Smart Snake - Reinforcement Learning

Phase 1. Selection of the state information and reward function

First Model

Selecting the attributes to represent the game state is a core aspect of this assignment for teaching the agent to learn. To create a simple approach, we started by developing a 'get state' function with four straightforward states, each representing the direction the food is relative to the snake's current position:

State 1: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is positive, i.e., the snake needs to go up.

State 2: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is negative, the snake should go down.

State 3: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is positive, the snake should go right.

State 4: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is negative, the snake should go left.

Following this initial approach, we developed a reward function that returns -2000 if the game ends and 1000 if the snake finds food. During this iteration, we experimented with various values, adjusting the penalties for crashing and the rewards for eating to find a balance. The most effective strategy showed to be the one when we increased the penalty for crashing more than the reward for eating, but without making the penalty excessively harsh.

Second Model

After seeing some deficiencies when developing the project with the states above, primarily because the snake ran into walls continuously, we decided to add another piece of information: whether the snake was close to running into the wall. To keep the state representation simple, we decided that it didn’t matter in which direction the snake was going to find a wall, it just mattered whether it was near one or not. For that reason, our new representation had 8 states.

State 1: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is positive, i.e., the snake needs to go up. The snake is not going to find a wall.

State 2: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is positive, i.e., the snake needs to go up. The snake is going to find a wall.

State 3: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is negative, the snake should go down. The snake is not going to find a wall.

State 4: If the distance on the y-axis between the food and the snake's head is greater than on the x-axis and it is negative, the snake should go down. The snake is going to find a wall.

State 5: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is positive, the snake should go right. The snake is not going to find a wall.

State 6: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is positive, the snake should go right.The snake is going to find a wall.

State 7: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is negative, the snake should go left. The snake is not going to find a wall.

State 8: If the distance on the x-axis between the food and the snake's head is greater than on the y-axis and it is negative, the snake should go left. The snake is going to find a wall.

The reward function stayed the same as in the previous iteration.

Third Model

Because we had seen that the simplest approach was not working as well as expected, we decided to increment the complexity of the state representation and the reward function.

For the states, we decided to create a “visor” that indicated the snake where the food was found. Hence, we had a different number of combinations. Additionally, we included the information about running into walls. Then, the states would be a tuple of 5 numbers, 1 or 0. [food above, food down, food left, food right, wall]. However, there are no 32 states because some combinations are not possible, like food being up and down at the same time. There are only 18 valid combinations. We made use of an additional function **translate_state** to allow this translation from the vector above to the corresponding number of the state.

Regarding the reward function, we penalized the snake with a -2000 when it died, we rewarded the snake with 1000 when it managed to eat food, and we also rewarded the snake when it moved correctly, i.e, when it moved up and the food was up, when it moved down when the food was down and so on.

Fourth Model

We decided to go back to the beginning and use simple configurations once more. However, we increased the number of states to 36. Our states were derived from a vector of flags which contained all the information.

- food_dir_x: 3 possible values:
    - 0 : the food is in the same x coordinate
    - 1 : the food is on the left side of the head ○
    2 : the food is on the right side of the head

- food_dir_y: 3 possible values:
    - 0 : the food is in the same y coordinate
    - 1 : the food is above the snake's head
    - 2 : the food is below the snake's head

- horizontal_obstacle: 2 possible values
    - 0: no horizontal obstacle (either wall or snake's body)
    - 1: horizontal obstacle

- vertical_obstacle: 2 possible values
    - 0: no vertical obstacle (either wall or snake's body)
    - 1: vertical obstacle

With the information above, we created the tuple [food_dir_x, food_dir_y, horizontal_obstacle, vertical_obstacles], and translated it into a number from 0 to 35.

For the reward function, we once again decided to use the simplest approach: +100 if the snake ate, - 100 if the snake died, -0.1 else. We also tried penalizing the snake more if it died, using -200 instead of -100.

Phase 2: Generation of the agent

For all models (First, second, third and fourth)

For the creation of ***update q table*** function we followed our code on tutorial 4,this function updates the Q-table values and is called after the agent takes an action in the given state.The function first retrieves the current Q-value for the state-action pair and then calculates the maximum Q-value for the next state across all possible actions. Then checks whether the state is a terminal state or not, if it is, the Q-value is updated only with the immediate reward.

Otherwise, the Q-value update incorporates both the immediate reward and the discounted maximum future Q-value from the next state.
The development of this function highlighted the need to introduce a terminal state, which we incorporated in subsequent iterations.

Furthermore, we put everything together on the SnakeGame file. First, we retrieve the current state information, then define the allowed actions for each direction. Afterward, we select the action to perform using the 'choose action' function. In case of training, we update the Q-table with its corresponding values.

For choosing the appropriate values of α (alpha), γ (gamma), and ε (epsilon), we utilized a combination of trial and error, guided by theoretical considerations:

Alpha (α): Alpha controls the magnitude of updates to the Q-value. A higher alpha allows the agent to learn faster by heavily weighting new rewards and rapidly adjusting from previous estimates. A lower alpha makes the learning process more conservative, where Q-values are updated more slowly, and past knowledge has a more substantial influence. The experimentation with different values can be found below in the results table.

Gamma (γ): Gamma determines the importance assigned to future rewards compared to immediate rewards. A high gamma, close to 1, encourages the agent to consider future rewards nearly as valuable as immediate ones, promoting long-term optimal strategies. A lower gamma would make the agent short-sighted, focusing mostly on immediate rewards. We studied different values of gamma to find the optimal value of the parameters. The experimentation with different values can be found below in the results table.

Epsilon (ε): Initially, a high epsilon encourages the agent to explore, which improves the chance of discovering good strategies by trying various actions.As learning progresses, epsilon decays, which means the agent gradually shifts from exploring the environment to exploiting the known, best strategies to maximize the reward. We started with a high initial epsilon of 0.9 to enhance exploration and then reduced it to 0.01 during testing phases to focus on exploitation. On both cases, we used a high value for the parameter of epsilon decay: 0.999

Additionally, we made use of the allowed actions. After analyzing the game, we discovered that if we allowed the snake to move in any direction at any time, it crashed with its body. Hence, we decided to only let the learning algorithm choose actions that would not allow for the snake to make opposite moves consecutively. That is, we wouldn't let the snake move first

up and immediately after down, or first right and immediately after left. This worked really well and solved this type of problem.

Phase 3. Enhancement of the agent

First Model

In this iteration, we hadn't managed for the snake to work with any board size, so we decided to not continue with the third phase until it worked sufficiently good in the phases before.

Second Model

Similarly as before, we decided to keep improving the algorithm before moving to doing something more difficult.

Third Model

We added the logic for the snake to study whether it will crash with its own body. Before, the snake only took into consideration whether it would crash with a wall.

Fourth Model

Our first phase model already implemented the logic for studying possible crashes with the snake's body. Hence, in this phase we just activated the growing body flag and studied how it performed. On a smaller frame size, it worked badly. This makes sense as as it got larger it had less space to maneuver and to refrain from crashing with itself or with the walls. However, when the frame size became larger, it performed much better.

Results. Quantitative and qualitative results

In the following table we summarize the results of the experimentation with the different models explained above. Some tweaks were made for each of the models, and hence we represent the different versions. Note that for model 1 and model 2 the results are not shown because it performed badly enough for us to decide not to continue with Phase 3 on these models.

In order to evaluate the performance of our models we used two metrics:

- The average number of food eaten on 200 episodes
- The average number of ticks per episode

For the first metric, the larger the value the better the model. It is good that the snake eats more food.

For the second metric, the smaller the value the better the model. We don't only want a snake to eat food, but also to do it in the most optimal way. However, we must bear in mind that for models where the average number of food eaten is higher, the average number of ticks per episode will be higher, because the snake will take more time to die. Then, in this case, a high number of ticks per episode is not bad.

Hence, to evaluate the model, the arguments below must be both applied to find the most optimal value. The chosen models for each of the phases (phase 2 and phase 3) are highlighted in the table below.

Model	Number of training episodes	Number of testing episodes	Frame Size	Growing body	Value of ε for training	Value of ε for testing	ε decay	Value of α	Value of γ	Average number of food eaten per episode	Average number of ticks per episode
3.1	50000	200	150x150	No	0.9	0.01	0.999	0.5	0.9	5.402	63.96
3.1	50000	200	150x150	No	0.9	0.01	0.999	0.15	0.7	0.315	11.115
3.1	50000	200	150x150	No	0.9	0.01	0.999	0.15	0.15	0.236	9.876
3.1	50000	200	480x480	No	0.9	0.01	0.999	0.5	0.9	13.643	471.365
3.1	50000	200	150x150	Yes	0.9	0.01	0.999	0.5	0.9	4.437	53.495
3.1	50000	200	480x480	Yes	0.9	0.01	0.999	0.5	0.9	10.357	421.72
3.2	150000	200	150x150	Yes	0.9	0.01	0.999	0.5	0.9	6.231	69.075
3.2	150000	200	480x480	Yes	0.9	0.01	0.999	0.5	0.9	7.904	327.35
4	100000	200	150x150	Yes	0.9	0.01	0.999	0.5	0.9	9.135	134.375
4	100000	200	480x480	Yes	0.9	0.01	0.999	0.5	0.9	11.95	580.91
4	5000	200	400x400	Yes	0.9	0.01	0.999	0.5	0.9	5.79	477.015

4	10000	200	150x150	Yes	0.9	0.01	0.999	0.5	0.9	0.13	90.12
4	50000	200	150x150	Yes	0.9	0.01	0.999	0.5	0.9	0.155	23.07

Model 3.1:

On this model we used the *get_state* function explained in the third iteration subsection, with a small frame size. Getting a snake that directly approaches the food and that eats an average of 5.4 per episode.

By trying different parameters of α, reducing the impact of historical data by reducing it , and decreasing γ value to increase the impact of past rewards. We prove as suspected that the model performance is degraded, eating an average of 0.315 per episode. Moreover, we reduced the gamma parameter to 0.15, and the eating accuracy decreased reaching 0.236 per episode and crashing in just 9 ticks. For these reasons the parameters of alpha, gamma and epsilon were established to be 0.5, 0.9 and 0.9 respectively in further models. When evaluating the model on an increased board size using the same Q-table trained on iteration 1 with the 18 states, the average of eating in an episode increased from 5.402 to 13.643, and the number of ticks increased proportionally.
Overall this simple model manages to find the food and avoid it a good enough number of times not to approach walls, but is not capable of learning how to avoid its tail.

Model 3.2:

In order to enhance the agent we turned the “growing body” to true. The performance slightly decreased with respect to the evaluation without body, since the chance of crashing itself were not being taken into account. This led us to modify the get_state by a slight difference, on this occasion the last element on the array “wall” was modified to also consider where the snake was about to crash itself. On a frame size of 150x150 we increased the performance compared to our previous approach increasing the eating average from 4.437 to 6.231, but when increasing the frame size to 480x480 the increase was not notable.

This model is more consistent with avoiding ending up in creating loops with its own body, however we think that having just a flag for all directions and both the body and the walls is not giving enough information to the agent to avoid this specific situation.

Model 4:

We created a last model to handle the cases where the snake was crashing with its body and we modified the get_state and reward functions as explained on the fourth iteration. We saw a significant improvement on the performance, increasing the eating average per tick in both frame sizes which led to more ticks per episode too. We noticed by observing the render

window that the behavior in this case was optimal compared to others, approaching the food straightforwardly and at minimum distances, avoiding loops. This final model manages to find food and eat several times without crashing with itself or the wall creating smart decisions.

Conclusions

Some of the key insights we have learned during the development of this practice are the following:

- The number of training episodes is key. We want many episodes so that there's space for exploration, but we need to be aware of the fact that the snake might overlearn a specific situation. Our snake learned more with 150000 episodes than with 5000, but it learned less with 150000 episodes than with 10000000. Hence, this number is not trivial and several options must be tried out before deciding the optimal value.
- The snake usually worked better with larger frame sizes because it was more difficult for it to crash.
- The value of epsilon and epsilon decay was key.
- Our snake worked better when the gamma value was higher rather than when it was lower.

- Setting the value of alpha to a value higher than 0.5 was not good for our snake. The optimal solutions occurred when alpha was 0.5.