Reinforcement Learning Project
Snake game

Gladys Roch / Aurélien Houdbert

April 2021

1 Motivation and Problem Definition

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. The game of snake is a perfect instance of an RL task : the snake is the agent, the environment is the game board and the cumulative reward is how many apples the snake manages to eat before the game ends. The game ends when the snake bumbs into the boundaries of the board or into itself.

We chose this game to illustrate Reinforcement learning because it is straight-forward enough to allow us to study various RL approaches and complexe enough to have to pay attention to dimensionality of the state space and to the changing environment (size of the snake, position of the apple). In this report, we discuss the state space and how to represent it, we implement statistical RL method as well as Deep-learning RL methods and we compare their training and their final performances.

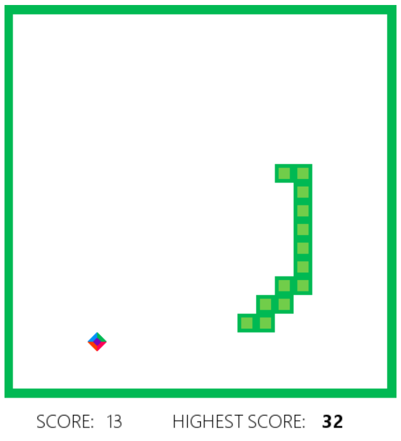

FIGURE 1 – Snake game -

2 Problem Definition

Let us define $(\mathcal{S}, \mathcal{R}, \mathcal{A})$ the set of possible states, rewards and actions. Let $R_t \in \mathcal{R}$ be the instantaneous reward obtained at time t and let γ be the discount factor with the condition $0 < \gamma < 1$. We can define the total discounted reward at time t as $G_t = \sum_{s=t+1}^{\inf} R_s \gamma^{s-1-t}$ with the resulting property $G_t = R_{t+1} + \gamma G_{t+1}$. We will also denote π a policy and $q_\pi(.,.)$ the action-value function following the policy π defined as $q_\pi(s,a) = \mathbb{E}(G_t | X_t = s, A_t = a)$ for all pairs $(s,a) \in \mathcal{S} \times \mathcal{A}$ that is the expected total discounted reward being in state s and choosing action a. Let q_* be the optimal action-value function (following the optimal policy).

3 Environment and Agent

The snake is the agent we want to teach. The environment in which the agent evolve is a 20x20 grid. The objective is to teach the snake to eat all the apples successively present in the environment. The state of the agent is defined by :

- where the head of the snake is,
- which tiles of board are occupied by the body of the snake,
- where the apple is,
- the borders of the game.

3.1 Defining the state space

The state space is all the possible configurations of the board. If we were to define a state by a matrix of size, the size of the board, the number of state to learn would be quasi-infinite. Although a neural-network might succeed in processing the full board, policy optimisation approaches will face a dimensionality issue. The policy is a matrix of size $n_A \times n_S$ where n_A is the number of possible actions and n_S is the number of states, hear n_S is close to infinite.

To counter this dimensionality issue we follow the idea of [1] and represent the state space with 11 features :

- if there's an immediate danger in the snake's proximity (right, left and straight).
- if the snake is moving up, down, left or right.
- if the food is above, below, on the left or on the right.

With this state space configuration, classic reinforcement learning algorithms are expected to work as the $n_A \times n_S$ should be of reasonable size.

3.2 Defining the reward policy

At each step of the game, the reward is given by the table 1. We do not give a reward for simply staying alive ("else") to deter the model from learning to go in circle without eating the food.

case	eat food	hit wall	hit snake	else
reward	+10	-10	-10	0

TABLE 1 – Reward policy

In section 5 we also studied another reward mechanism to assess the impact of the reward policy on the performance of the agent.

4 Methods

In this section we describe and give pseudo code for the different methods we studied in this project. We implemented the different algorithm according to [2] and all following pseudo code come from the book (it has been adapted to fit to our environment).

Except for the Monte Carlo method, we used the ϵ-greedy with decay over ϵ technique to derive a policy from the approximated action-value function $\mathcal{Q}$. This kind of policy is a simple way to find a correct balance between exploration and exploitation. This policy also has the advantage to converge towards a greedy policy resulting in a fully deterministic (no more exploration) method.

4.1 Monte Carlo

In this project we used Monte Carlo with ϵ-soft policy. This method is a famous ways to solve the problem based on averaging sample returns. To use Monte Carlo method, we simply have to ensure that no matter the decisions taken by the agent, the game will eventually come to an end (of course here we can think of combination of action leading to an infinite run but those runs have a zero probability assuming that the snake movement are random during the first games).

Algorithm 1: MC Algorithm

```
for all $(s,a) \in \mathcal{S} \times \mathcal{A}$ do
    Q(s,a) ← randomly;
    Returns(s,a) ← ∅ list;
    $\pi(a|s)$ ← random $\epsilon$-soft policy;
end

while forever do
    Generate an episode using $\pi$;
    for $(s,a) \in episode$ do
        r ← return that follows the first occurrence of (s,a);
        G ← r;
        Returns(s,a) append G;
        Q(s,a) ← avg(Returns(s,a));
    end
    for $s \in episode$ do
        $A^* \leftarrow argmax_a Q(s,a)$;
        for $a \in \mathcal{A}(s)$ do
            $\pi(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a = A^* \\ \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{otherwise} \end{cases}$
        end
    end
end
```

4.2 Sarsa

Sarsa stands short for "state-action-reward-state-action". This method is derived from equation 1. The Sarsa algorithm tries to optimize q_π and π simultaneously and its pseudo code is given in 2.

$$\begin{aligned} \mathcal{Q}(S_t,a_t) \leftarrow & Q(S_t,a_t) \\ & + \alpha(R_t + \gamma Q(S_{t+1},a) - Q(S_t,a_t)) \end{aligned} \tag{1}$$

Algorithm 2: SARSA Algorithm

```
initialize Q to 0 on terminal states and randomly
  elsewhere;
for each episode do
    Initialize $S$;
    Choose an action A from S following policy
      derived from Q;
    while $S$ is not terminal do
        Take action A and observe R and $S^{'}$;
        Choose an action $A^{'}$ from $S^{'}$ following
          policy derived from Q;
        Update Q using equation 1 ;
        $S \leftarrow S^{'}$;
        $A \leftarrow A^{'}$;
    end
end
```

4.3 Q-Learning

The Q-Learning algorithm tries to optimize the action-value function $Q(.,.)$ which will directly approximate the optimal action-value function q_* using the following update rule :

$$\begin{aligned} \mathcal{Q}(S_t, a_t) \leftarrow & Q(S_t, a_t) \\ & + \alpha(R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a_t)) \end{aligned} \tag{2}$$

Algorithm 3: Q Learning Algorithm

```
initialize Q to 0 on terminal states and randomly
  elsewhere;
for each episode do
    Initialize $S$;
    while $S$ is not terminal do
        Choose an action A from S following the
          policy derived from Q;
        Take action A and observe R and $S^{'}$;
        Update Q using equation 2;
        $S \leftarrow S^{'}$;
    end
end
```

4.4 Deep Q-Learning

When it comes to large state space, the $\mathcal{Q}$ table might become huge and thus require important memory resources (of course in our case, the state-action space is large but it can still be handled by classic reinforcement learning approaches). In order to prevent this to happen, an idea would be to replace the action-value function by a neural network able to learn this action-value function. In this case, the Q function no longer maps each action-value pairs to an expected reward but this reward is computed at each step directly by the network.

The loss we use to update the weights of the network is directly derived from equation 2.

$$\mathcal{L} = R + \gamma \max_{a'} \mathcal{Q}(s', a') - \mathcal{Q}(s, a) \tag{3}$$

where s' denotes the next state after choosing action a in state s.

Algorithm 4: Deep Q Learning Algorithm

```
initialize Q randomly;
for each episode do
    Initialize $S$;
    while $S$ is not terminal do
        Choose an action A from S following the
          policy derived from Q;
        Take action A and observe R and $S'$;
        Update network Q using the loss described
          in 3;
        $S \leftarrow S'$;
    end
end
```

The neural network we used is a simple feed forward network (figure 2) composed of 3 linear layers taking as input the 11 dimension state and that outputs the expected total discounted reward for the 3 possible actions.

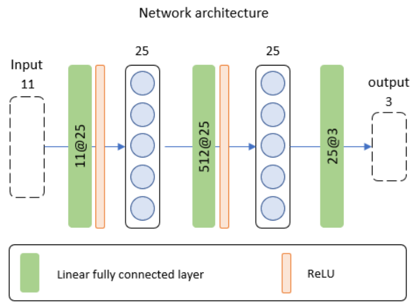

FIGURE 2 – Neural Network architecture

5 Evaluation

For each strategy, we learn the model or Q over 1000 games and then tested the learned policy for 500 games. The test statistics are presented in table 3.

Method	Average score (Std)	Best score recorded
Random	0.08 (0.3)	2
Monte Carlo	-	-
Sarsa	27.3 (10.7)	66
Q-Learning	28.5 (11.3)	68
Human (us)	44.2	56
DQL	13.83	50

Table 2 – Comparison of the average score for various learning methods. Models are trained on 1000 games, the average is computed over 500 games where the agent followed the policy learnt

We compared the methods studied with an agent following a random policy as well as with ourselves. After 1000 training games the RL-agents are able to reach or out-perform us (humans) which give good insights on the great capabilities of RL agents. Among the statistical learning strategies, Q-Learning got the highest score but the results are not significantly better than Sarsas'. The neuron-network agent does not out performed the traditional strategies. Standard tuning did not allow us the get better result with the DQL agent, we believe that much more work would be needed to build our own high performing DQL agent.

With 1000 games we were not able to train the Monte Carlo agent. Monte Carlo only updates the learnt policy at the end of each game (compared to at every steps for the other strategies), therefore the training is much time-intensive and require a lot to game to be played. This setback lead us to explore other definition of the state space and a new reward policy. These experiments are described in sections 5.2 and 5.3.

5.1 Results

The figure 3 illustrates the behavior of the SARSA, Q-Learning and Deep-Q-Learning agents when danger is detected nearby their location. The rows of the matrices in the figures represent the location of the danger and the columns are the average weight given to each action in the situation given by the rows. The darker the cell, the more weight is attributed to the corresponding action.

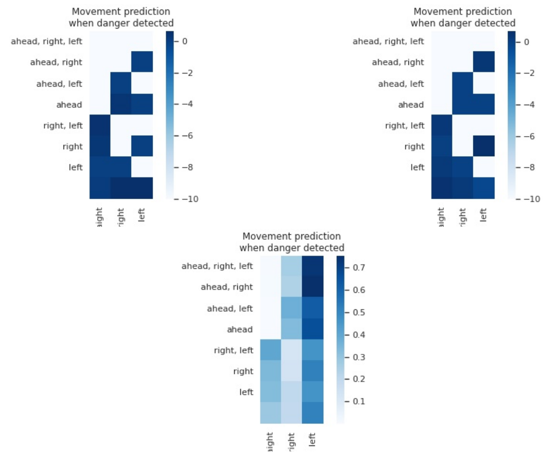

Figure 3 – Agent movement choice when danger is located nearby. **Top-left** SARSA agent, **top-right** Q-learning agent, **bottom** Deep-Q-Learning agent.

We can observe that SARSA and Q-Learning agents give almost the same predictions and that the most logical action is always the one with the highest weight. For example, on the second row, the danger is located ahead and on the right side and therefore the agent decides the go left. The Deep-Q-Learning plot is a bit less deterministic. Indeed it seems that the agent is most of the time confused and not sure of itself. It seems to have identified clearly that when danger is located ahead, it should not go ahead but for the other directions, its decision is unclear. We could say the DQL agent is a lot less assertive than the other agents.

One possible explanation could be the bad design of the neural network which could have either too much or too few layers or neurons or not complex enough. However, the DQL agent still learned something during the training as it is able to reach scores around 50 and maintaining an average score of 14. The other agent are still a lot better (by over 10 points in average and over 15 points in best score).

What is also striking is that the agents were trained only for 1000 episodes (games) and are already able to compete with humans and we are sure that not every human can beat our agents at this game.

5.2 Learning with a smaller state space

The state space of the game is defined by 11 boolean features. That makes 2048 different states. Sarasa, Q-Learning and DQL are able to learn an estimation of the optimal policy rather quickly. Monte-carlo on the other hand is much slower. We wanted to see if a smaller state space could allow monte-carlo to level with the other strategies and also how all the strategies would perform compared to the initial state definition.

To reduce the state space we decided to get rid of the information on the direction of the snake. A state is then defined by 7 features :

- if there's an immediate danger in the snake's proximity (right, left and straight),
- if the food is above, below, on the left or on the right.

This new definition of the state space yields a total of 128 states : a 16-fold reduction of the number of state to learn.

Method	Average score	Best score recorded
Monte Carlo	0.8 (3.2)	5
Sarsa	7.6 (6.2)	32
Q-Learning	4.9 (4.1)	21

Table 3 – Comparison of the average score for various learning methods with new state space definition. Models are trained on 1000 games, the average is computed over 500 games where the agent followed the policy learnt. Monte Carlo is trained over 10'000 games.

The performances recorded are worst than with the first state definition. The statistical models Sarsa and QL learn the best policy slower which explains the bad result. This is counter intuitive, we would expect the agent to learn faster a simpler state space. We believe that the training is slowed down by the missing information on the direction of the moving snake : a state encompass more situations than before and therefore the best action is harder to learn because it is a superposition of the best actions of each sub-state gathered in our new large state.

To train the Monte Carlo agent we had to push the number of training games to 10 000. Even with this extended training phase the test results are very poor : the record during the training stage was 11 and the test record is 5...

5.3 Learning with another reward policy

We also studied the impact of the reward policy on the learning of the different strategies. Table 4 presents the reward for each state. The difference with the initial reward mechanism is that, at each step, the snake is rewarded if it moves closer to the apple and is penalized otherwise. Our agent now receives an feedback on its behavior at each step of the game. The distance snake to food is the euclidean distance, although we could have used the Manhattan distance since the snake only moves along horizontal and vertical axes.

case	eat food	hit wall/ snake	closer to food	further away from food
reward	+10	-10	+1	-1

Table 4 – Reward policy

The results are shown in the following table. The learning capabilities of Sarsa and QL are

Method	Average score	Best score recorded
Monte Carlo	0.8 (1.2)	5
Sarsa	1.7 (1.9)	14
Q-Learning	0.2 (0.6)	6

Table 5 – Comparison of the average score for various learning methods with new reward policy. Models are trained on 1000 games, the average is computed over 500 games where the agent followed the policy learnt. Monte Carlo is trained over 10'000 games.

Sadly the results with the new reward strategy are very disappointing. Strategies that worked before are enable to learn and Monte Carlo still needs extensive training to start learning (barely). We believe that the -1 and +1 feedback disorient the snake : the policy fluctuates to much which doesn't allow the agent to learn efficiently.

6 Conclusion

During this project we trained different agents with different strategies to play snake game. We implemented SARSA, Q-Learning and Deep-Q-Learning and compared their performances. As a result, we obtained that SARSA and Q-Learning gave similar results and performed a lot better than Deep-Q-Learning. These agents were trained on 1000 games and are already able to give similar results as humans and even reaching higher scores. We explored to features that define the environment of the agent : the definition of the state space and the reward policy. Unfortunately, these attempts to help training Monte Carlo, only gave worst results on the strategies that worked initially. We are therefore not able to conclude on the impact of these features on the training of RL agents.

The decay rate of epsilon was tuned manually. An idea for further work would be to study the behavior of these agents as this parameter varies. We could also try to improve our Deep-Q-Learning agent by changing the structure of the network or tuning differently the hyper-parameters.

Références

[1] Mauro Comi. *How to teach AI to play Games : Deep Reinforcement Learning*. 2018. URL : `https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a`.

[2] Richard S. Sutton et Andrew G. Barto. *Reinforcement Learning : An Introduction*. 1992.