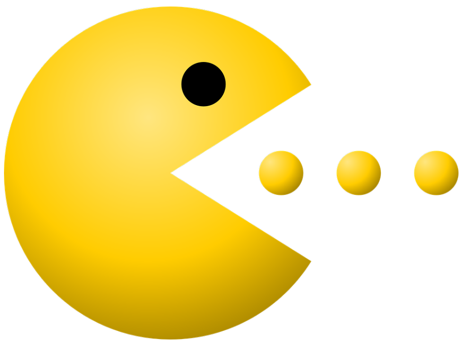
Mathematics and Data Science

Reinforcement Learning

RL Project : Pac-Man

Authors:
MAHMOUDI Amir
DAEMS Chloé
LAISNEY Anne-Claire
March 29, 2022



CentraleSupélec

Abstract

This is our report for the Project in Reinforcement Learning, a Pac-Man. The purpose of this work was to train in the best way possible different agents (Q-Learning, Expected-Sarsa, Deep Q-Learning) in a Reinforcement Learning environment of the Game Pac-Man.

1 Introduction and Motivation

1.1 Description

Pac-Man is a famous maze game created in 1979 by designer Toru Iwatani for the Japanese company Namco. The object of the game is to move Pac-Man, a character in the shape of a pie chart, inside a maze, and make him collect all the pac-gums (fruits) there while avoiding being hit by ghosts. If Pac-Man makes contact with a ghost, he lost and the game is over.
We searched for related work and found many different environments for Pac-Man. We wanted to use the agents we studied in class in order to put it in application. We also were motivated to include Deep Learning in the process, as we haven't tried deep in reinforcement before. We found that a variant of Q-Learning, Deep Q-Learning that uses neural networks, is common for Atari games, but isn't really a break-through and common for Pac-Man.

1.2 Summary

In this report, we will first describe our environment, that includes two different layouts **smallGrid** and **mediumGrid**. Secondly, we will describe the three agents that we tried: **Q-Learning**, **Expected-Sarsa** and **Deep Q-Learning**. Finally, we will discuss the results and conclude.

2 Environment

We used an existing environment from Free Python Games [1] that we modified, adding a step method and a render method, as their render was not separated from the code.
We created an environment with two different fields: the **smallGrid** layout, and the **mediumGrid** layout.

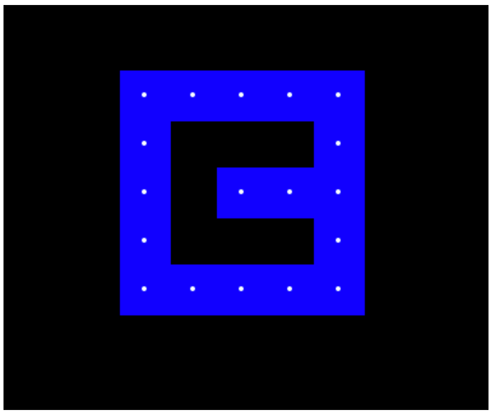
2.1 Small Grid

In the class *PacManEnv2*, we implemented a field of the size $(5, 5)$, with a Pac-man, 1 ghosts and fruits at every spot that isn't a wall (the tile is a 1 for a fruit, and 0 for a wall). The pros for this grid is the computational time for testing the agent but it is a easy pacman problem since the maze is pretty simple. Also, in the [2] paper, it was states that SARSA was good on this kind of grid.

2.2 Medium Grid

In the class *PacManEnv2*, we implemented a field of the size $(9, 15)$, with a Pac-man, 2 ghosts and fruits at every spot that isn't a wall (the tile is a 1 for a fruit, and 0 for a wall). This one is more complicated as you can see in the Fig. 1 and the agent takes more time to train.

2.3 Basic sum of rewards


(a) small env

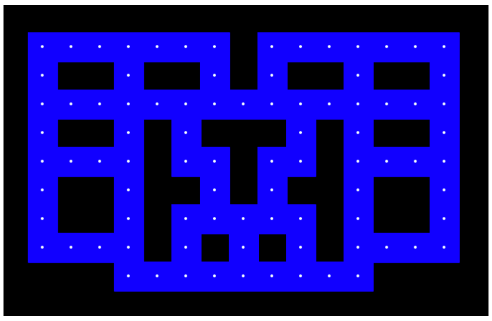

(b) medium env

Figure 1: the different environment

2.4 Rewards

If the pac-man moves through all the squares without being eaten by a ghost, the reward is 100. Going into the wall gives him a reward of -5 in order to prevent it. If the player goes on a new case, he will get a reward of 10. If the ghost eats the player, the player receives a reward of -100. We also added a reward of -1 for each action where the agent does not eat a fruit. This will prevent noisy actions and the agent will be more eager to go on a position with a fruit than a fruitless one.

2.5 Action Space

Pac-Man just needs to know about the status of the neighbors cells to know where to go. It can take the five directions: left, right, up, down and stay. Hence, the action space is a set of 5 actions: L, R, U, D and S.

2.6 Observation Space

We tried different types of observation spaces. At first, we tried taking all the map for the *Q-Learning* and *Sarsa*. However, by flattening the field, we add too many possible states. This resulted into a high computational time and not so good results. And anyways, it would have been too adapted to this field.
Secondly, we tried taking neighbor spaces, that includes two neighbors in order to have $t + 1$ for the *Sarsa*. Undeniably, it decreased the computational time. However, for the *mediumGrid*, when the pac-man had eaten all the fruits of a zone, he stayed in the same zone and went around in circles.
After, we had to think of a way to lead him towards the uneaten fruits. First, we thought of separating the field into 4 zones, and we would indicate him the zone with the maximum fruits. However, after thinking, as we have only $t + 1$ vision, it didn't bring a lot of information, and wasn't a game changer for the pac-man.
After reflection, we were inspired by *Berkeley* [6] practical work on Pac-Man, and our final observation space is: **(dx_closest_fruit, dy_closest_fruit, dx_closest_ghost, dy_closest_ghost, bool_if_there_is_a_wall_up, bool_if_there_is_a_wall_down, bool_if_there_is_a_wall_left,**

bool_if_there_is_a_wall_right). This observation space gives us all the information needed to avoid loosing points.

3 Agent : Q-Learning

We decided to implement first a off-policy based algorithm, which updates with regards to the optimal policy.
For Q-learning, the inputs are the states, actions and rewards generated by the Pacman game. To chose its action, the agent follows an epsilon-greedy policy. We reused the agent seen in TP4 and the presented methods. We expect not good results, as it isn't a $t + 1$ vision.
According to [3], given a policy, the corresponding action-value function Q (in the state s and action a, at time step t), i.e. $Q(s_t, a_t)$, can be updated as follows [3]:
$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma * max_a Q(s_t + 1, a) - Q(s_t, a_t))$

The learning-rate (α) and discount (γ) factors are fixed parameters that do not change. We update the current-q of that action given a state with a new-q. The process of q learning and updating the q-table is repeated until the agent has explored all possible states with all possible actions.

4 Agent : Expected-Sarsa

Sarsa is an on-policy reinforcement learning algorithm that estimates $Q_\pi(s, a)$ for a given state- action pair (s, a), where the agent is following some policy π.
According to [3], in the Sarsa algorithm, given a policy, the corresponding action-value function Q (in the state s and action a, at time step t), i.e. $Q(s_t, a_t)$, can be updated as follows:
$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma * Q(s_t + 1, a_t + 1) - Q(s_t, a_t))$ where gamma is the discount factor and r_t is the reward received from the environment at time step t.
We reused the agent seen in TP4 and the presented methods. You can find on our GitHub [7] two Gifs of the Sarsa agent training on both the *smallGrid* and *largeGrid*.

5 Agent : Deep Q-Network

According to [2], Deep Q-learning (DQL) can implicitly 'extract' important features, and interpolate Q-values of enormous state-action pairs without consulting large data tables. We tried to implement it, however we didnt obtain great results. You can find our classes in the DQN.py file, which includes the classes *ReplayMemory()*, *DQN()* and *DQLAgent()*.

5.1 Architecture

To implement the DQN, we used a similar architecture than [2] because their results (winning rate of approximately 92%) are really encouraging and better than any agent we have implemented. Therefore, we used TensorFlow to implement a Convolutional Neural Network (CNN), to generate the Q-values corresponding to the possible directions at each state. After fine-tuning, the architecture is composed of:

- 3 convolutional layers (3x3x8), (3x3x16) and (4x4x32) (*nn.Conv2d()*)
- a flattening layer (*nn.Flatten()*)

- one fully connected layer with 256 neurons (*nn.Linear()*)
- Relu activation function for all layers

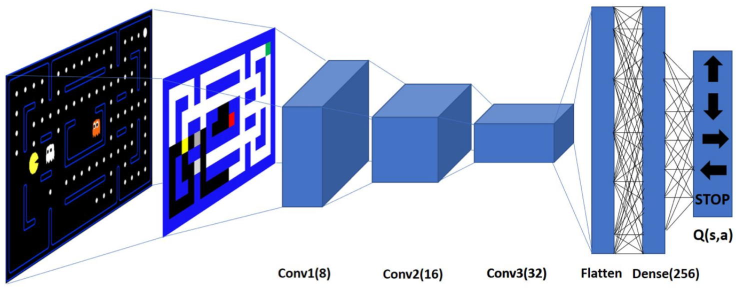


Figure 2: DQN Architecture [2]

We used an epsilon-greedy strategy for exploration.

5.2 Replay Memory

We used replay memory (coded in the class *ReplayMemory*) to store he last 100,000 experiences (state, action, reward, next state). These are stored in each transition of Pac-man game, which is a *namedtuple* from the *collections* class.
We stored state space information to the replay memory at every step of a Pac-man game and we access information from it at every step of training. We used the *deque* data structure from the *collections* class.

5.3 Parameters

We took inspiration from [4] to code our training function. We used the following hyper-parameters:

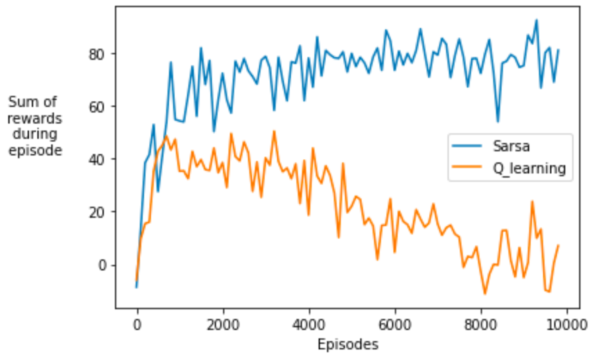
- Size of replay memory batch size: 32
- optimizer: Adam
- learning rate: 0.01
- loss function: MSE

6 Discussion and Results

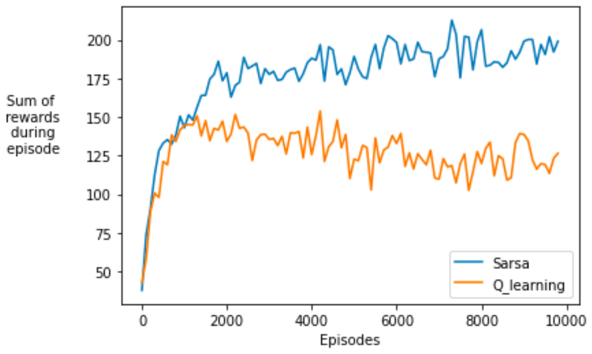
To have a raw estimation on how our agents are performing, we ran 1,000 games for each agents on each environment. We removed the rendering to help the computation and calculate a ratio of the games won over the 1,000 games.

For the small environment, we first had 100% game wins on Sarsa but the agent was really careful thus was really slow to collect all fruits. Adding the reward rule of -1 as stated before helped it to win faster but decreased the winning rate to 99.2% - the agent takes more risks. In the other hand, the Q Learning Agent had 89% game won. For the medium environment, Sarsa had 14.8% games won and Q-Learning had 6.5% games won. This is not surprising has it was mentioned in the state of the art that those Agent weren't really good on big grids.

6.1 Basic sum of rewards



(a) Sum of rewards on small env



(b) Sum of rewards on medium env

Figure 3: Basic sum of rewards

Here in Fig.3, you can observe the sum of rewards for both agents on the different environments. We see here that the Sarsa Agent performs better than the Q-learning one in both. This can be explain by the fact that the Sarsa Agent is an on-policy algorithm meaning it relays more on its past experience sampled π. Moreover, we see that the gap between both agents at the end of the run is pretty similar for both environments, approximately of 50.
Also, we added a reward of -1 for each action where the agent does not eat a fruit to speed it up. But Q-Learning can't see at $t + 1$ so the agent tends to move blindly decreasing his sum of rewards. Finally, the Sarsa Agent tends to be quicker so it is doing less noisy action (action with no fruit eaten).

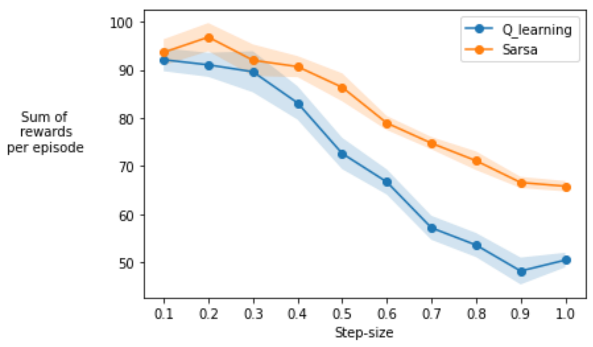
6.2 Parameter-sweep

To improve our agents, we did a *parameter sweep* on the step sizes from 0.1 to 1.0. For each of them, we computed 10 runs of 1,000 episodes for the small environment and 100 episodes for the medium, each point is the mean sum of rewards obtained at each step size and the area in light color represent the mean value the standard deviation. We clearly see on Fig 4 (a) here that the best step size in the small environment for the Q-learning is **0.1** with a sum of 90 and for the Sarsa it is **0.2** with a sum of 97. With 1,000 episodes per run, the slope is smoother than the one in Fig 4 (c), but the trend shows a clear decrease of the sum of rewards the greater the step size is.

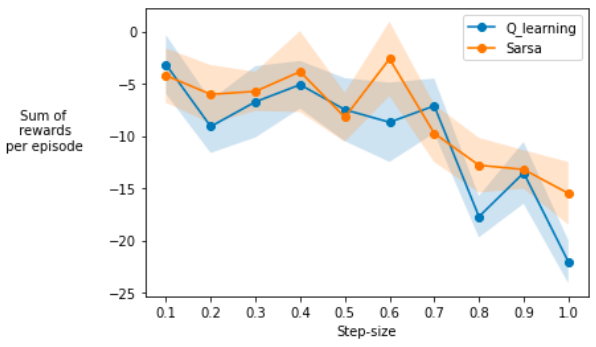
The graph shows the evolution for the Q-learning and Expected-Sarsa agents regarding the step size. We see that the difference in performance persists and gets even bigger, as the difference is proportional to the step size. This can be explain by the fact that the Q-learning Agent is more optimistic while updating, so when it is taking a wrong decision it is amplified by the step size. Thus having a bigger step size tends to make the Q-learning agent doing more bad moves with high negative rewards than Sarsa Agent.

N.B. : the sum of rewards are negative because the agent is still bad after 100 episodes and it earns less than a 100 rewards of fruit. When it gets killed by a ghost, -100 points are added to its reward.

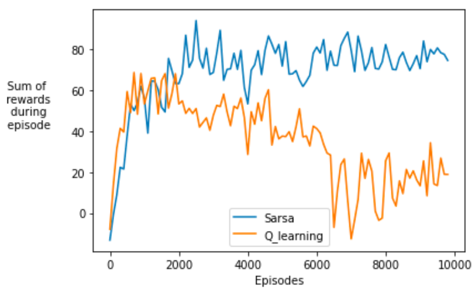
In Fig 4 (c), we observe that both agents are converging, Sarsa is around 80 and 20 for the Q-Learning. Also, Sarsa agent converges around the episode 2,000, on the contrary,



(a) Sum of rewards on each step sizes 1000 episodes



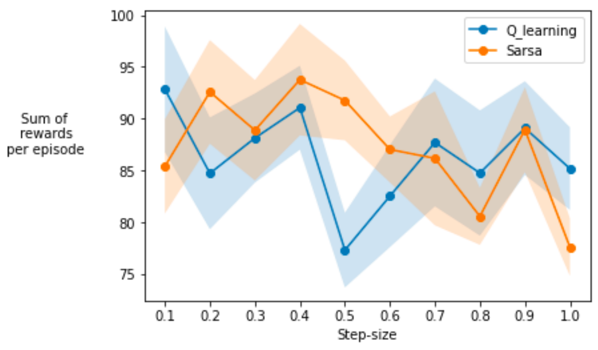
(b) Sum of rewards on each step sizes 100 episodes



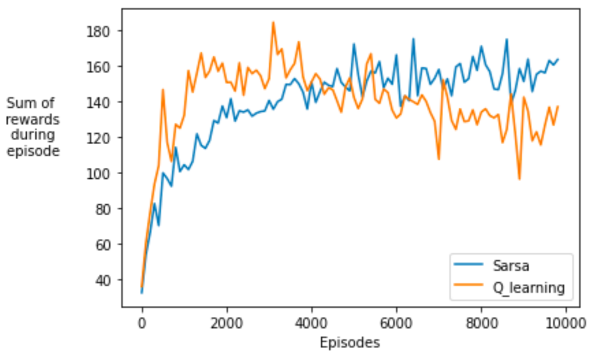
(c) Sum of rewards with best step sizes

Figure 4: Parameter Sweep on small environment

it is more around the 8,000. For the medium environment, the best step size are 0.1 for the Q-Learning with a sum of 93 and 0.4 for Sarsa with a sum of 93 also.



(a) Sum of rewards on each step sizes



(b) Sum of rewards with best step sizes

Figure 5: Parameter Sweep on medium environment

We then tried our agents with the best parameters for each one on 10,000 episodes to see if they converge on Fig 5 (c). It is the case for both. The sum of rewards for the Sarsa agent is now around 160 and around 140 for the Q-Learning agent. Also, Sarsa agent converges around the episode 5,000, while the Q Learning agent converges around 7,000 episodes. With their best parameters, the gap between both agents is smaller but we still have the trend of *fast increasing followed by a decrease to stabilize*. Whereas, Sarsa agent is constantly increasing before converging.

6.3 DQN

The results, as mentioned in the paper [2], weren't good. This explains why we chose not to add it to our notebook, but put it only in a *py* file. Indeed, we trained our model on 1,000 episodes, and the results showed that only 43 out of them were winning games for the *mediumGrid*. We suspect we didn't quite understand totally the concept, which falsifies the results, and stopped us from improving and making the model more robust.

7 Conclusion

To conclude, we found this project much more challenging than we expected it to be. At first, we were planning on using an OpenAI Gym environment [5], thinking there will be all the rewards and that we could just focus on the implementation and training of the agents. This environment didn't fit to our needs, as it was image-based. Therefore, we spent a lot of time of the modification of the FreeGames Pac-Man environment [1]. We saw that changing the rewards and the type of observation spaces was a game changer.

We were frustrated with the results of the DQN because we spent a lot of time on it, and the results aren't good, as mentioned in the paper [2], and found elsewhere throughout different articles on the Internet.

On another hand, we had to change the render, as the original one that uses *Turtle* library took a really long time to train, as the render was not distinct from the training. Finally, our best agent is definitely the Sarsa, as it gave us the best results. In its way of playing, we can definitely see that it is learning quite well.

References

[1] https://grantjenks.com/docs/freegames/pacman.html

[2] Abeynaya Gnanasekaran, Jordi Feliu Faba, Jing An, "Reinforcement learning in Pacman".

[3] Reinforcement Learning: An Introduction, Richard S. Sutton, and Andrew G. Barto,MIT Press, Cambridge, MA, 2018w

[4] https://github.com/H13r0glyph/Pacman-Deep-Q-Network

[5] https://gym.openai.com/envs/MsPacman-ram-v0/

[6] http://ai.berkeley.edu/reinforcement.html

[7] https://github.com/chloedia/PacMan