Learn to play Snake using Reinforcement Learning

Anonymous Authors

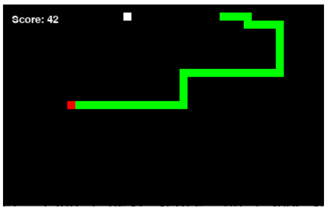


Fig. 1: Snake made in `Pygame`

***Abstract*—Snake is a classic game that can be easily adapted to a reinforcement learning approach. We started by developing an environment where we could work and test different reinforcement learning methods. We then concentrated on finding states that would allow us to describe the environment in the most complete and lightest way possible. Then we were able to exploit the SARSA and *Q*-Learning methods in various ways before focusing on Deep *Q*-Learning methods. We developed a small framework that allowed us to obtain convincing results and agents with an interesting strategy.**

I. Introduction

Snake has to be one of the most popular games of the early 2000's and was the first game that all four of us ever played on a cellphone. This nostalgic vibe led us to select this game for the project. The concept is as follows: the player maneuvers a line which grows in length, with the line itself being a primary obstacle. The line resembles a moving snake that becomes longer as it eats food. The player loses when the snake runs into the screen border or itself. The goal is to get as long as possible before dying. In this report, we introduce different reinforcement learning approaches in order to efficiently train an agent to play the Snake game.

To begin, we found an implementation of the Snake game in `python` using `Pygame`. We made this interface compatible with a learning agent and implemented an easy SARSA method, allowing us to test different state and reward models. We then tried different learning algorithms, such as Q-learning and Deep Q-learning, leading us to start fine tuning our model's hyper-parameters with a simple grid search.

Our code can be found here: https://github.com/SnakeRL-INF581/snake-reinforcement-learning

Fig. 2: QR code to Github repository

To start training an agent, run the `main.py` file with `python3`. The `Pygame` module is required in order to visualize the game. Hyper-parameters and the learning algorithm (SARSA, Q-learning or DQL) can be changed directly inside this file. We also implemented a `grid_search.py` file in order to choose the best hyper-parameters for our algorithms. More details about executions can be found in the GitHub repository.

II. Background and Related Work

As we used Q-learning to train our agent, it is important to recall the definition of the Q-function. $Q_t(a_t, s_t)$ can be seen as the expected long term reward by taking action a_t, given state s_t and policy π. A policy gives in each state s the action a to take. The Q-function is given by the formula :

$$Q^\pi(s,a) = \mathbf{E}[R_t | s_t = s, a_t = a] \tag{1}$$

The main strategy for a learning agent is to maximize its long term reward and all our algorithms will therefore try to learn the most accurate Q-function in order to take the most appropriate action in each state. More information about rewards and the Q-function can be found in lecture IV of the course.

The two first learning algorithms that we tried to implement were SARSA and Q-learning. These two algorithms are very similar as they try to build the best Q-function, but they differ in their way of updating the Q-table at each time step. Their two update formula are given below. The main difference is that the Q-learning algorithm does not need to know the next action a_{t+1} to update its Q-function at time $t+1$. In both methods, α is the learning rate and γ the discount rate, telling how important the expected rewards are to get to the target. More information on SARSA and Q-learning can be found in lecture VI of the class.

$$\begin{aligned} Q_{t+1}(s_t, a_t) \leftarrow & \, Q_t(s_t, a_t) \\ & + \alpha(r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) \end{aligned} \tag{2}$$

▷ SARSA update

$$\begin{aligned} Q_{t+1}(s_t, a_t) \leftarrow & \, Q_t(s_t, a_t) \\ & + \alpha(r_t + \gamma \max_{a \in A(s_t)} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \end{aligned} \tag{3}$$

▷ Q-learning update

However, as the agent needs to discover enough states to learn its Q-function properly, it won't be relevant to apply those algorithms without any randomness, especially in the beginning, when the "right" decision doesn't make sense, i.e. the Q-function isn't properly trained yet. To do so, we applied the so-called ε-greedy strategy for SARSA, and Softmax strategy for Q-learning. ε-greedy means that the agent will either choose an action randomly with probability ε, or take the best decision according to its experience with probability $1-\varepsilon$. The Softmax strategy is quite different in the way that at every step, the action is chosen according to the probability derived from the Softmax of each action for a given state. As the agent learns the best strategy, better actions will be performed with higher probability.

$$a_{t+1} = \begin{cases} \text{argmax}_{a \in A(s_t)} Q_t(s_{t+1}, a) \text{ with probability } 1 - \varepsilon \\ a \text{ random with probability } \varepsilon \end{cases} \tag{4}$$

▷ Decision making with ε-greedy

$$a_{t+1} = a \text{ with probability } \frac{\exp\left(Q_t(s_{t+1}, a)/\tau\right)}{\sum_{a'} \exp\left(Q_t(s_{t+1}, a')/\tau\right)} \tag{5}$$

▷ Decision making with Softmax

Some improvements of the simple ε-greedy algorithm consist of storing individual values for ε in each state rather than having a global parameter. The method can be called ε_s-greedy.

$$a_{t+1} = \begin{cases} \text{argmax}_{a \in A(s_t)} Q_t(s_{t+1}, a) \text{ with probability } 1 - \varepsilon_{s_t} \\ a \text{ random with probability } \varepsilon_{s_t} \end{cases} \tag{6}$$

▷ Decision making with ε_s-greedy

The main issue with those methods is that the agent needs to discover every state and action to learn, which means that every state of the world needs to be explored. Therefore, we could not choose a complex environment architecture and we modeled the agent's states in a rather simple way. Moreover, to be able to truly play the Snake game, one has to be able to deal with a long snake all over the screen. In particular, trap situations (see Figure 8) are to be avoided, i.e. the player should not walk on purpose into a self-created deadlock. This can only be avoided with a wide overview of the world, which also means an increased state complexity that classic RL methods fail to deal with because of the exponential state space. This is why we tried to implement Deep Q-learning (DQL) since neural networks are capable of approximating a function (the Q-function here) by updating the weights of the model. At every step, unlike in SARSA or Q-learning, the network doesn't update a particular value of the Q-function, but only its weights. Therefore, one can choose to define a much more complex world architecture, providing more information to the agent. By calling q_w the network with weights w, our learning algorithm is as follows:

$$w \mathrel{+}= \alpha(r_t + \gamma \max_{a'}[q_w(s_{t+1})]_{a'} - [q_w(s_t)]_a)\nabla_w[q_w(s_t)]_a \tag{7}$$

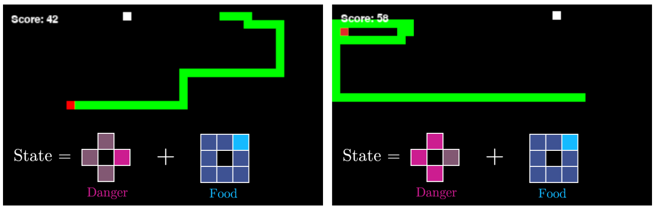


Fig. 8: Modeling of states in *Environment B*

The main issue with DQL is that it is very sensible to hyper-parameters tuning, making it complicated for the agent to learn efficiently.

III. The Environment & The Agent

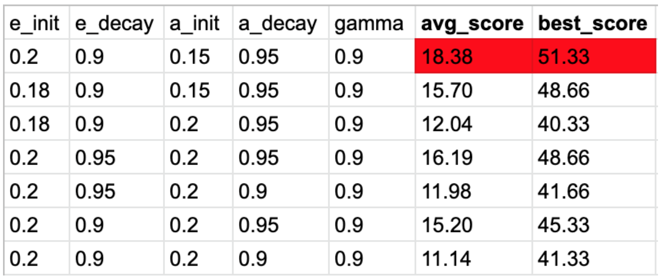
As mentioned earlier, the environment couldn't be fully observed as it would take too much time for our agent to explore the whole world and learn efficiently with SARSA or Q-learning. We therefore tried pretty much different ways of state modeling. The first one, quite intuitive, is as follows: a state is given by a binary array of size 12 indicating the direction of the snake (up, down, right, left), the immediate danger (up, down, right, left) and the location of the food (up, down, right, left). This environment is called *Environment A*. We then tried a different approach that turned out to work better: a binary array of size 12 indicating the immediate danger (up, down, right, left) and the relative position of the food, but with a little more precision: (up-left, up, up-right, right, down-right, down, down-left, left), named *Environment B*. We did not need the direction since the immediate danger gives the opposite of the direction (as long as we are not facing a wall). This model is described in Figure 8.

One of the main difficulties our agent had when training with those environments was that it wasn't able to anticipate danger, and often gets trapped by its own body. This was easily explained by the fact that it could only see the immediate danger, situated only one pixel away. To avoid this issue, we decided to give the agent a bit more information: the agent is now able to see 3 pixels away in every direction, giving it a 7×7 square vision around its head, providing a binary matrix indicating the potential danger within its vision. We also provided the relative position of the food from its head, with a 2-dimension vector. This yields a more complicated environment *Environment C*. Those states are far too complex for a SARSA or Q-learning agent, since exploring all the states would take a huge amount of time (a 40×25 world would give the snake $4 \times 40 \times 25 = 4000$ states to visit for the food alone, and up to $2^{7\times7-1}$ for the states of its vision). Exploring all of that is clearly impossible, so we only applied this model to Deep Q-learning. Finally, we also combined the snake's peripheral vision with a binary table of size 8 representing the position of the food as we did for the second environment of the SARSA and Q-learning algorithms. This defines *Environment D*.

Once the states defined, we tried to implement different rewarding strategies. The most intuitive and the very first one we tried rewarded the agent with $+100$ upon finding the food and -100 upon dying. Moreover, we observed that the efficient winning strategies consist in following straighter trajectories, since the agent has a better chance to avoid its own body by moving in an L-shape towards the food instead of zigzagging in diagonal. It means that we can encourage the agent to move a lot, but without changing its direction too often. We therefore tried a rewarding strategy as follows: we rewarded with $+30$ for eating, -30 for dying, -1 for changing direction and $+1$ otherwise. Unfortunately, due to its narrow vision, the fact to force the movement of the agent doesn't improve its score at all, which finally made us decide to keep the original rewarding strategy.

IV. Results and Discussion

We fix the game size to a 40×25 grid for SARSA and Q-learning. It is important to keep this size fixed so as to compare performance yielded by different algorithms since the agent dies more easily in a small world, but has a higher probability of finding the food. We decided to keep the classic strategy with a reward of $100/-100$ for food/dying inside *Environment B* which yields the best results with SARSA and Q-learning. After several grid searches over hyper-parameters, we managed to find our best results for SARSA and Q-learning. A part of the results for SARSA are shown in Figure 9 where we display the evolution of the average score over 100 iterations with the best hyper-parameters (cf. Figure 10).



e_init	e_decay	a_init	a_decay	gamma	avg_score	best_score
0.2	0.9	0.15	0.95	0.9	18.38	51.33
0.18	0.9	0.15	0.95	0.9	15.70	48.66
0.18	0.9	0.2	0.95	0.9	12.04	40.33
0.2	0.95	0.2	0.95	0.9	16.19	48.66
0.2	0.95	0.2	0.9	0.9	11.98	41.66
0.2	0.9	0.2	0.95	0.9	15.20	45.33
0.2	0.9	0.2	0.9	0.9	11.14	41.33

Fig. 9: Grid search (SARSA ε-greedy)

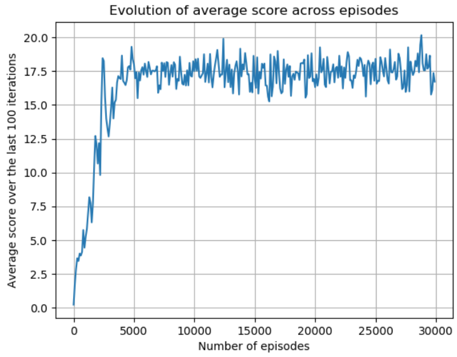


Fig. 10: Average score (SARSA ε-greedy)

In the same way we analyzed the influence of hyper-parameters on the Q-learning method. The best combination allowed us to draw the curve shown in Figure 11 with the Softmax method and in Figure 12 with the ε_s-greedy method.

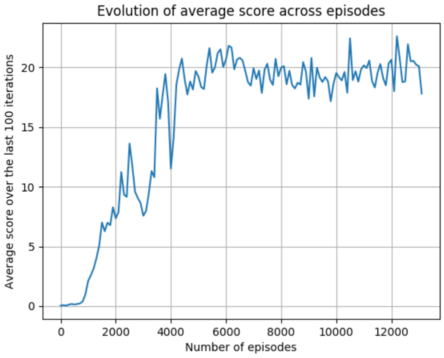


Fig. 11: Average score (Q-learning in Softmax)

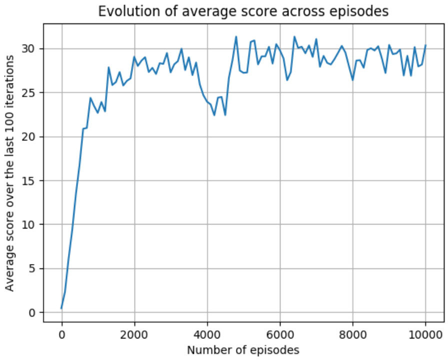


Fig. 12: Average score (Q-learning in ε_s-greedy)

These relatively simple algorithms provided convincing results, and the trained agent seems to be able to obtain an efficient strategy with these methods: in the best configuration ever tested, the best score could reach 80 with an average score around 35 over the last 100 iterations. Q-learning performed better than SARSA, which could be explained by the fact that Q-learning is more poised to do *cliff-walking*, that is, follow an optimal strategy even though it passes near high risks [6]. However, it has a very limited vision of the world around and cannot anticipate certain problems such as getting trapped by its own body. As a result, the agent can behave randomly in different learning sessions with different hyper-parameters. That's why we decided to enlarge the state space using *Environment D* and implement Deep Q-Learning methods. We used a simple dense neural network with 2 hidden layers, we took care to give the same importance to the danger and the presence of food. See figure 13 for an explanation of the architecture used for the neural network. Training is particularly complicated with this type of method. We had difficulty having the agent understand that it has to take the direction towards the food. However, the use of a smaller environment (10×10 size) allowed us to obtain some first-hand results (best score above 20, average around 10). We then placed the agent in the reference environment (40 * 25 size) and continued the training by keeping the weights of the network trained in the previous smaller environment. We were thus able to obtain convincing results, a maximum score of 56 and an average score over 100 iterations up to 23.

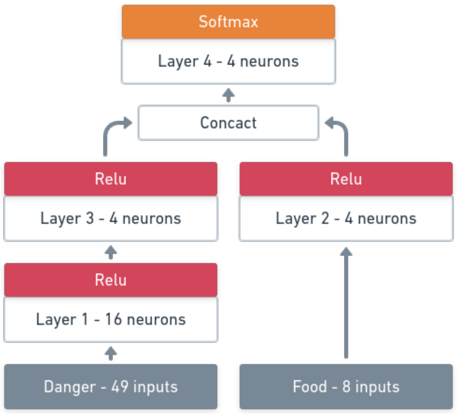


Fig. 13: The architecture of the neural network

The use of a relatively simple `PyTorch` neural network provided as much weight to information related to food as to the presence of a hazard. Furthermore, the use of the Google Cloud Platform allowed us to accelerate the apprenticeship, yet still lasted several hours for this agent. The results of DQL are good but they did not outperform the classical methods, we hope that by modifying our model and adapting our learning method we will be able to obtain better results.

V. Conclusion and Future Work

We finally got some exceptional results with basic reinforcement learning methods. These results were obtained after carrying out an effective research of the most relevant hyper-parameters. Moreover, the research led us to developing a small framework, allowing to quickly test new methods. The use of Deep Learning provided good results, while we hope to be able to make use of the full potential of this method in the coming days and improve the performance achieved today.

References

[1] Richard S. Sutton, Andrew G. Barto. Reinforcement Learning, an Introduction, *The MIT Press*, 2012.
[2] As mentioned, Lecture IV and VI - Reinforcement Learning. *INF581 Advanced Topics in Artificial Intelligence*, 2020.
[3] Christopher Watkins. Learning from Delayed Rewards, *King's College*, 1989
[4] Square Robots. AI learns to play SNAKE using Reinforcement Learning. *https://www.youtube.com/watch?v=8cdUree20j4*, 2019
[5] PyTorch. Reinforcement Learning (DQN) Tutorial. *https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html*, 2017
[6] Neil Slater on ”When to choose SARSA vs. Q-learning” *https://stats.stackexchange.com/questions/326788/when-to-choose-sarsa-vs-q-learning*, 2018