Project - Reinforcement Learning - Snake Game

Dargier Antoine - Demy Ugo

CentraleSupélec University, Gif-sur-Yvette, France
antoine.dargier@student-cs.fr
ugo.demy@student-cs.fr

Abstract. In our project, we created several Snake environments with different levels of complexity, mainly to deal with death when crossing the snake body, which appears to be the most critical constraint. We implemented three different agents to see and compare their performances: Q-Learning, Expected Sarsa agents and a Deep Q-Network. While it was quite difficult to have performant Q-Learning And Expected Sarsa agents, the Deep Q-Network appears to be also a good solution to solve this type of problem, but seems longer to train.

Keywords: Reinforcement Learning · Snake Game · Gymnasium Environment · Q-Learning · Expected Sarsa · Deep-Q Network

1 Introduction & Motivation

Reinforcement learning is a rapidly growing field of machine learning and artificial intelligence used to tackle various challenging tasks. One such task is to train an agent to play the game of Snake. This classic arcade game allows the player to control a snake that moves around a board and eats food to grow longer while avoiding obstacles and its own body.

The motivation behind this project is to apply reinforcement learning techniques to train an agent to play the game of Snake. This game has lulled our childhood; despite its simplicity, it remains one of the most famous games in the world. Moreover, the game Snake provides an ideal environment for learning such a policy since the game has a well-defined set of actions and rules, and the rewards are straightforward.

2 Environments

2.1 Overview of the two environments

First of all, to summarize the framework, the objective is that the snake learns to eat the apples with some constraints: it must not cross itself, it must not go out of the screen and it cannot turn in circles (limited number of steps to reach an apple). For each apple eaten, he grows by one square, and his objective here is to reach a length of 30 squares. In the project, we have decided to begin

with a first environment, where the snake can cross itself, and then complexify the task with all the constraints. That's why we have developed two different environments.

In the first environment, we have removed the crossing constraint which is the most difficult. The snake only has to take the shortest route to the apple. To do this, we have defined two states: IsAppleDown (1 if the apple is lower, 0 on the same line, -1 if higher) and IsAppleRight (1 if the apple is on the right, 0 on the same column, -1 if on the left). For the reward, it is:

- -1 if the snake gets closer to the apple, -3 otherwise;
- 100 if the snake eats the apple;
- 300 if the snake wins by reaching its length goal;
- -100 if the snake does not reach the apple within the number of moves allowed, which ends the game;
- -100 if the snake leaves the screen, which ends the game

For this agent, the pipeline was the following:

1. moves the snake w.r.t the action
2. if the apple is eaten, print a new apple
3. calculates the reward: if the snake gets close to the apple or not, and if it has eaten the apple
4. gives the new state, i.e the position w.r.t the apple
5. looks at if the game is finished and changes the reward if necessary

For the second environment, the difference is that we have a new constraint: the snake cannot cross itself. We use the same environment, just adding two more pieces of information to the states: if the head of the snake is along his body and the direction of that; and if the snake is along a boundary and which one.

2.2 Challenges & solutions

The conceptualization of the Gymnasium environment was the most important challenge of this project. Indeed, the training of the agents and the way they learn completely depends not only on what observation you decide to show the agent but also on how you design the reward mechanism. Regarding these 2 parts, we experienced several approaches. First, we tried to highly reward (+50) the agent only when it would eat an apple and penalize it with a -1 reward for each movement done. However, we quickly realised that because the agent would never randomly eat an apple, it would never understand the fact that doing so is highly rewarding it. Thus, we decided to add to the reward the idea of closeness to the apple by adding a Euclidean distance term between the apple and the snake's head to the reward returned for each action taken. This greatly improved our snake's behaviour by encouraging it to get closer and closer to the apple. However, we also had to increase the apple's reward otherwise, the snake would just turn around it to get a constant relatively high reward.

Another observation we made was that the more complex the state structure, the more difficult it was to train the agents and the worse the results were. This is why we considered two different environments explained above for our experiments.

3 Agents

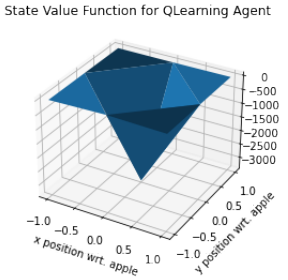
When looking at the mean cumulative reward, you have to consider the fact that the agent is penalized with the time passing by and that the goal of our agents is to get the highest reward all the time even if it is usually negative.

3.1 Q-learning Agent

Q-learning is a popular model-free, off-policy reinforcement learning algorithm. The Q-values are updated using the maximum Q-value of the next state-action pairs. The agent was also trained following an epsilon-greedy policy for 2,000 episodes which took about 1 minute. The chosen hyper-parameters were the *epsilon* equal to 0.2, and the *discount* equal to 0.8. The Q-learning agent reached a mean cumulative reward during its training of **-294.403** in the first environment, **-48.18605** in the second.

Q-learning is known to perform well in the same environments as Expected SARSA but is usually less stable than this one since it updates the Q-values with a maximum instead of an expected value. However, fine-tuning the algorithm also made us realise its sensitivity to hyper-parameters such as the learning rate, discount factor, and exploration rate. Here is the state-value plot showing the decision taken by the algorithm depending on the snake position wrt. the apple:

Fig. 1. State-Value plot - Q-Learning



On both axis, the snake is on the good line when the value is 0. The state-value function seems logical because the values are getting higher when we get close to (0,0). For (0,0), the state is a little bit more complicated to understand, because

we have a new apple appearing on the grid. As the snake is now further than the previous stage, he is penalised. We obtain the same form of function for the two agents.

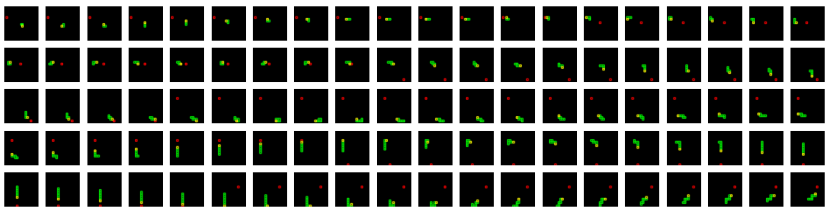
3.2 Expected SARSA Agent

The Expected SARSA agent is a model-free, on-policy reinforcement learning algorithm that estimates the Q-value function using a linear function approximator. Each one of them is then updated using the expected value of the next state-action pairs, rather than the maximum Q-value as in Q-learning. The agent was trained following an epsilon-greedy policy for 4000 episodes and took about 2 minutes. The following parameters were found to be the most appropriate ones:

- Epsilon: 0.2
- Step size: 0.7
- Discount factor: 0.8
- **Mean cumulative reward in first environment: -169.74625**
- **Mean cumulative reward in second environment: -47.4129**

You can find here the beginning of the history of the snake. In the zip file, you can find other GIFs, with all the models tested on both environments.

Fig. 2. History of the snake, with trained Expected Sarsa Agent

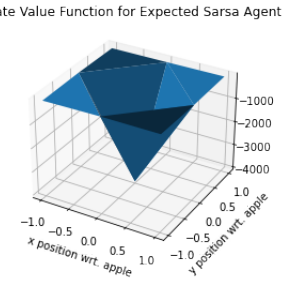


The Expected SARSA algorithm is known to perform well in environments with discrete state and action spaces and as expected, it is computationally efficient and converges to an optimal policy in a reasonable amount of time. Here is the state-value plot showing the decision taken by the algorithm:

3.3 Deep-Q Network

The Deep Q-Network (DQN) agent is a model-free, off-policy reinforcement learning algorithm that uses a neural network as a function approximator for estimating the Q-value function. The agent updates its neural network using a combination of stochastic gradient descent and experience replay. DQN is a powerful algorithm for learning complex policies in high-dimensional state and action spaces. DQN has been successfully applied to various reinforcement learning problems, including playing Atari games, and has been shown to achieve state-of-the-art performance.

Fig. 3. State-Value plot - Expected SARSA



For this agent, we used a 3-hidden-layer dense network, with each layer being respectively composed of 128, 128 and 64 neurons. It was trained for 500 episodes with the following parameters:

- Epsilon: 0.1 (init. value), 0.99999 (decay), 0.01 (minimum)
- Learning rate: 0.0001
- Discount factor: 0.9

By reaching a **mean cumulative reward of -189.85** in the first environment, the Deep Q-Network has very good performances, almost the same as Expected Sarsa. Using the same functions and ideas as Q-learning, we can see that it outperforms the Q-Learning agent a lot. However, one of the main drawbacks of the Deep Q-Network algorithm is its training time and the number of episodes required for such complex environments. Indeed, ours took $\tilde{3}0$ minutes to complete the 500 episodes. With appropriate computing resources, it could have achieved even better performances.

4 Discussions & Conclusion

We could see in this project a large part of the challenges of reinforcement learning. Indeed, by implementing our own environment, we saw that it was very important to model it well to have efficient agents. The order of the steps, the rewards, and the stopping conditions are all very important parameters that must be well understood. We have seen that the Q-Learning agent is very good to start with: it learns very quickly, but can sometimes fall into circuits where the snake goes around in circles. Despite our best efforts to help the snake always head towards the apple, it sometimes gets into loops. The Deep Q-Network model seems to perform better in all environments, but unfortunately, it takes much longer to train.

For an even more complete project, we could think about new state models to avoid crossings, for example by keeping in memory the previous movements. However, with these first models, we obtained very good results and very satisfactory agents.