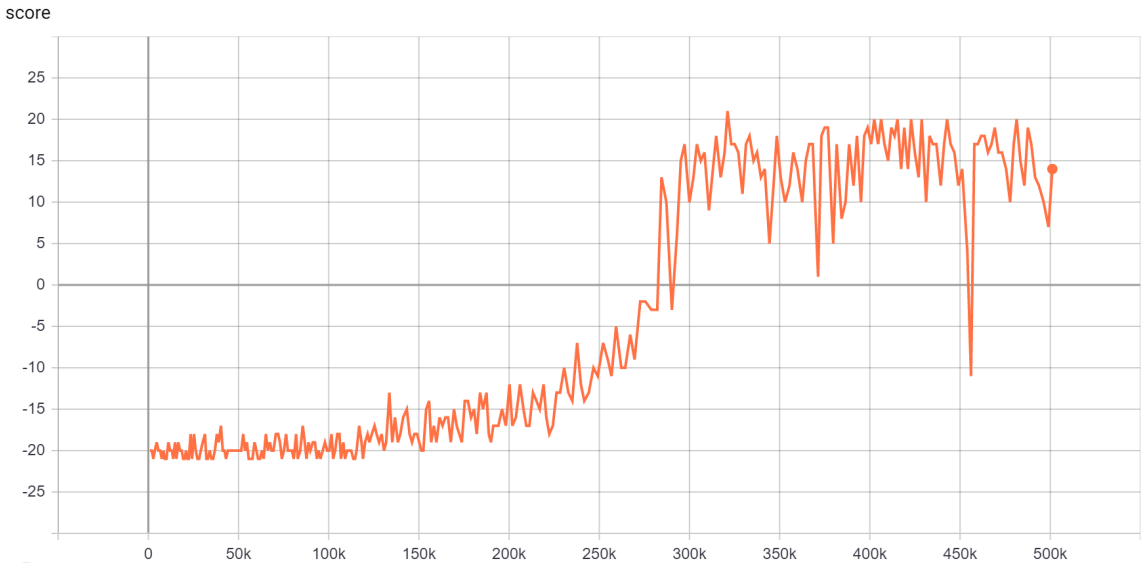

Figure 56: Improvement of the learning process

In this last version (the script can be found in the annex) the training time is only of an hour.

4.8.11 Breakout

Below is presented an example where the training was stopped and restarted.

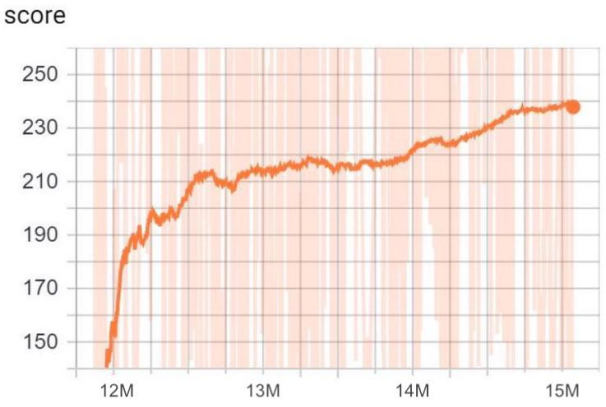

Figure 57: Improvements of breakout between frames 12M and 15M

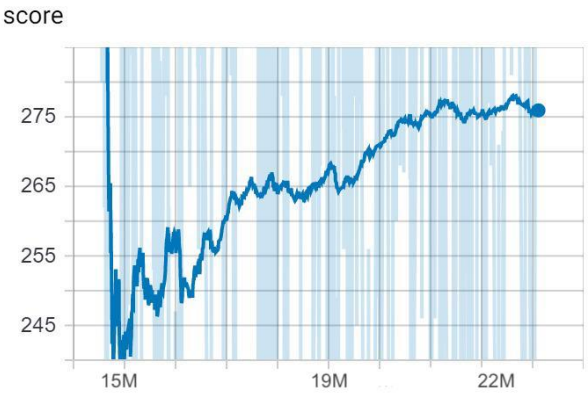

Figure 58: Improvements of breakout between frames 15M and 22M (training restarted)

With 40M of frames processed the mean score was around 320 which is not far from Deepmind's performance. However the agent was still improving and the full potential wasn't reached. Maximum values of 650 were achieved.

An interesting behaviour that has been observed is the fact the agent finds the strategy to dig a tunnel in order to destroy more blocks.

Figure 59

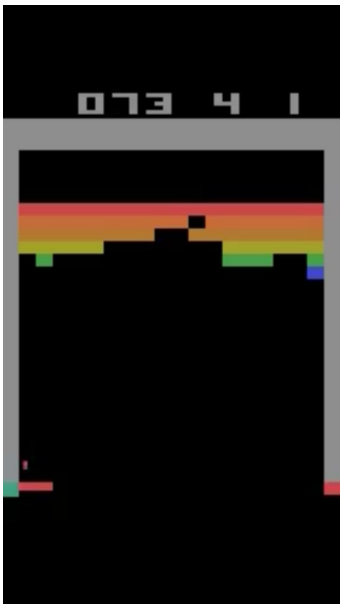

Figure 60

Figure 61

Figure 62

Something strange was observed during the training. The agent happened to be stuck in a loop, the ball was

always bouncing back in the same way without touching any blocks. After some time, the score and the ball disappeared and the screen started to flash in different colors.

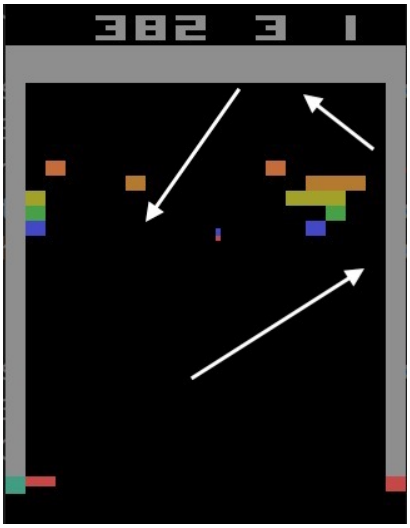

Figure 63: Path of the ball

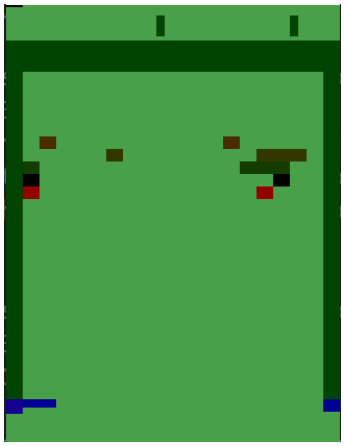

Figure 64: Screen flashing

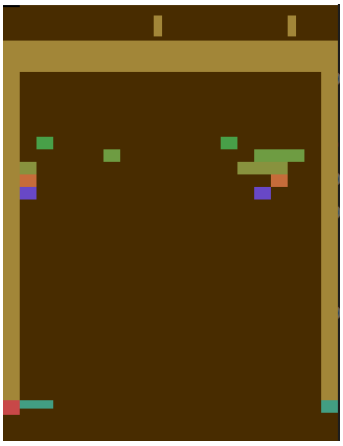

Figure 65: Screen flashing

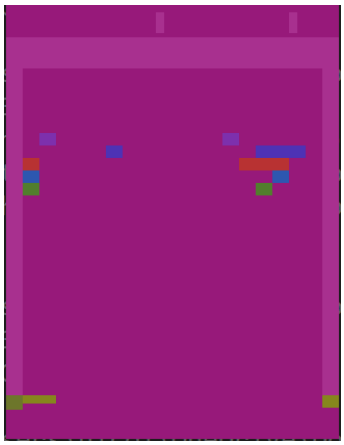

Figure 66: Screen flashing

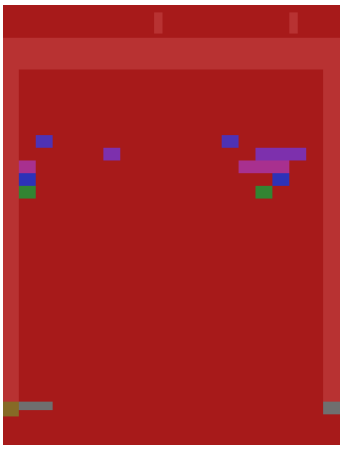

Figure 67: Screen flashing

Is it a bug of the game or an easter egg that the agent would have found?

4.8.12 MsPacman

MsPacman wasn't trained to his full potential. However it showed a good and relatively linear progression which is surprising for a such non linear game

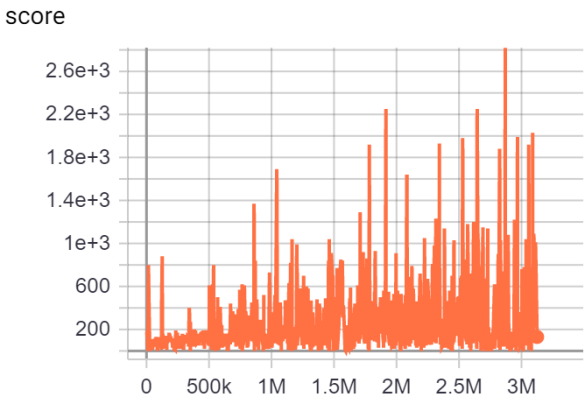

Figure 68: Improvements over the first 3M frames (not smoothed)

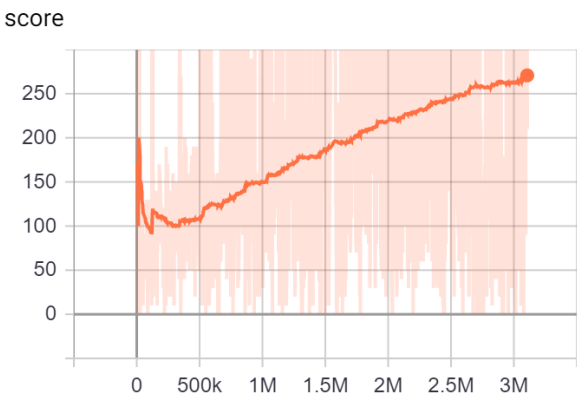

Figure 69: Improvements over the first 3M frames (smoothed)

N.B. The rewards shown on the graph are the rewards accumulated during a single life. Pacman has three lifes. After a longer training scores of around 3k could be regularly achieved.

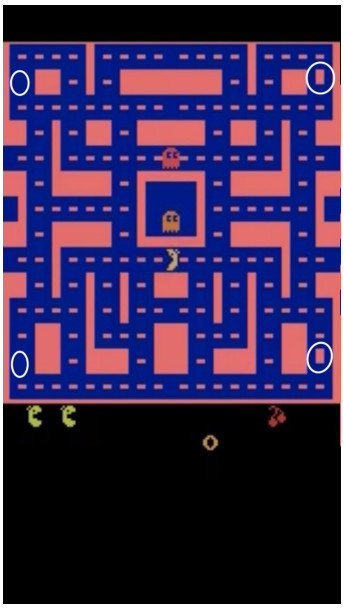

Figure 70: Locations of the four points where Pacman eats the superfood

It is interesting to see that Pacman moves directly to the four corners where it can find the superfood allowing it to eat ghosts. In fact Pacam was trained with a clipped reward (not on the graphs) and therefore tries to maximize its living time, Eating superfood allows him to be protected from the ghosts.

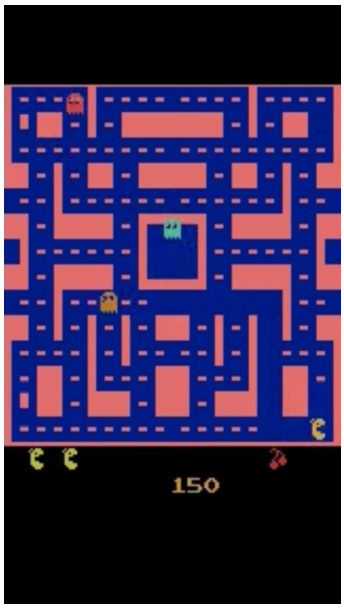

Figure 71: First superfood

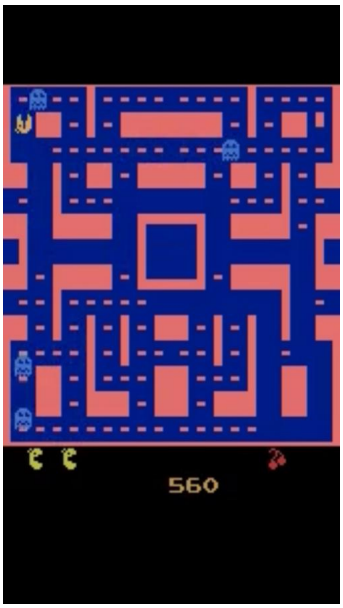

Figure 72: Second superfood

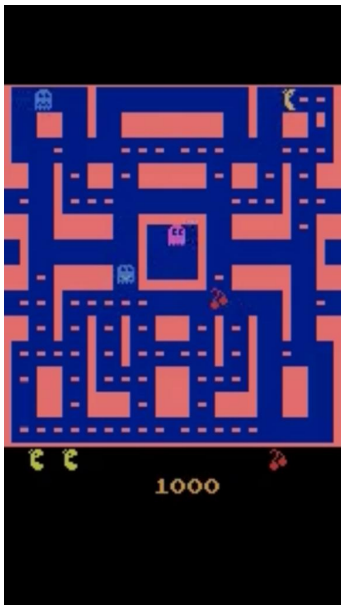

Figure 73: Third superfood

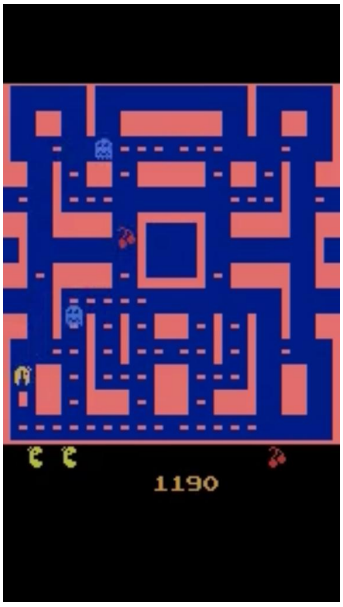

Figure 74: Fourth superfood

5 Conclusion

To conclude, the objective throughout this project was to learn and understand reinforcement learning in its whole and discover neural networks in other to apply them to games, from simple ones to Atari games.
Several reinforcement learning techniques were explored during this project. First the theory has been widely explored and then translated into python code in order to win many games. Each implemented method worked well to reach the objective but each showing slightly different results. Finally, the neural networks made it possible to go much further by successfully solving more complicated games such as pong, breakout and pacman.
A future goal we could envision would be to apply all of our new knowledge to real cases of automated mechanical systems.

6 Acknowledgements

We would like to thank a thousand times all those who accompanied us during this project. Thank you for managing the digital transition so well. The weekly reports and meetings allowed us to move forward with a good pace thanks to your feedback. Thank you for all your suggestions, ideas and support. Thank you for always encouraging us in what we wanted to do. It has been a pleasure to work with you, we hope you are also happy with us and satisfied with our work.

We hope you will find a pleasant new occupation for your Monday afternoons !

Looking forward to seeing you in person when we will able to.

7 Appendix

A Pseudocodes

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 75: Q-learning: An off-policy TD control algorithm (2).

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
     $\Delta \leftarrow 0$
     For each $s \in \mathcal{S}$:
       $v \leftarrow V(s)$
       $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
     $a \leftarrow \pi(s)$
     $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
     If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Figure 76: Policy iteration algorithm (2).

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 77: SARSA: an on-policy TD control algorithm (2).

B Image processing

B.1 Mountain car

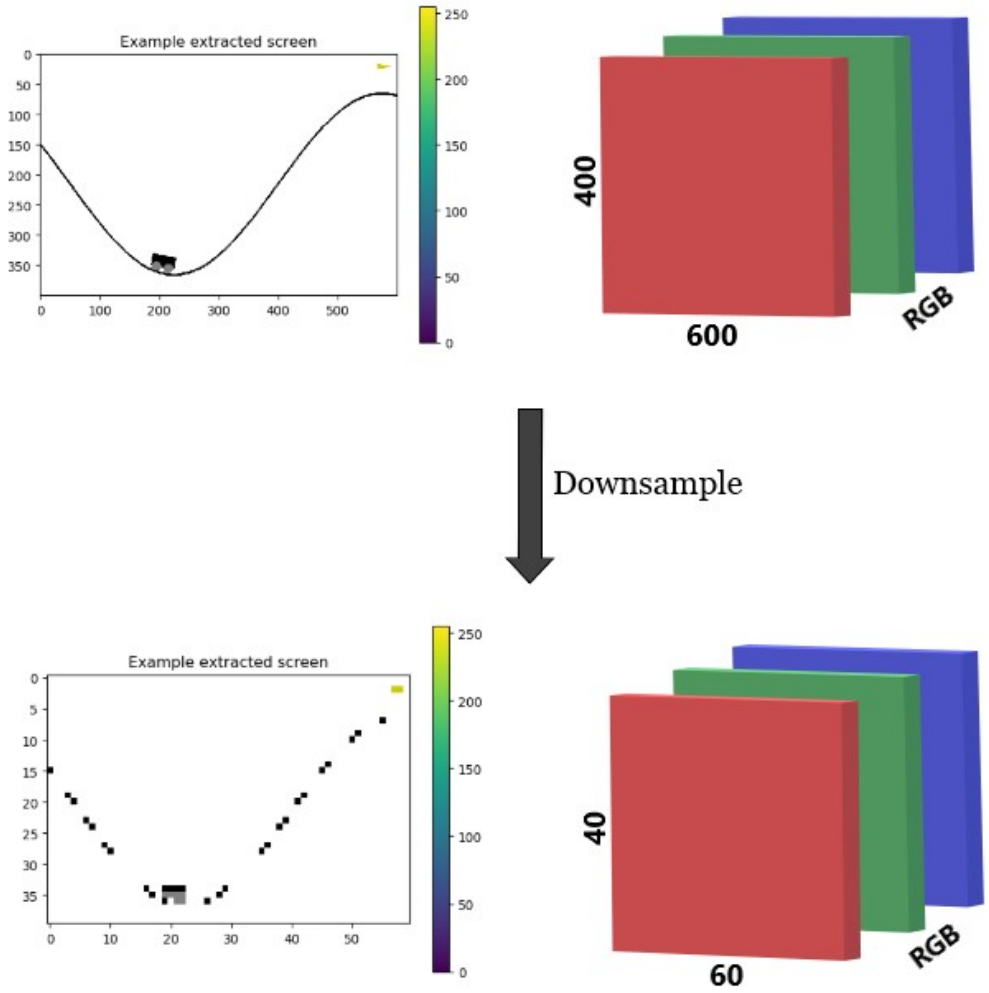

Figure 78: Step 1

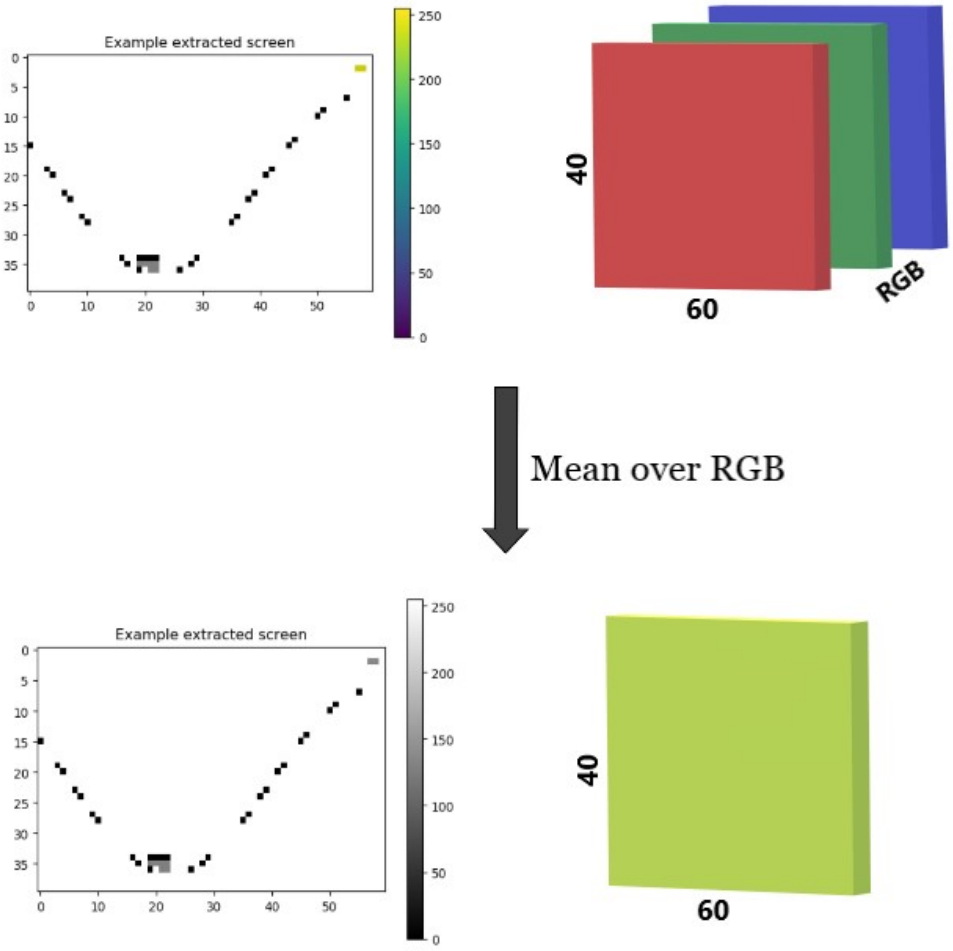

Figure 79: Step 1

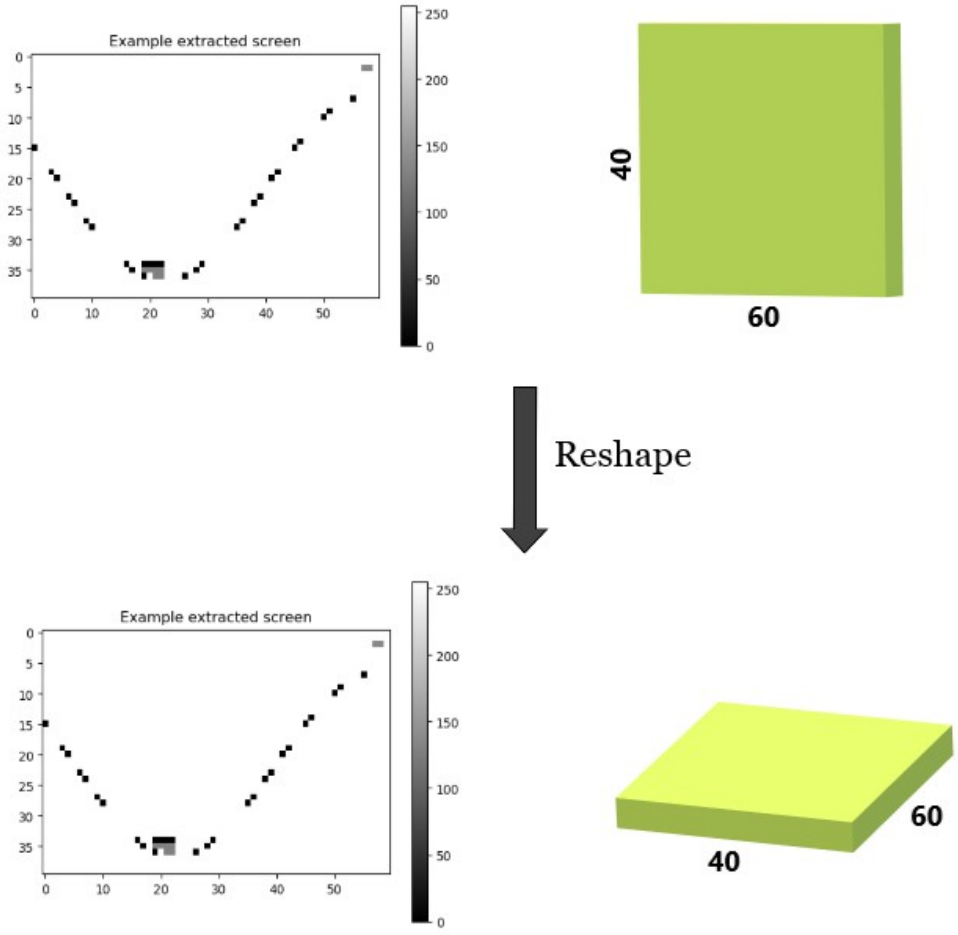

Figure 80: Step 1

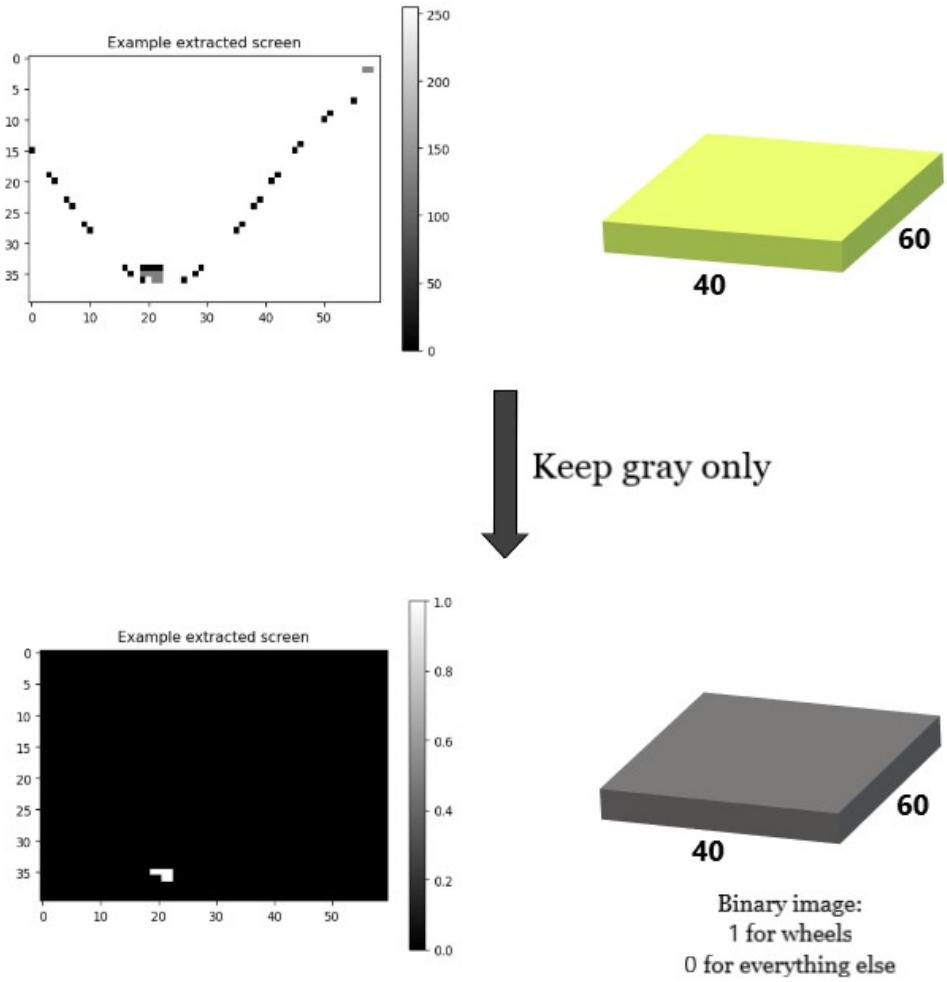

Figure 81: Step 1

B.2 Breakout and pong

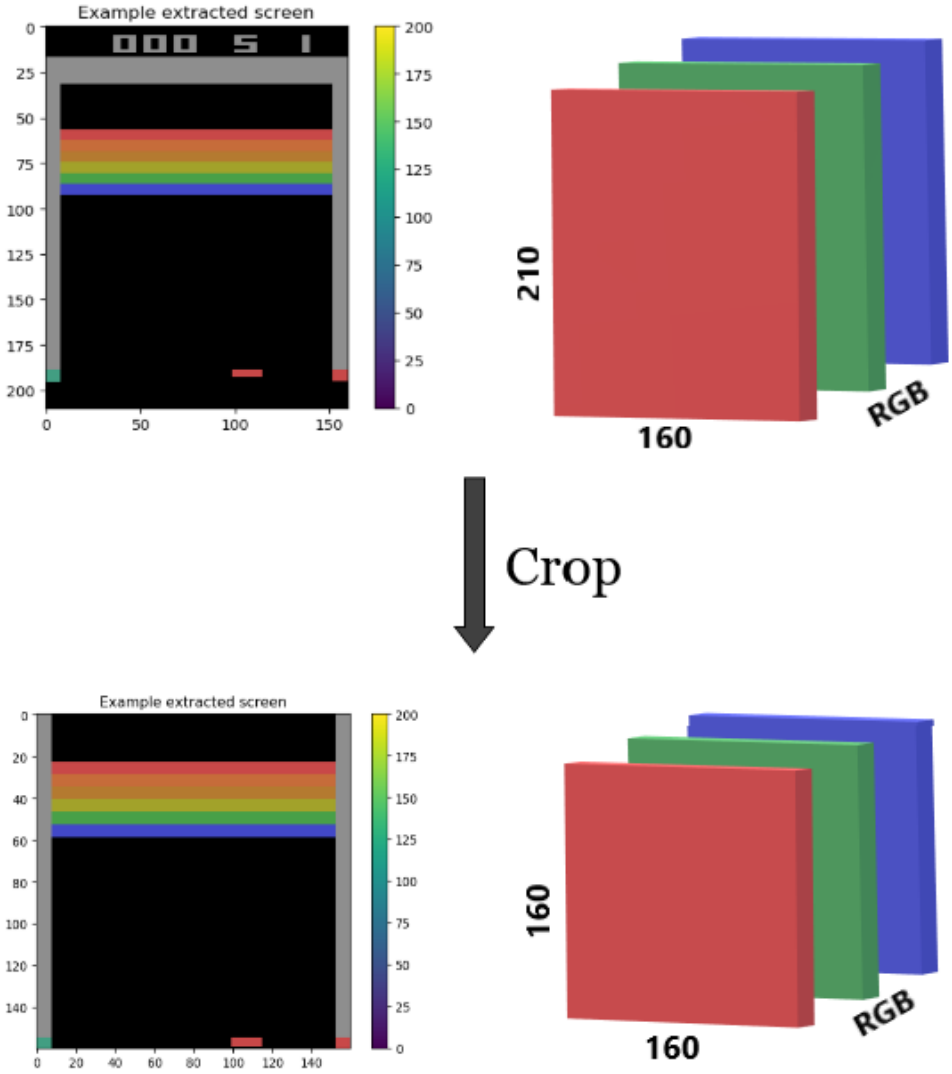

Figure 82: Step 1

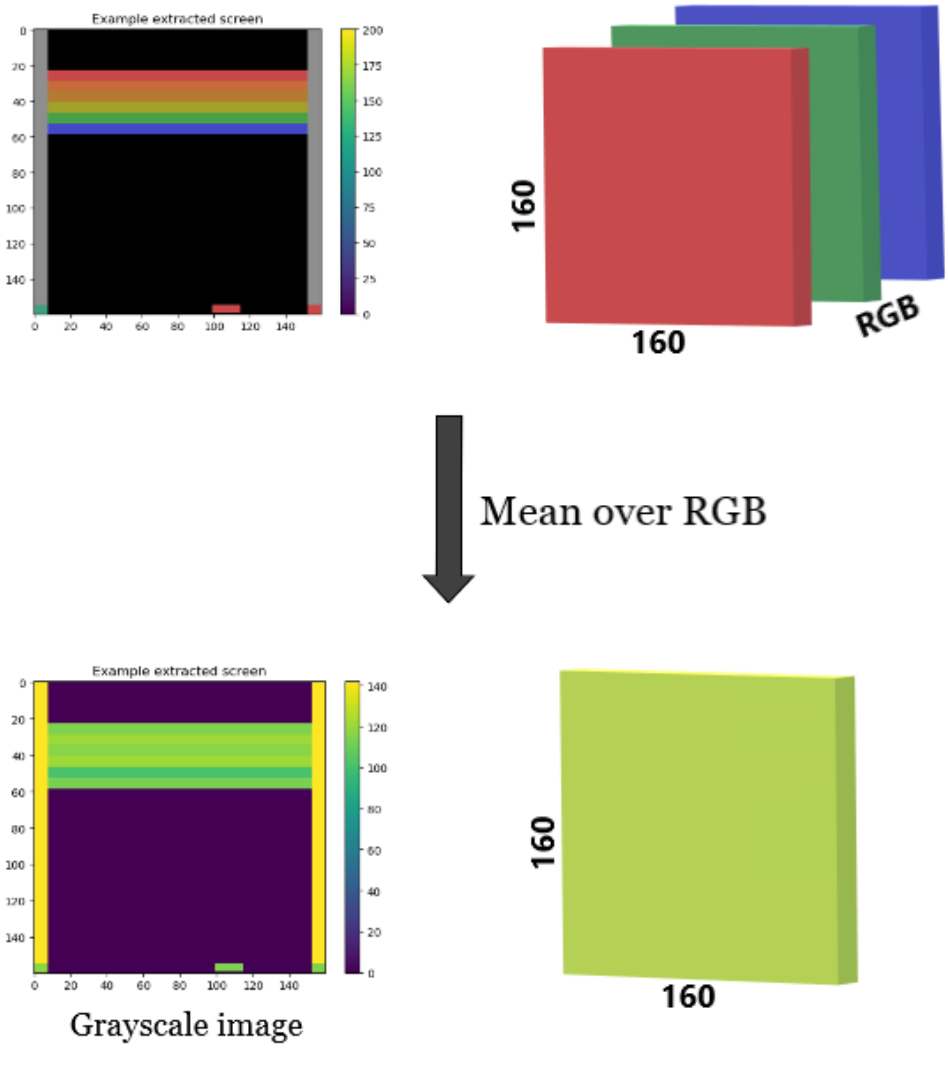

Figure 83: Step 2

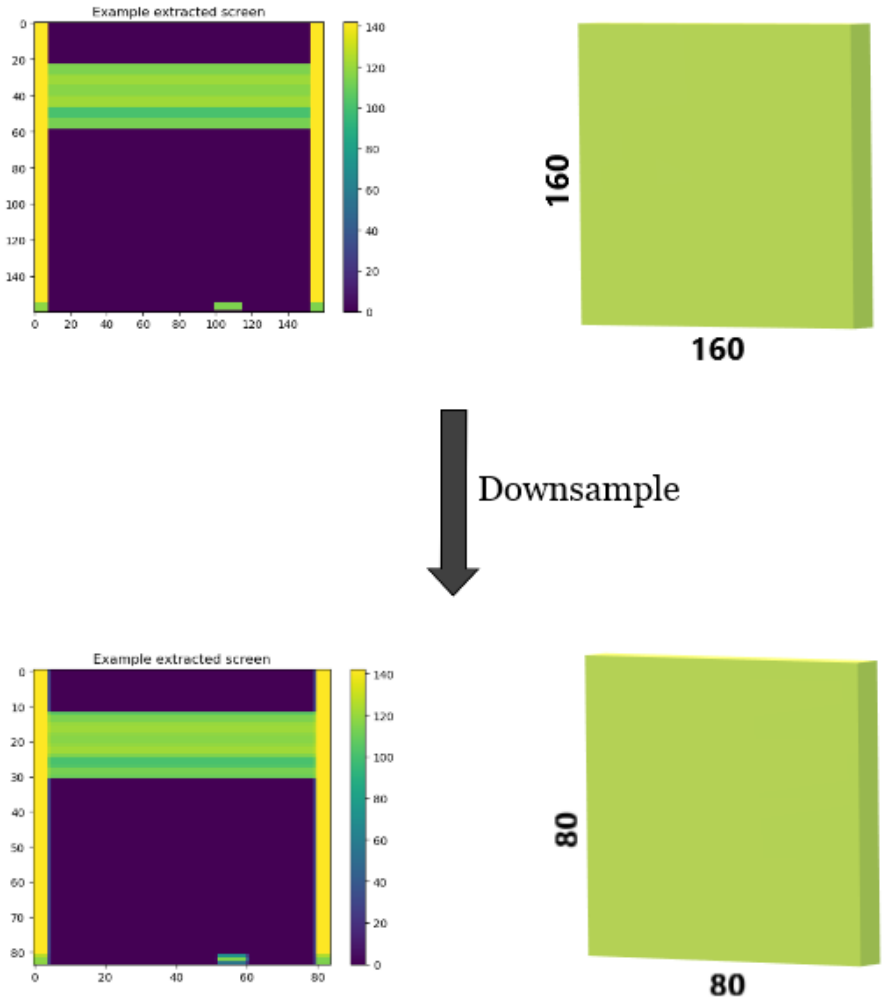

Figure 84: Step 3

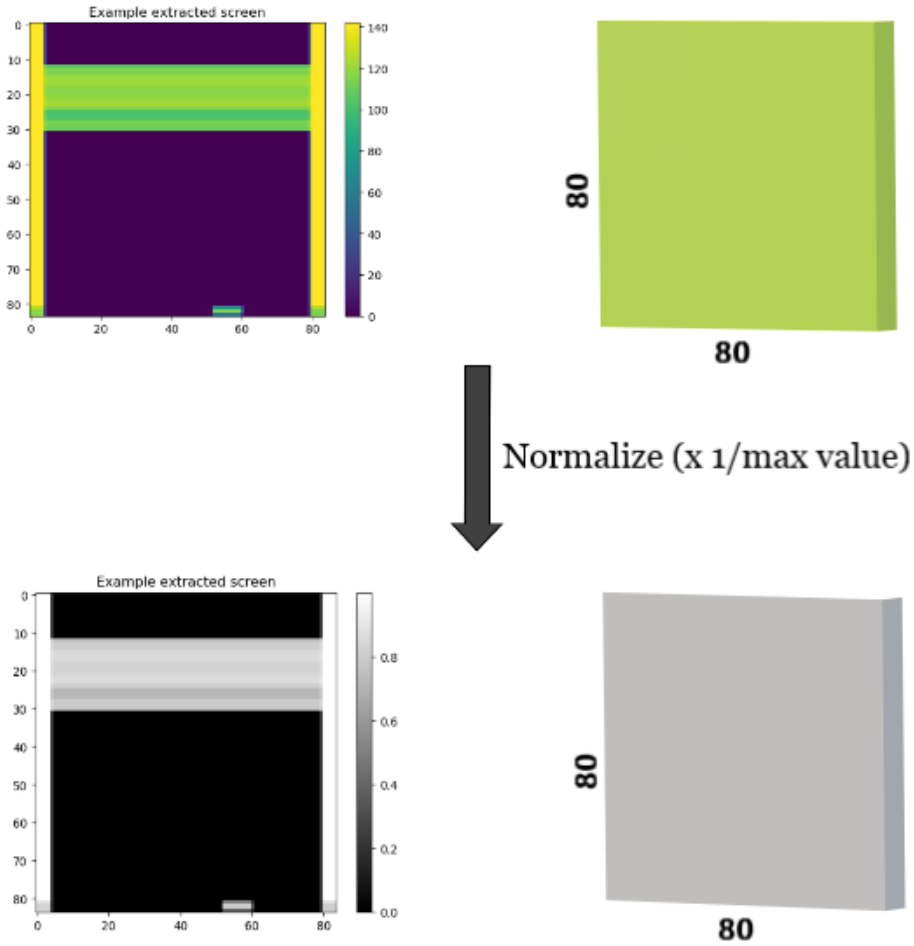

Figure 85: Step 4

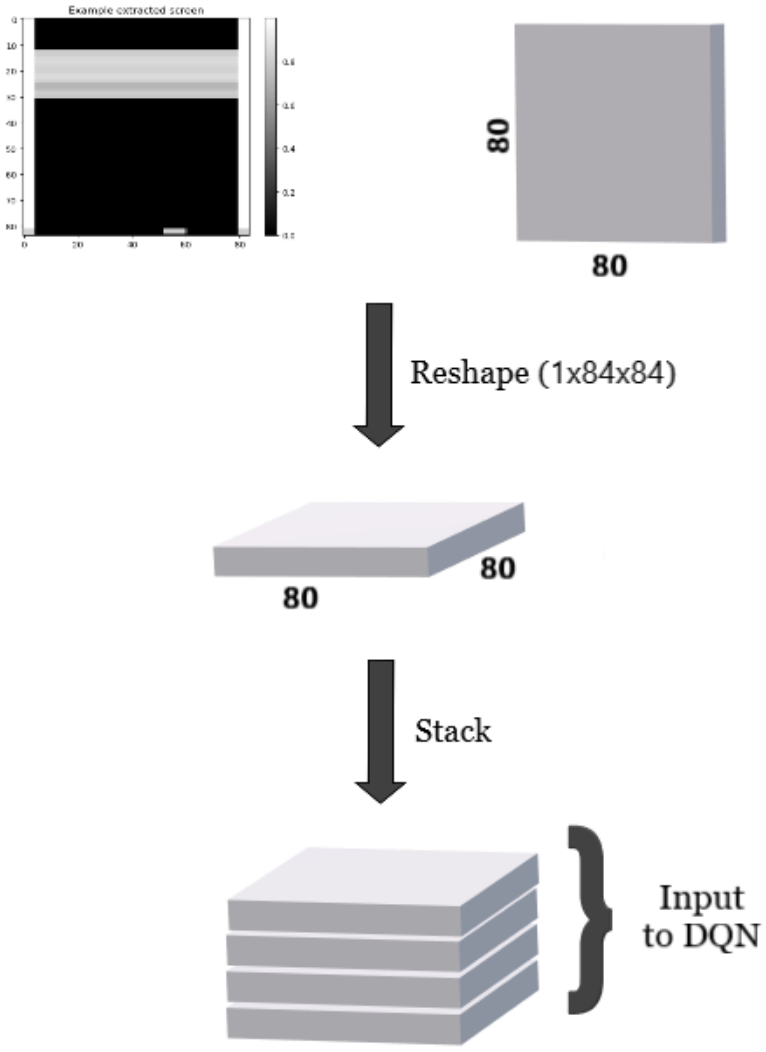

Figure 86: Step 5

C Functions taken from the internet

These two functions were used to plot the trend for SARSA-mountaincar and DQN-mountaincar-pixels.

```

# Defining the bell shaped kernel function - used for plotting later on
def kernel_function(xi, x0, tau=.005):
    return np.exp(- (xi - x0) ** 2 / (2 * tau))

def lowess_bell_shape_kern(x, y, tau=.005):
    """lowess_bell_shape_kern(x, y, tau = .005) -> yest
    Locally weighted regression: fits a nonparametric regression curve to a scatterplot.
    The arrays x and y contain an equal number of elements; each pair
    (x[i], y[i]) defines a data point in the scatterplot. The function returns
    the estimated (smooth) values of y.
    The kernel function is the bell shaped function with parameter tau. Larger tau will result in a
    smoother curve.
    """
    m = len(x)
    yest = np.zeros(m)

    # Initializing all weights from the bell shape kernel function
    w = np.array([np.exp(- (x - x[i]) ** 2 / (2 * tau)) for i in range(m)])

    # Looping through all x-points
    for i in range(m):
        weights = w[:, i]
        b = np.array([np.sum(weights * y), np.sum(weights * y * x)])
        A = np.array([[np.sum(weights), np.sum(weights * x)],
                      [np.sum(weights * x), np.sum(weights * x * x)]])
        theta = linalg.solve(A, b)
        yest[i] = theta[0] + theta[1] * x[i]

    return yest
```

Listing 40: Locally weighted linear regression (4)

D Codes

D.1 Q-learning for mountain car

```
import gym
import numpy as np

env=gym.make("MountainCar-v0")
env.reset()

LEARNING_RATE = 0.1
DISCOUNT = 0.95
EPISODES = 20000

DISCRETE_OS_SIZE = [20] * len(env.observation_space.high)
discrete_os_win_size = (env.observation_space.high - env.observation_space.low)/DISCRETE_OS_SIZE


epsilon = 0.5 #hasard entre 0 et 1, a quel point il explore
START_EPSILON_DECAYING = 1
END_EPSILON_DECAYING = EPISODES // 2 #get an integer
epsilon_decay_value = epsilon/(END_EPSILON_DECAYING - START_EPSILON_DECAYING)



q_table = np.random.uniform(low=-2, high=0, size=(DISCRETE_OS_SIZE + [env.action_space.n]))


def get_discrete_state(state):
  discrete_state = (state - env.observation_space.low) / discrete_os_win_size
  return tuple(discrete_state.astype(np.int)) #we use this tuple to look in the 3 Q-values which is
     good for the action


for episode in range(EPISODES):

  discrete_state = get_discrete_state(env.reset())
  done=False
  while not done:
    if np.random.random() > epsilon:
      action = np.argmax(q_table[discrete_state]) #action from Q table
    else:
      action = np.random.randint(0, env.action_space.n) #random

    new_state, reward, done, _ = env.step(action)

    new_discrete_state = get_discrete_state(new_state)

    if not done:
      max_future_q = np.max(q_table[new_discrete_state]) #max Q value in step
      current_q = q_table[discrete_state + (action, )] #current Q value

      new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE * (reward + DISCOUNT * max_future_q)
      q_table[discrete_state+(action, )] = new_q #updatee Q table

    elif new_state[0] >= env.goal_position:

      print(f"Gagn     l' pisode  {episode}")
      np.save(f"qtables/{episode}-qtable.npy", q_table)
      q_table[discrete_state + (action, )] = 0
```

```
      break

    discrete_state = new_discrete_state

    print(episode)

  if END_EPSILON_DECAYING >= episode >= START_EPSILON_DECAYING:
    epsilon -= epsilon_decay_value

env.close()
```

Listing 41: Q-learning on mountain car

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np
from matplotlib import cm
from numpy import linalg
x=np.linspace(0,20,20)
y=np.linspace(0,20,20)

fig = plt.figure(figsize=(20, 20))

for i in range(8990, 9000, 10):
    print(i)
    ax1 = fig.add_subplot(311, projection='3d')
    ax2 = fig.add_subplot(312, projection='3d')
    ax3 = fig.add_subplot(313,projection='3d')
    q_table = np.load(f"qtables/{i}-qtable.npy")
    a,b,c = np.dsplit(q_table, 3)
    z1=np.amin(a)
    z2=np.amin(b)
    z3=np.amin(c)
    z4=np.amax(a)
    z5=np.amax(b)
    z6=np.amax(c)
    a.shape=(20, 20)
    b.shape=(20, 20)
    c.shape=(20, 20)
    ind1 = np.unravel_index(np.argmin(a, axis=None), a.shape)
    ind2 = np.unravel_index(np.argmin(b, axis=None), b.shape)
    ind3 = np.unravel_index(np.argmin(c, axis=None), c.shape)
    ind4 = np.unravel_index(np.argmax(a, axis=None), a.shape)
    ind5 = np.unravel_index(np.argmax(b, axis=None), b.shape)
    ind6 = np.unravel_index(np.argmax(c, axis=None), c.shape)

    X, Y = np.meshgrid(x, y, copy=False)
    X=X.flatten()
    Y=Y.flatten()
    A = np.array([X*0+1, X, Y, X**2, X**2*Y, X**2*Y**2, Y**2, X*Y**2, X*Y]).T
    B=a.flatten()
    C=b.flatten()
    D=c.flatten()
    d, r, rank, s = np.linalg.lstsq(A, B, rcond=None)
    e, r, rank, s = np.linalg.lstsq(A, C, rcond=None)
    f, r, rank, s = np.linalg.lstsq(A, D, rcond=None)

    #def poly2Dreco(X, Y, c):
        #return (c[0] + X*c[1] + Y*c[2] + X**2*c[3] + X*2*Y*c[4] + X**2*Y**2*c[5] + Y**2*c[6] + X*Y
    **2*c[7] + X*Y*c[8])


    #zip = poly2Dreco(X, Y, coeff)
    x1=np.linspace(0,20,20)
    y1=np.linspace(0,20,20)
    X1, Y1= np.meshgrid(x1, y1)

    z = d[0] + X1*d[1] + Y1*d[2] + X1**2*d[3] + X1**2*Y1*d[4] + X1**2*Y1**2*d[5] + Y1**2*d[6] + X1*
    Y1**2*d[7] + X1*Y1*d[8],
    ax1.scatter(X, Y, a, color='r')
    ax2.scatter(X1, Y1, z, color='r', marker='x')

```

```
    ax1.set_title("Q values after 900 episodes")
    ax2.set_title("approx poly : %f + %fx + %fy + %fx^2 + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy
    " %(d[0], d[1], d[2], d[3], d[4], d[5], d[6], d[7], d[8]))

    plt.show()

    '''ax1.scatter(X, Y, a, color='r')
    ax2.scatter(X, Y, b, color='b')
    ax3.scatter(X, Y, c, color='black')

    ax1.set_title("Action 0 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %fy + %fx^2
    + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z1, str(ind1), z4, str(ind4), d[0], d[1], d[2],
    d[3], d[4], d[5], d[6], d[7], d[8]))
    ax1.set_xlabel("position")
    ax1.set_ylabel("velocity")
    ax1.set_zlabel("q_value")
    ax2.set_title("Action 1 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %fy + %fx^2
    + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z2, str(ind2), z5, str(ind5), e[0], e[1], e[2],
    e[3], e[4], e[5], e[6], e[7], e[8]))
    ax2.set_xlabel("position")
    ax2.set_ylabel("velocity")
    ax2.set_zlabel("q_value")
    ax3.set_title("Action 2 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %fy + %fx^2
    + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z3, str(ind3), z6, str(ind6), f[0], f[1], f[2],
    f[3], f[4], f[5], f[6], f[7], f[8]))
    ax3.set_xlabel("position")
    ax3.set_ylabel("velocity")
    ax3.set_zlabel("q_value")
    if (i >= 570):
        ax1.set_title("WON!! Action 0 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %
    fy + %fx^2 + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z1, str(ind1), z4, str(ind4), d[0], d
    [1], d[2], d[3], d[4], d[5], d[6], d[7], d[8]))
        ax2.set_title("WON!! Action 1 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %
    fy + %fx^2 + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z2, str(ind2), z5, str(ind5), e[0], e
    [1], e[2], e[3], e[4], e[5], e[6], e[7], e[8]))
        ax3.set_title("WON!! Action 2 min = %f at %s; max = %f at %s \n approx poly : %f + %fx + %
    fy + %fx^2 + %fx^2y + %fx^2y^2 + %fy^2 + %fxy^2 + %fxy" %(z3, str(ind3), z6, str(ind6), f[0], f
    [1], f[2], f[3], f[4], f[5], f[6], f[7], f[8]))

    plt.savefig(f"qtable_charts/{i}.png")
    plt.clf()'''

    '''def polyfit2d(x, y, z, kx=3, ky=3, order = None):
        x, y = np.meshgrid(x, y)
        coeffs = np.ones((kx+1, ky+1))
        u = np.zeros((coeffs.size, x.size))
        #fitted_surf = np.polynomial.polynomial.polyval2d(x, y, soln.reshape((kx+1, ky+1)))
        #plt.matshow(fitted_surf)

        for index, (j, i) in enumerate (np.ndindex(coeffs.shape)):
            if order is not None and i + j > order:
                arr = np.zeros_like(x)
            else:
                arr = coeffs[i, j] * x* i * y**j
            u[index] = arr.flatten()

        return np.linalg.lstsq(u.T, np.ravel(z), rcond=None)

    soln, residuals, rank, s = polyfit2d(x1, y1, a)

```

```
    fitted_surf = np.polynomial.polynomial.polygrid3d(0, x1, y1, soln.reshape((3+1, 3+1)),)
    plt.matshow(fitted_surf)
    plt.show()'''

'''
#style.use('axes3d.grid')


def get_q_color(value, vals):
    if value == max(vals):
        return "green", 1.0
    else:
        return "red", 1.0

for x, x_vals in enumerate(q_table):
        for y, y_vals in enumerate(x_vals):
            ax1.scatter(x, y, y_vals[0], color='r')
            ax2.scatter(x, y, y_vals[1], color='b')
            ax3.scatter(x, y, y_vals[2], color='black')

            ax1.set_title("Action 0 min = %f" %z1)
            ax1.set_xlabel("position")
            ax1.set_ylabel("velocity")
            ax1.set_zlabel("q_value")
            ax2.set_title("Action 1 min = %f" %z2)
            ax2.set_xlabel("position")
            ax2.set_ylabel("velocity")
            ax2.set_zlabel("q_value")
            ax3.set_title("Action 2 min = %f" %z3)
            ax3.set_xlabel("position")
            ax3.set_ylabel("velocity")
            ax3.set_zlabel("q_value")

    plt.savefig(f"qtable_charts/{i}.png")
    plt.clf()
#plt.grid(True)'''
#plt.show()

'''fig = plt.figure(figsize=(12, 9))
ax1 = fig.add_subplot(111, projection='3d')
#ax2 = fig.add_subplot(312, projection='3d')
#ax3 = fig.add_subplot(313, projection='3d')
i=537
q_table = np.load(f"qtables/{i}-qtable.npy")
#print(q_table)


for x, x_vals in enumerate(q_table):
    for y, y_vals in enumerate(x_vals):
        #array = np.array([y_vals[0]])

        print(x_vals[0])
        #ax1.plot_surface(x, y, x_vals, color='r')
        #ax2.scatter(x, y, c=get_q_color(y_vals[1], y_vals)[0], marker=",", alpha=get_q_color(
    y_vals[1], y_vals)[1])
        #ax3.scatter(x, y, c=get_q_color(y_vals[2], y_vals)[0], marker=",", alpha=get_q_color(
    y_vals[2], y_vals)[1])

#plt.show()'''
```

Listing 42: Display and save Q-learning results

D.2 Policy iteration on grid world

Note that the environment that is used (3) already implements the Q-learning. The only part that has been added is the policy iteration one. Set the variable "algo" to 1, to run the code with policy iteration.

```
import numpy as np

# global variables
BOARD_ROWS = 3
BOARD_COLS = 4
WIN_STATE = (2, 3)
LOSE_STATE = (1, 1)
START = (0, 0)
DETERMINISTIC = True
discount_factor=0.9 # policy evaluation is not converging for discount factor =1


                  ####### Choose Algo : [0 for q-learning], [1 for policy iteration] #######
algo=1


class State:
    def __init__(self, state=START):
        self.board = np.zeros([BOARD_ROWS, BOARD_COLS])
        self.board[1, 1] = -1
        self.state = state
        self.isEnd = False
        self.determine = DETERMINISTIC      # No noise


    def giveReward(self): #Reward protocol
        if self.state == WIN_STATE:
            return 1
        elif self.state == LOSE_STATE:

            return -1
        else:
            return -0.1

    def giveNxtReward(self,nxtpos): # Useful for policy iteration algo
        if nxtpos == WIN_STATE:
            return 10
        elif nxtpos == LOSE_STATE:
            return -10
        else:
            return -0.1

    def isEndFunc(self):
        if (self.state == WIN_STATE) or (self.state == LOSE_STATE):
            self.isEnd = True

    def nxtPosition(self, action):

        if self.determine:
            if action == 'u':
                nxtState = (self.state[0] - 1, self.state[1])
            elif action == 'd':
                nxtState = (self.state[0] + 1, self.state[1])
            elif action == 'l':
```

```
                nxtState = (self.state[0], self.state[1] - 1)
            else:
                nxtState = (self.state[0], self.state[1] + 1)

            if (nxtState[0] >= 0) and (nxtState[0] <= BOARD_ROWS-1):
                if (nxtState[1] >= 0) and (nxtState[1] <= BOARD_COLS-1):
                    #if nxtState != (1, 1): Don't bump into wall
                    return nxtState
            return self.state

class Agent:

    def __init__(self):
        self.states = []
        self.actions = ['u', 'd', 'l', 'r']
        self.State = State()
        self.lr = 0.2
        self.exp_rate = 0.3

        # initial state reward
        self.state_values = {}
        for i in range(BOARD_ROWS):
            for j in range(BOARD_COLS):
                self.state_values[(i, j)] = 0

    def reset(self):
        self.states = []
        self.State = State()

    def chooseAction(self):

        mx_nxt_reward = 0 # to store best next reward among all possible actions
        action = ""
        if np.random.uniform(0, 1) <= self.exp_rate:
            action = np.random.choice(self.actions)
        else:

            for a in self.actions:
                # if the action is deterministic
                nxt_reward = self.state_values[self.State.nxtPosition(a)]
                if nxt_reward >= mx_nxt_reward:
                    action = a
                    mx_nxt_reward = nxt_reward
        return action


    def takeAction(self, action):
        position = self.State.nxtPosition(action)
        return State(state=position)


                                            # policy iteration methods #
    def policy_evaluation(self,theta,V,V_old):
        print("Policy evaluation")
        compteur_iteration = 0
        while True:
             delta=0
             for x in range(0, BOARD_ROWS):
                 for y in range(0, BOARD_COLS):
                    V_old[(x,y)]=V[(x,y)] # pour ne pas avoir le meme objet V_old=V
```

```
                    print(V_old)
            for x in range(0, BOARD_ROWS):
                for y in range(0, BOARD_COLS):
                    self.State.state = (x, y)
                    nxtState = self.State.nxtPosition(policy[(x, y)])
                    if (x,y)== WIN_STATE or (x,y)== LOSE_STATE:
                        continue
                    nxtreward = self.State.giveNxtReward(nxtState)
                    reward=self.State.giveReward()
                    v = V[(x, y)]
                    V[(x, y)] = reward + discount_factor * V_old[nxtState]
                    delta = max(delta, abs(v - V[(x, y)]))
            compteur_iteration+=1
            if delta<theta:
                print(compteur_iteration)
                return V
                break

    def policy_improvement(self,V):
        print("Policy improvement")
        policy_stable = True

        for x in range(0, BOARD_ROWS):
            for y in range(0, BOARD_COLS):
                self.State.state = (x, y)
                a = policy[(x, y)]
                v = V[self.State.nxtPosition(policy[(x, y)])]
                a_star = a
                v_star = v
                for action in ['u', 'd', 'l', 'r']:
                    v_prime = self.State.giveReward() + discount_factor * V[self.State.nxtPosition(
action)]
                    if v_prime > v_star:
                        a_star = action
                        v_star = v_prime
                        policy_stable = False
                    policy[(x, y)] = a_star
        return policy_stable


    def play(self, rounds=10):
            i = 0
            while i < rounds:
                if self.State.isEnd:
                    reward = self.State.giveReward()
                    self.state_values[self.State.state] = reward
                    print("Game End Reward", reward)
                    for s in reversed(self.states):
                        reward = self.state_values[s] + self.lr * (reward - self.state_values[s])
                        self.state_values[s] = round(reward, 3)
                    self.reset()
                    i += 1
                else:
                    action = self.chooseAction()

                    self.states.append(self.State.nxtPosition(action))
                    print("current position {} action {}".format(self.State.state, action))
                    self.State = self.takeAction(action)
                    self.State.isEndFunc()
                    print("nxt state", self.State.state)
```

```
                    print("----------------------")

    def showValues(self):
        for i in range(0, BOARD_ROWS):
            print('-----------------------------------')
            out = '| '
            for j in range(0, BOARD_COLS):
                out += str(self.state_values[(i, j)]).ljust(6) + ' | '
            print(out)
        print('-----------------------------------')

    def showValueFunction(self,V):
        for i in range(0, BOARD_ROWS):
            print('-----------------------------------')
            out = '| '
            for j in range(0, BOARD_COLS):
                out += str(round(V[(i, j)], 3)) + ' | '
            print(out)
        print('-----------------------------------')


if __name__ == "__main__":

    ag = Agent()
    if algo==0:  #Q-learing
         ag.play(100)
         print(ag.showValues())

    elif algo==1: #Policy-iteration
        policy = np.empty(shape=(BOARD_ROWS, BOARD_COLS),dtype=str)
        for x in range(0, BOARD_ROWS):
            for y in range(0, BOARD_COLS):
                policy[(x, y)] = np.random.choice(ag.actions)
        print(policy)
        V = np.zeros(shape=(BOARD_ROWS, BOARD_COLS))
        V[(1, 1)] = -10
        V[(2, 3)] = 10
        V_old = np.zeros(shape=(BOARD_ROWS, BOARD_COLS))

        while True:
            V=ag.policy_evaluation(0.01,V,V_old)
            policy_stable=ag.policy_improvement(V)
            print(policy)
            if(policy_stable==True):
                print(policy)
                ag.showValueFunction(V)
                break
```

Listing 43: Q-learning and policy iteration on grid world

D.3 Policy iteration for mountain car

```
import gym
import math
import numpy as np
import random

env = gym.make("MountainCar-v0")

LEARNING_RATE = 0.1
DISCOUNT = 0.8
EPISODES = 1000
SHOW_EVERY = 100
force = 0.001
gravity = 0.0025
max_speed = 0.07
min_position = -1.2
max_position = 0.6
goal_position = 0.5
goal_velocity = 0
discretisation = 20

DISCRETE_OS_SIZE = [discretisation] * len(env.observation_space.high)
discrete_os_win_size = (env.observation_space.high - env.observation_space.low) / DISCRETE_OS_SIZE


def get_discrete_state(state):
    discrete_state = (state - env.observation_space.low) // discrete_os_win_size
    return tuple(discrete_state.astype(
        np.int))

def nxtState(state, action):
    position = state[0] * (max_position - min_position) / discretisation - 1.2
    velocity = state[1] * (max_speed + max_speed) / discretisation - 0.07
    velocity += (action - 1) * force + math.cos(3 * position) * (-gravity)
    position += velocity
    position = np.clip(position, min_position, max_position)
    velocity = np.clip(velocity, -max_speed, max_speed)
    if (position == min_position and velocity < 0): velocity = 0
    done = bool(position >= goal_position and velocity >= goal_velocity)
    reward = -1.0
    position, velocity = get_discrete_state((position, velocity))

    return position, velocity, reward, done, {}

def giveReward(state):
    if (state[0] == WIN_STATE[0]) and (state[1] > 0):
        return 0
    else:
        return -0.01


def policy_evaluation(theta, V):
    print("Policy Evaluation")
    compteur_iteration = 0
    new_state = get_discrete_state(env.reset())
    while True:
        delta = 0
        for x in range(0, discretisation):
            for y in range(0, discretisation):
```

```
                V_old[(x, y)] = V[(x, y)]  # pour ne pas avoir le m me objet V_old=V
        for x in range(0, discretisation):
            for y in range(0, discretisation):
                state = (x, y)
                position, velocity, reward, done, _ = nxtState(state, policy[state])
                nextstate = (position, velocity)
                if state == WIN_STATE:
                    continue
                reward = giveReward(state)
                v = V[state]
                V[state] = reward + DISCOUNT * V[nextstate]
                delta = max(delta, abs(v - V[state]))

        compteur_iteration += 1
        if delta < theta:
            print('Value table updated', compteur_iteration, 'times')
            return V
            break


def policy_improvement(V, StateAccessible):
    print("policy improvement...")
    policy_stable = True

    for x in range(0, discretisation):
        for y in range(0, discretisation):
            state = (x, y)
            position, velocity, reward, done, _ = nxtState(state, policy[state])
            nextstate = (position, velocity)
            a = policy[state]
            v = V[nextstate]
            a_star = a
            v_star = v
            for action in [0, 1, 2]:
                positionn, velocityy, _, _, _ = nxtState(state, action)
                exploreState = (positionn, velocityy)
                StateAccessible[(exploreState[0], exploreState[1])] = 1
                v_prime = giveReward(state) + DISCOUNT * V[exploreState]
                if v_prime > v_star:
                    print("The state (%d,%s) changed its value" % (state[0], state[1]))
                    a_star = action
                    if policy[(x, y)] != a_star:
                        policy_stable = False
                    v_star = v_prime
                policy[(x, y)] = a_star
    print(StateAccessible)
    return policy_stable


if __name__ == "__main__":

    WIN_STATE = get_discrete_state((goal_position, goal_velocity))

    policy = np.empty(shape=(discretisation, discretisation), dtype=int)
    for x in range(0, discretisation):
        for y in range(0, discretisation):
            policy[(x, y)] = np.random.randint(0, env.action_space.n)
    print(policy)

    V = np.zeros(shape=(discretisation, discretisation))
```

```
    for x in range(0, discretisation):
        for y in range(0, discretisation):
            V[(x, y)] = -random.uniform(0, 1)

    StateAccessible = np.zeros(shape=(discretisation, discretisation))
    V[WIN_STATE] = 0  # Verifier s'il s'agit bien du win state
    V_old = np.zeros(shape=(discretisation, discretisation))

    iteration = 0
    policy_stable = False

    while not policy_stable:

        V = policy_evaluation(0.001, V)
        print(V)
        policy_stable = policy_improvement(V, StateAccessible)
        iteration += 1

        if (policy_stable == True):
            print(policy)
            print("Optimal policy found after %d iterations" % iteration)

    env.reset()
    episodes = 3000
    i = 0

    for j in range(episodes): # Render with optimal policy
        done = False
        env.reset()
        action = np.random.randint(0, env.action_space.n)
        while not done:
            env.render()
            observation_, reward, done, info = env.step(action)
            position, velocity = get_discrete_state((observation_[0], observation_[1]))
            #print((position, velocity))
            action = policy[(position, velocity)]
```

Listing 44: Policy iteration for mountain car

D.4 Sarsa for mountain car

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import cv2
from decimal import *

env = gym.make("MountainCar-v0")
env.reset()

episodes = 30000
alpha = 0.1
gamma = 0.99
a = 0.1     # to define epsilon function's offset
b = 7/10        # epsilon function crosses x-axis at b*episodes
eps = a
discretisation = 20
meanreward = 0
indxmean = 0     # from which we start computing mean reward
meanscore = 0
std = 0            # standard deviation
SHOW_EVERY = 100

def get_discrete_state(state):
    discrete_state = (state - env.observation_space.low) // discrete_os_win_size
    return tuple(discrete_state.astype(
        np.int))  # on utilise ce tuple pour chercher dans les 3 Q values laquelle est bonne pour l
    'action

def max_action(Q, state, actions=[0,1,2]):
    values = np.array([Q[state, a] for a in actions])
    action = np.argmax(values)
    return action

if __name__=='__main__':

    #env = gym.make("MountainCar-v0")
    DISCRETE_OS_SIZE = [discretisation] * len(env.observation_space.high)
    discrete_os_win_size = (env.observation_space.high - env.observation_space.low) /
    DISCRETE_OS_SIZE
    action_space = [0, 1, 2]
    states = []
    for s1 in range(discretisation):
        for s2 in range(discretisation):
            states.append((s1, s2))
    Q = {}
    for state in states:
        for action in action_space:
            Q[state, action] = 0
    score = 0

    total_reward = []
    epsilon = []
    finalPosition = np.zeros(episodes)
    finalVelocity = np.zeros(episodes)
    win = 0
    i = 0
    successiveWins = 0
```

```
57
58      for j in range(episodes):
59      #loop imposing specific number of wins to be achieved in a raw
60      #while not win:
61          if i % SHOW_EVERY == 0:
62              render = True
63          else:
64              render = False
65          i += 1
66          obs = env.reset()
67          done = False
68          score = 0
69          state = get_discrete_state(obs)
70          action = env.action_space.sample()
71          steps = 0
72          while not done:
73              observation_, reward, done, info = env.step(action)
74              state_ = get_discrete_state(observation_)
75              action_ = max_action(Q, state_) if np.random.random() > eps else env.action_space.
    sample()
76              score += reward
77              Q[state, action] = Q[state, action] + alpha * (reward + gamma * Q[state_, action_] - Q[
    state, action])
78              state = state_
79              action = action_
80              steps += 1
81              if observation_[0] >= 0.5 and observation_[1] >= 0:  # Setting maximum steps to one
    thousand
82                  #done = True
83                  #Q[state, action] = 0
84                  finalPosition[i-1] = observation_[0]
85                  finalVelocity[i-1] = observation_[1]
86              #if steps >= 1000:
87                  #done = True
88
89          if i % 100 == 0:
90              print('episode', i, 'score', score, 'eps', eps)
91
92          if (finalPosition[i-1] < 0.5 or finalVelocity[i-1]<0):
93              win = 0
94              successiveWins = 0
95          else:
96              successiveWins += 1
97          #if successiveWins > 100:
98              #print('Won', successiveWins, 'times in a row at episode', i, ' with position=',
    observation_[0], ' and velocity=', observation_[1])
99              #win = 1
100         total_reward.append(score)
101         epsilon.append(eps)
102         if eps > 0:
103             eps = -a*(1/((b*episodes)*(b*episodes)))*((i*i)-((b*episodes)*(b*episodes))) #second
    order polynomial decresase
104             #eps -= a*1/episodes
105         else:
106             eps = 0
107             if(epsilon[i-2] > 0):
108                 indxmean = i
109
110     samples = total_reward[indxmean:]
111     yy = np.mean(samples)
```

```
    print(yy)
    meanscore = round(yy, 2)
    std = round(Decimal(np.std(samples)), 2)
    max = round(Decimal(np.max(samples)), 2)
    print('standard deviation =', std)
    print('Mean score from epsilon = 0 :', meanscore)
    print('Maximum score after epsilon = 0 :', max)
    legend = "Mean score = %.2f \nStandard deviation = %.2s \nMaximum score = %.2f" % (meanscore,
    std, max)
    fig, ax1 = plt.subplots()
    color = 'tab:blue'
    ax1.set_xlabel('Episodes')
    ax1.set_ylabel('Final Reward', color=color)
    plt.ylim(-200, -40)
    ax1.plot(total_reward, color=color)
    ax1.tick_params(axis='y', labelcolor=color)
    ax2 = ax1.twinx()
    ax1.text(7.5 / 10 * episodes , -60, legend, fontsize=10,
             verticalalignment='top', bbox=dict(facecolor='none', edgecolor='black', boxstyle='
    round,pad=1'))

    plt.axvline(x=7 / 10 * episodes, linewidth=1, color='k', linestyle='--')
    plt.title('Evolution of epsilon and final reward', fontsize=12)
    color = 'tab:orange'
    ax2.set_ylabel('Epsilon', color=color)
    plt.ylim(0, 0.12)
    ax2.plot(epsilon, color=color)
    ax2.tick_params(axis='y', labelcolor=color)

    #fig.tight_layout()  # otherwise the right y-label is slightly clipped
    plt.show()

env.close()
```

Listing 45: SARSA for mountain car

D.5 Regression for mountain car

```
import gym
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

def regression(action,states,T):
    X1 = (np.array(states[action]).T)[0]
    X2 = (np.array(states[action]).T)[1]
    A = np.array([X1*0+1, X1, X2, X1*X2, X1**2, X2**2]).T
    coeffs, r, rank, s = np.linalg.lstsq(A, T[action], rcond=None)
    return coeffs

def getpoly(X, d, e, f):
    Q0= d[0] + d[1]*X[0] + d[2]*X[1] + d[3]*X[0]*X[1] + d[4]*X[0]**2 + d[5]*X[1]**2
    Q1= e[0] + e[1]*X[0] + e[2]*X[1] + e[3]*X[0]*X[1] + e[4]*X[0]**2 + e[5]*X[1]**2
    Q2= f[0] + f[1]*X[0] + f[2]*X[1] + f[3]*X[0]*X[1] + f[4]*X[0]**2 + f[5]*X[1]**2
    Qarray = Q0, Q1, Q2
    return Qarray


def main():
    env = gym.make('MountainCar-v0')
    env.seed(0)

    action, reward, nxt = [], [], []
    states = [[], [], []]
    T = [[], [], []]
    state = env.reset()

    d = np.ones((6,), dtype=int)
    e = np.ones((6,), dtype=int)
    f = np.ones((6,), dtype=int)

    Qarray = getpoly(np.array(state), d, e, f)
    learning_rate = 0.5
    EPISODES = 3500
    discount_factor = 0.95
    epsilon = 0.5 #hasard entre 0 et 1, a quel point il explore
    START_EPSILON_DECAYING = 1
    END_EPSILON_DECAYING = EPISODES // 2 #get an integer
    epsilon_decay_value = epsilon/(END_EPSILON_DECAYING - START_EPSILON_DECAYING)
    compteur = 0
    score = 0

    total_reward = np.zeros(EPISODES)
    epsilontab = []

    for episode in range(EPISODES):
        state = env.reset()
        score = 0
        done = False
        steps = 0
        while not done:
            compteur+=1
            if np.random.random() > epsilon:
              action = np.argmax(Qarray) # action depuis Q table
            else:
              action = np.random.randint(0, env.action_space.n) #random
```

```
            states[action].append(state)
            state2, reward, done, _ = env.step(action)
            done = False
            Qarray = getpoly(np.array(state2), d, e, f)
            Qtarget = reward + discount_factor * np.max(Qarray) #target function
            # Add the following line to implement the alternative target function
            #Qtarget = (1 - learning_rate) * Qarray[action] + learning_rate * Qtarget
            T[action].append(Qtarget)
            state = state2
            score += reward
            steps += 1

            if compteur %20000 == 0:
                if ((np.array(states[0]).T).size!=0 and (np.array(states[1]).T).size!=0 and (np.
    array(states[2]).T).size!=0) :
                    d = regression(0, states, T)
                    e = regression(1, states, T)
                    f = regression(2, states, T)
                    states = [[], [], []]
                    T = [[], [], []]

            if (state[0] >= env.goal_position)
                done=True
                np.array(Qarray)[action]=0

            if (steps >= 1000): #done after 1000 steps
                done =True
                break

        epsilontab.append(epsilon)
        if END_EPSILON_DECAYING >= episode >= START_EPSILON_DECAYING:
            epsilon -= epsilon_decay_value

        total_reward[episode] = score

    fig, ax1 = plt.subplots()
    color = 'tab:blue'
    ax1.set_xlabel('Episodes')
    ax1.set_ylabel('Final Reward', color=color)
    plt.ylim(-1000, 0)
    ax1.plot(total_reward, color=color)
    ax1.tick_params(axis='y', labelcolor=color)
    ax2 = ax1.twinx()

    plt.title('Evolution of epsilon and final reward', fontsize=12)

    color = 'tab:orange'

    ax2.set_ylabel('Epsilon', color=color)
    ax2.plot(epsilontab, color=color)
    ax2.tick_params(axis='y', labelcolor=color)
    plt.show()

    x=np.linspace(-1.2,0.6,100)
    y=np.linspace(-0.07,0.07,100)
    X,Y=np.meshgrid(x,y)
    Z=d[0] + d[1]*X + d[2]*Y + d[3]*X*Y + d[4]*X**2 + d[5]*Y**2
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis', edgecolor='none')
```

```
    ax.set_xlabel('Position')
    ax.set_ylabel('Velocity')
    ax.set_zlabel('Qfunction approximation - action 0')
    plt.show()

    x=np.linspace(-1.2,0.6,100)
    y=np.linspace(-0.07,0.07,100)
    X,Y=np.meshgrid(x,y)
    Z=e[0] + e[1]*X + e[2]*Y + e[3]*X*Y + e[4]*X**2 + e[5]*Y**2
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis', edgecolor='none')
    ax.set_xlabel('Position')
    ax.set_ylabel('Velocity')
    ax.set_zlabel('Qfunction approximation - action 1')
    plt.show()

    x=np.linspace(-1.2,0.6,100)
    y=np.linspace(-0.07,0.07,100)
    X,Y=np.meshgrid(x,y)
    Z=f[0] + f[1]*X + f[2]*Y + f[3]*X*Y + f[4]*X**2 + f[5]*Y**2
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis', edgecolor='none')
    ax.set_xlabel('Position')
    ax.set_ylabel('Velocity')
    ax.set_zlabel('Qfunction approximation - action 2')
    plt.show()

if __name__ == '__main__':
    main()
```

Listing 46: Regression on mountain car

D.6 Mountain car with pytorch

```
import gym
import numpy as np
import torch
import random
from torch import nn
import torch.optim as optim
import torch.nn.functional as F
import copy
import matplotlib.pyplot as plt
from torch.utils.tensorboard import SummaryWriter
from torch.utils.tensorboard import SummaryWriter
from google.colab import files
import os
from os.path import exists


logs_base_dir = "runs"
os.makedirs(logs_base_dir, exist_ok=True)
%load_ext tensorboard
%tensorboard --logdir {logs_base_dir}

%reload_ext tensorboard
%tensorboard --logdir {logs_base_dir}



class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):  # nn model 2 states --> 64layer ---> 64layer ----> 3 Q-action pairs
        super().__init__()
        self.fc1 = nn.Linear(state_dim, 512, bias=False)  # 2d state space
        #self.fc2 = nn.Linear(64, 64, bias=False)
        self.fc3 = nn.Linear(512, action_dim, bias=True)  # one output per action
        self.device = torch.device("cpu")

    def forward(self, x):  # forward function
        state = torch.Tensor(x).to(self.device)
        x = F.relu(self.fc1(state))
        #x = F.relu(self.fc2(x))
        actions = self.fc3(x)
        return actions

class Agent:
    def __init__(self, state_dim, action_dim):
        self.network = DQN(state_dim, action_dim).to(device)  # creates network from nn model
        self.target_network = copy.deepcopy(self.network).to(device)  # create target_network from network
        self.optimizer = optim.Adam(self.network.parameters(), lr=0.005)  # optimizer, parameters
        self.memory = []
        self.states_memory = []
        self.actions_memory = []
        self.rewards_memory = []
        self.next_states_memory = []
        self.dones_memory = []

    def update(self, batch, update_dqn_target):
        states, actions, rewards, next_states, dones = zip(*batch)  # from random batch ( from buffer )
```

```
        self.states_memory = torch.from_numpy(np.array(states)).float().to(device)  # collect states
        self.actions_memory = torch.from_numpy(np.array(actions)).to(device).unsqueeze(1)  # collect actions
        self.rewards_memory = torch.from_numpy(np.array(rewards)).float().to(device).unsqueeze(1)  # collect rewards
        self.next_states_memory = torch.from_numpy(np.array(next_states)).float().to(device)  # etc ...
        self.dones_memory = torch.from_numpy(np.array(dones)).to(device).unsqueeze(1)
        if not update_dqn_target % UPDATE_TARGET:
            agent.target_network = copy.deepcopy(agent.network)

    def train(self):
        target = self.rewards_memory + (gamma * self.target_network(self.next_states_memory).detach().max(1)[0].unsqueeze(1)) * (~self.dones_memory)  # q-target


        Q_current = self.network(self.states_memory).gather(-1, self.actions_memory)  # Q_current, gather will index the rows of the q-values
        self.optimizer.zero_grad()  # optimizer
        loss = F.smooth_l1_loss(target, Q_current)  # loss
        loss.backward()  # backward
        self.optimizer.step()  # step optimizer -----> improves network

    def act(self, env, state, eps):
        if random.random() < eps:
            return env.action_space.sample()
        state = torch.tensor(state).to(device).float()
        Q_values = self.network(state.unsqueeze(0)).detach()
        return np.argmax(Q_values.cpu().data.numpy())  # choose action from state ---> network

def plotLearning(x, scores, epsilons, filename, lines=None):
    fig=plt.figure()
    ax=fig.add_subplot(111, label="1")
    ax2=fig.add_subplot(111, label="2", frame_on=False)

    ax.plot(x, epsilons, color="C0")
    ax.set_xlabel("Game", color="C0")
    ax.set_ylabel("Epsilon", color="C0")
    ax.tick_params(axis='x', colors="C0")
    ax.tick_params(axis='y', colors="C0")

    N = len(scores)
    running_avg = np.empty(N)
    for t in range(N):
        running_avg[t] = np.mean(scores[max(0, t-20):(t+1)])

    ax2.scatter(x, scores, color="C1")
    #ax2.xaxis.tick_top()
    ax2.axes.get_xaxis().set_visible(False)
    ax2.yaxis.tick_right()
    #ax2.set_xlabel('x label 2', color="C1")
    ax2.set_ylabel('Score', color="C1")
    #ax2.xaxis.set_label_position('top')
    ax2.yaxis.set_label_position('right')
    #ax2.tick_params(axis='x', colors="C1")
    ax2.tick_params(axis='y', colors="C1")

    if lines is not None:
       for line in lines:
```

```
            plt.axvline(x=line)

    plt.savefig(filename)


if __name__ == '__main__':
    writer = SummaryWriter()
    BATCH_SIZE = 64
    UPDATE_TARGET = 128
    gamma = 0.98
    seed = 0
    np.random.seed(seed)
    random.seed(seed)
    env = gym.make('MountainCar-v0')
    env.seed(seed)
    torch.manual_seed(seed)
    device = torch.device("cpu")
    agent = Agent(state_dim=2, action_dim=2)
    episodes =3000
    eps = 0.5
    eps_coeff = 0.995
    dqn_updates = 0
    rewards = []
    epsh= []
    mean_rh = []
    cpt = 0
    goal = 0.8/0.9

    step = -1
    for i in range(1, episodes + 1):
        state = env.reset()
        done = False
        total_reward = 0
        while not done:
            step += 1
            action = agent.act(env, state, eps)
            next_state, reward, done, _ = env.step(action)
            #next_state = relu_compat(next_state)
            total_reward += reward
            if step < 20000:
                agent.memory.append(None)
            agent.memory[step % 20000] = (state, action, reward, next_state, done)


            if step >= 10000: #and total_reward<-120:
                sample = random.sample(agent.memory, BATCH_SIZE)
                agent.update(sample, step)
                agent.train()


            state = next_state
        if state[0] >= env.goal_position:

              cpt += 1
              break
        eps *= eps_coeff

        rewards.append(total_reward)
        writer.add_scalar('score', total_reward, i)
        epsh.append(eps)
```

```
        mean_r = np.mean(rewards)
        mean_rh.append(mean_r)
        max_r = np.max(rewards)
        if i % 5 == 0:
            print(f'\repisode {i}, eps = {eps}, mean = {mean_r}, max = {max_r}, accuracy = {cpt}/{i
    }')
x = [i+1 for i in range(episodes)]
filename = "plot.png"
plotLearning(x, rewards, epsh, filename)
writer.close
```

Listing 47: Mountain car with pytorch

D.7 Final deep Q-learning for Atari

```
# -*- coding: utf-8 -*-
"""Untitled1.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1MqYktxF7eixpfMaRkBL15K7qU1Uxjs1i
"""

import numpy as np
from collections import deque
import gym
from gym import spaces
import cv2
cv2.ocl.setUseOpenCL(False)

class NoopResetEnv(gym.Wrapper):
    def __init__(self, env, noop_max=30):
        """Sample initial states by taking random number of no-ops on reset.
        No-op is assumed to be action 0.
        """
        gym.Wrapper.__init__(self, env)
        self.noop_max = noop_max
        self.override_num_noops = None
        self.noop_action = 0
        assert env.unwrapped.get_action_meanings()[0] == 'NOOP'

    def reset(self, **kwargs):
        """ Do no-op action for a number of steps in [1, noop_max]."""
        self.env.reset(**kwargs)
        if self.override_num_noops is not None:
            noops = self.override_num_noops
        else:
            noops = self.unwrapped.np_random.randint(1, self.noop_max + 1) #pylint: disable=E1101
        assert noops > 0
        obs = None
        for _ in range(noops):
            obs, _, done, _ = self.env.step(self.noop_action)
            if done:
                obs = self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)

class FireResetEnv(gym.Wrapper):
    def __init__(self, env):
        """Take action on reset for environments that are fixed until firing."""
        gym.Wrapper.__init__(self, env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3
    def reset(self, **kwargs):
        self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(2)
        if done:
```

```
            self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)

class EpisodicLifeEnv(gym.Wrapper):
    def __init__(self, env):
        """Make end-of-life == end-of-episode, but only reset on true game over.
        Done by DeepMind for the DQN and co. since it helps value estimation.
        """
        gym.Wrapper.__init__(self, env)
        self.lives = 0
        self.was_real_done  = True

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.was_real_done = done
        # check current lives, make loss of life terminal,
        # then update lives to handle bonus lives
        lives = self.env.unwrapped.ale.lives()
        if lives < self.lives and lives > 0:
            # for Qbert sometimes we stay in lives == 0 condtion for a few frames
            # so its important to keep lives > 0, so that we only reset once
            # the environment advertises done.
            done = True
        self.lives = lives
        return obs, reward, done, info

    def reset(self, **kwargs):
        """Reset only when lives are exhausted.
        This way all states are still reachable even though lives are episodic,
        and the learner need not know about any of this behind-the-scenes.
        """
        if self.was_real_done:
            obs = self.env.reset(**kwargs)
        else:
            # no-op step to advance from terminal/lost life state
            obs, _, _, _ = self.env.step(0)
        self.lives = self.env.unwrapped.ale.lives()
        return obs

class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env, skip=4):
        """Return only every `skip`-th frame"""
        gym.Wrapper.__init__(self, env)
        # most recent raw observations (for max pooling across time steps)
        self._obs_buffer = np.zeros((2,)+env.observation_space.shape, dtype=np.uint8)
        self._skip       = skip

    def reset(self):
        return self.env.reset()

    def step(self, action):
        """Repeat action, sum reward, and max over last observations."""
        total_reward = 0.0
        done = None
        for i in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            if i == self._skip - 2: self._obs_buffer[0] = obs
```

```
            if i == self._skip - 1: self._obs_buffer[1] = obs
            total_reward += reward
            if done:
                break
        # Note that the observation on the done=True frame
        # doesn't matter
        max_frame = self._obs_buffer.max(axis=0)

        return max_frame, total_reward, done, info

    def reset(self, **kwargs):
        return self.env.reset(**kwargs)

class ClipRewardEnv(gym.RewardWrapper):
    def __init__(self, env):
        gym.RewardWrapper.__init__(self, env)

    def reward(self, reward):
        """Bin reward to {+1, 0, -1} by its sign."""
        return np.sign(reward)

class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env):
        """Warp frames to 84x84 as done in the Nature paper and later work."""
        gym.ObservationWrapper.__init__(self, env)
        self.width = 84
        self.height = 84
        self.observation_space = spaces.Box(low=0, high=255,
            shape=(1, self.height, self.width), dtype=np.uint8)

    def observation(self, frame):
        frame = frame[34: 194, :160]
        frame = frame.mean(2)
        frame = cv2.resize(frame, (self.width, self.height), interpolation=cv2.INTER_AREA)
        frame = np.reshape(frame, [1, 84, 84])
        return frame


class FrameStack(gym.Wrapper):
    def __init__(self, env, k):
        """Stack k last frames.
        Returns lazy array, which is much more memory efficient.
        See Also
        --------
        baselines.common.atari_wrappers.LazyFrames
        """
        gym.Wrapper.__init__(self, env)
        self.k = 4
        self.frames = deque([], maxlen=k)
        shp = env.observation_space.shape
        self.observation_space = spaces.Box(low=0.0, high=1.0, shape=(4, shp[1], shp[2]), dtype=np.
    uint8)

    def reset(self):
        ob = self.env.reset()
        for _ in range(self.k):
            self.frames.append(ob)
        return self._get_ob()

    def step(self, action):
```

```
        ob, reward, done, info = self.env.step(action)
        self.frames.append(ob)
        return self._get_ob(), reward, done, info

    def _get_ob(self):
        assert len(self.frames) == self.k
        return LazyFrames(list(self.frames))

class ScaledFloatFrame(gym.ObservationWrapper):
    def __init__(self, env):
        gym.ObservationWrapper.__init__(self, env)

    def observation(self, observation):
        # careful! This undoes the memory optimization, use
        # with smaller replay buffers only.
        return np.array(observation).astype(np.float32) / 255.0

class LazyFrames(object):
    def __init__(self, frames):
        """This object ensures that common frames between the observations are only stored once.
        It exists purely to optimize memory usage which can be huge for DQN's 1M frames replay
        buffers.
        This object should only be converted to numpy array before being passed to the model.
        You'd not believe how complex the previous solution was."""
        self._frames = frames
        self._out = None

    def _force(self):
        if self._out is None:
            self._out = np.concatenate(self._frames, axis=0)
            self._frames = None
        return self._out

    def __array__(self, dtype=None):
        out = self._force()
        if dtype is not None:
            out = out.astype(dtype)
        return out

    def __len__(self):
        return len(self._force())

    def __getitem__(self, i):
        return self._force()[i]

def make_atari(env_id):
    env = gym.make(env_id)
    assert 'NoFrameskip' in env.spec.id
    env = NoopResetEnv(env, noop_max=30)
    env = MaxAndSkipEnv(env, skip=4)
    return env

def wrap_deepmind(env, episode_life=True, clip_rewards=True, frame_stack=True, scale=False):
    """Configure environment for DeepMind-style Atari.
    """
    if episode_life:
        env = EpisodicLifeEnv(env)
    if 'FIRE' in env.unwrapped.get_action_meanings():
        env = FireResetEnv(env)
    env = WarpFrame(env)
```

```
    if scale:
        env = ScaledFloatFrame(env)
    if clip_rewards:
        env = ClipRewardEnv(env)
    if frame_stack:
        env = FrameStack(env, 4)
    return env

env_id = "PongNoFrameskip-v4"
env    = make_atari(env_id)
env    = wrap_deepmind(env)

# Commented out IPython magic to ensure Python compatibility.
import gym
import numpy as np
import torch
import random
from torch import nn
import torch.optim as optim
import torch.nn.functional as F
import copy
import matplotlib.pyplot as plt
import matplotlib
import cv2
from torch.utils.tensorboard import SummaryWriter
from google.colab import files
import os
from os.path import exists


logs_base_dir = "runs"
os.makedirs(logs_base_dir, exist_ok=True)
# %load_ext tensorboard
# %tensorboard --logdir {logs_base_dir}

# %reload_ext tensorboard
# %tensorboard --logdir {logs_base_dir}

load = False


class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(state_dim[0], 32, kernel_size=8, stride=4, bias=False)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
        self.fc1 = nn.Linear(3136, 512)
        self.fc2 = nn.Linear(512, action_dim)
        self.device = 'cuda'

    def forward (self, x):
        state = x.to(self.device).float()/255
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.fc1(x.view(x.size(0), -1)))   #flattening
        actions = self.fc2(x)
        return actions

```

```
298
299
300 class Agent:
301     def __init__(self, state_dim, action_dim):
302         self.network = DQN(state_dim, action_dim).to(device)  # creates network from nn model
303         self.target_network = copy.deepcopy(self.network).to(device)  # create target_network from
    network
304         self.optimizer = optim.Adam(self.network.parameters(), lr=0.00025)  # optimizer, parameters
    , lr very low
305         self.memory = []
306         self.states_memory = []
307         self.actions_memory = []
308         self.rewards_memory = []
309         self.next_states_memory = []
310         self.dones_memory = []
311
312     def update(self, batch, update_dqn_target):
313         states, actions, rewards, next_states, dones = zip(*batch)  # from random batch ( from
    buffer )
314         self.states_memory = torch.from_numpy(np.array(states)).float().to(device)  # collect
    states
315         self.actions_memory = torch.from_numpy(np.array(actions)).to(device).unsqueeze(1)  #
    collect actions
316         self.rewards_memory = torch.from_numpy(np.array(rewards)).float().to(device).unsqueeze(1)
    # collect rewards
317         self.next_states_memory = torch.from_numpy(np.array(next_states)).float().to(device)  # etc
    ...
318         self.dones_memory = torch.from_numpy(np.array(dones)).to(device).unsqueeze(1)
319         if not update_dqn_target % UPDATE_TARGET:
320             agent.target_network = copy.deepcopy(agent.network)
321
322     def train(self):
323         with torch.no_grad():
324           argmax = self.network(self.next_states_memory).detach().max(1)[1].unsqueeze(1)  # action
    max from next_states from batch above
325           target = self.rewards_memory + (gamma * self.target_network(self.next_states_memory).
    detach().gather(1, argmax)) * (~self.dones_memory)  # q-target
326
327         Q_current = self.network(self.states_memory).gather(-1, self.actions_memory)  # Q_current
328         self.optimizer.zero_grad()  # optimizer
329         loss = F.mse_loss(target, Q_current)  # loss
330         loss.backward()  # backward
331         self.optimizer.step()  # step optimizer -----> improves network
332
333     def act(self, env, state, eps):
334         if random.random() < eps:
335             return env.action_space.sample()
336         state = torch.tensor(state).to(device).float()
337         with torch.no_grad():
338             Q_values = self.network(state.unsqueeze(0))
339         return np.argmax(Q_values.cpu().data.numpy())  # choose action from state ---> network
340
341
342     def save(self, episode, mean_reward):
343         file_name = "/content/gdrive/My Drive/" + "pacman" + ".ptm"
344         agent_state = { "episode" : episode,
345                         "network" : self.network.state_dict(),
346                         "optimizer_state_dict": self.optimizer.state_dict(),
347                         "mean_reward": mean_reward};
348         torch.save(agent_state, file_name)
```

```
        print("Agent's state saved to ", file_name)

    def load(self):
        file_name = "/content/gdrive/My Drive/" + "pacman" + ".ptm"
        checkpoint = torch.load(file_name)
        self.network.load_state_dict(checkpoint["network"])
        self.network.to(device)
        self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        self.episode = checkpoint["episode"]
        self.mean_reward = checkpoint["mean_reward"]
        print("Loaded network model state from", file_name,
              "stopped at episode:", self.episode,
              " which fetched a mean reward of:", self.mean_reward)




if __name__ == '__main__':
    writer = SummaryWriter()
    BATCH_SIZE = 32
    UPDATE_TARGET = 1000
    gamma = 0.99
    seed = 0
    np.random.seed(seed)
    random.seed(seed)
    env.reset()
    env.seed(seed)
    torch.manual_seed(seed)
    device = torch.device("cuda")
    agent = Agent(env.observation_space.shape , env.action_space.n)
    episodes = 50000
    eps = 1
    eps_coeff = 0.99
    dqn_updates = 0
    rewards = []
    epsh= []
    mean_rh = []
    moy = 0
    true_rew = 0
    debut = 1
    if load == True :
      agent.load()
      debut = np.uint64(agent.episode)

    step = -1
    for i in range(debut, episodes + 1):
        state = env.reset()
        done = False
        total_reward = 0

        while not done:
            #env.render()
            step += 1
            action = agent.act(env, state, eps)
            next_state, reward, done, _ = env.step(action)

            total_reward += reward
            eps = max(0.02, 1 - step / (3*10**5))

            if step < 75000:
```

```
                agent.memory.append(None)
            agent.memory[step % 75000] = (state, action, reward, next_state, done)

            if step >= 10000:
                sample = random.sample(agent.memory, BATCH_SIZE)
                agent.update(sample, step)
                agent.train()


            state = next_state


        writer.add_scalar('score', total_reward, step)
        moy += total_reward

        if i % 10 == 0 :
            moy10 = moy/10
            print(f'\rmoyenne10ep {moy10},episode {i},epsilon = {eps}, step = {step}')
            #if i % 100 == 0 :
              #agent.save(i, moy10)
            moy = 0

writer.close
```

Listing 48: Final Deep Q-learning for Atari

References

[1] M. Lapan, *Deep Reinforcement Learning Hands-On*. Packt Publishing Ltd, 2020.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] J. Zhang, "Implement Grid World with Q-Learning," Aug. 2019. Library Catalog: towardsdatascience.com.

[4] "Locally Weighted Linear Regression (Loess) — Data Blog." https://xavierbourretsicotte.github.io/loess.html.

[5] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*, vol. 39. CRC press, 2010.

[6] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[7] H. van Hasselt, "Estimating the Maximum Expected Value: An Analysis of (Nested) Cross Validation and the Maximum Sample Average," *arXiv:1302.7175 [cs, stat]*, Mar. 2013. arXiv: 1302.7175.

[8] "openai/baselines." Library Catalog: github.com/openai/baselines/blob/master/baselines/.