Technical Specification

Snake reinforcement learning agent with interactive user front-end

Nicolas Oyeleye - 19359231
Ionut Eusebiu Andrici - 19702361

Supervisor - Alessandra Mileo

03/03/2022

1. Overview **3**
1.1 Glossary 3

2. System Architecture **4**
2.1. Agent 4
2.2. Player 5
2.3. Scoreboard 5
2.4. Game Scripts 5

3. High Level Design **5**
3.1 Class Diagram 5
3.2 Data Flow Diagram 6

4. Implementation **7**
4.1 User Interface 7
4.2 The Environment 7
4.3 A.I. Agent 9

5. Testing **10**
5.1 Performance Testing 10
5.2 Smoke Testing 10
5.3 Integration Testing 10
5.4 User Testing 11

6. Problem Resolution **11**
6.1 States & Memory Constraints 11
6.2 Training Time 12
6.3 Size of the board 12

7. Future Work **12**

1. Overview

In this project we developed an agent that learns how to play the game snake and get better after training using reinforcement learning. The game snake consists of a player manoeuvring a snake through the game field, trying to eat as many food items as it can avoiding the obstacles that appear in the game. Every time the snake eats these food items, its length increases and so does the intensity and difficulty. The game contains two modes. A single player mode and an agent mode. In the player mode the user can change the environment variables of the game before playing the game in which they are the one manoeuvring the snake though the game field. In the agent mode the user again has the option of changing the environment variables, but this time the agent is the one that will be playing the game. The player is able to watch as the agent plays, trains and learns the rules of the game and gradually increases its score.

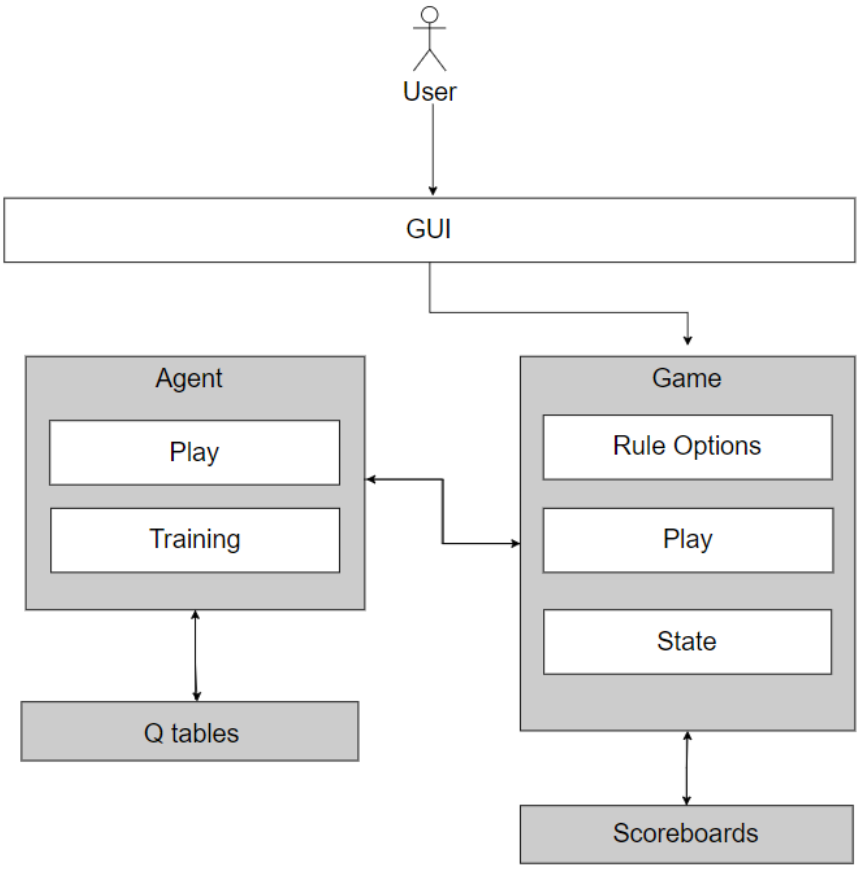
The main focus of this project was to develop the AI agent using Q-learning, which is a value based reinforcement learning algorithm, train it to maximise the score and analyse the performance of the agent in learning the game and adapting when the rules of the environment change.

The primary programming language used for the implementation of the snake game and the agent was Python and the graphical interface of the game was implemented using the Pygame library.

1.1 Glossary

AI	*Artificial Intelligence is intelligence demonstrated by machines, as opposed to natural intelligence displayed my humans*
Reinforcement Learning	*Machine learning sub-field, in which a computer's actions are taken based on the current state of the environment and the previous results of actions.*
Agent	*An entity that perceives/explore the environment and act upon it*
Q Learning	*Value-based reinforcement learning algorithm to learn the value of an action in a particular state*
Q Value	*A value given to an action in a certain state.*
GUI	*Graphical User Interface*
Pygame	*Is an external graphical and sound Python library which allows the implementation of python applications.*

2. System Architecture



2.1. Agent

The agent is the entity that will manoeuvre the snake through the game environment and act upon it. It's reasoning and decisions will be derived from Q-Learning which is a reinforcement learning algorithm. Every decision the agent takes is sent to a Q table. Any time the snake eats the food item, it is rewarded with a score of fifty. Any time the snake dies the agent is given a score of -50. Any empty cell the snake enters where nothing happens the agent is given a score of 0. The agent uses this Q table to help it get around the game field. The longer it trains the more it knows about the environment. At the start of the training the agent will be doing more exploration than exploitation. It will be using Epsilon which represents the trade off between exploration and exploitation. The longer it trains the more the agent uses exploitation as it knows more about the environment.

2.2. Player

The game contains a single player mode where the user has the chance to play the game themselves. Before playing, the user is given options on how they want the environment to be set. From here they can choose the combination of rules and setting they prefer before playing the game. The player can also watch the agent play and train. They simply click on the agent button and configure the rules.

2.3. Scoreboard

There are many combinations and configurations of the rules which means playing with no walls, borders enabled and growth set to back is a different difficulty to playing with walls, borders disabled and growth set to front. As a result of this, there are different scoreboards for every combination of rules.

2.4. Game Scripts

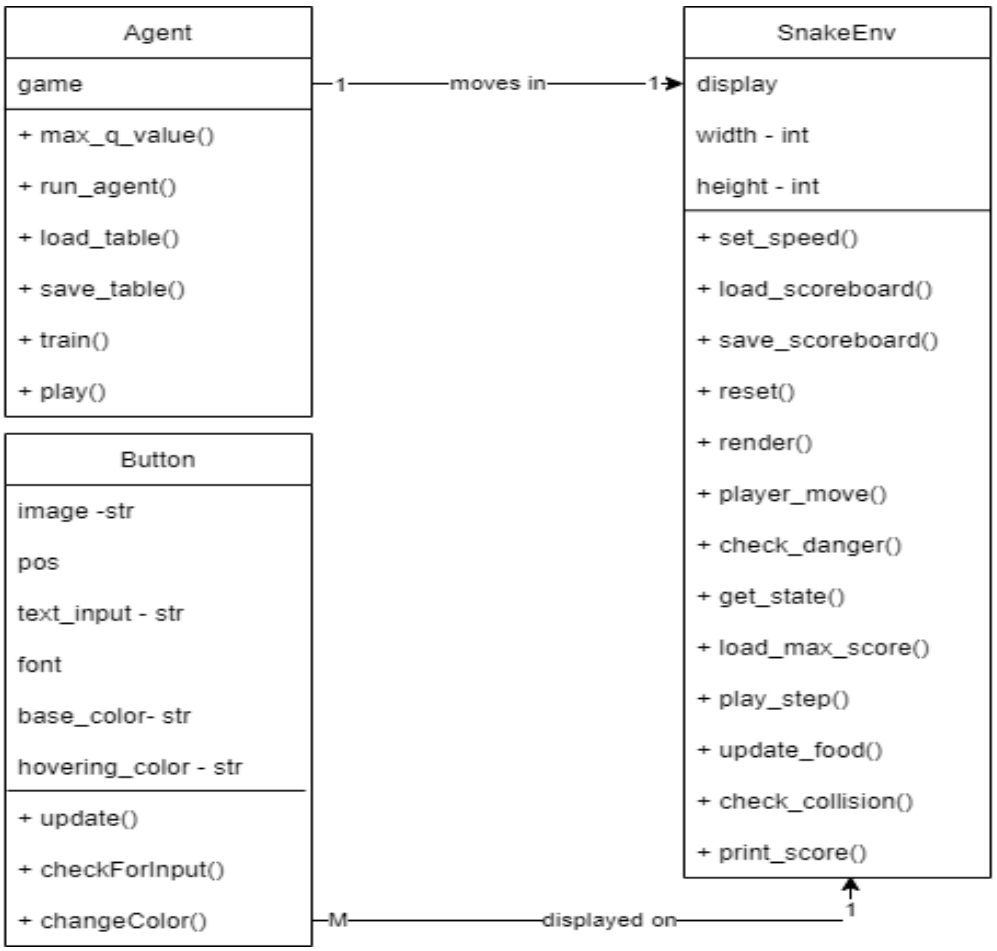
The game scripts are the files that will run the game. Both the backend and frontend will be written in Python.

The game has been converted into an executable file that can be installed on any windows device. The user has to simply download the files and click on the file named “main.exe” to get started.

3. High Level Design

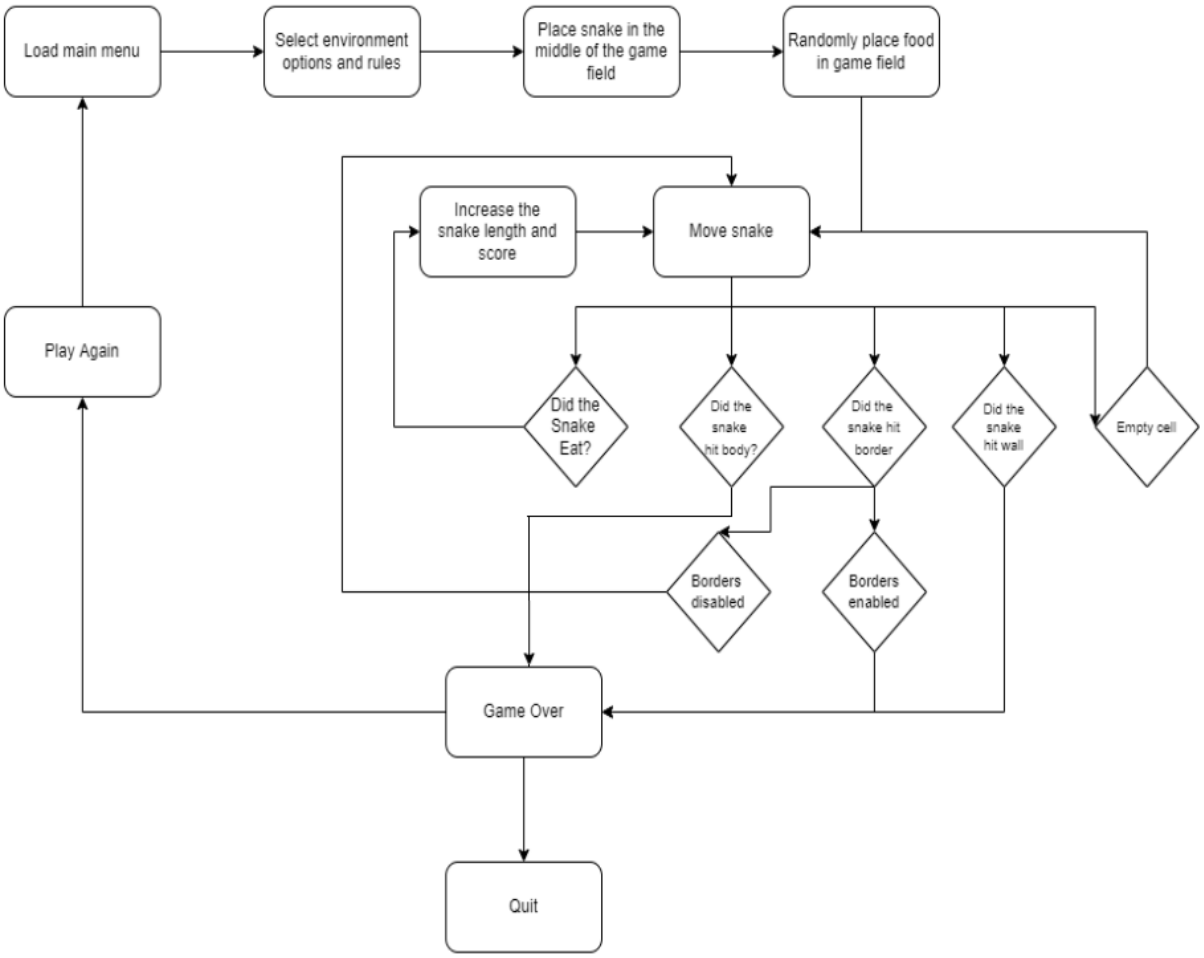
3.1 Class Diagram

This class diagram is a type of static structure diagram that describes the structure of our snake game, showing the system's classes, attributes, methods, and the relationships between them.



3.2 Data Flow Diagram

This data flow diagram (DFD) maps out the flow of information for our snake game.



4. Implementation

4.1 User Interface

The user interface is designed to allow the user to have full control. The user has the option to add and remove walls to the game, decide if the snake dies if it hits the borders of the window and if when the snake eats it will grow in the tail or in the head. These three rules can also be combined between each other. The interface was created using the set of python modules, Pygame.

4.2 The Environment

The snake environment implementation is a core element of the application and underwent major changes since its first implementation and design. In the initial design the responsibilities of moving the snake, increasing the size of the snake, updating the food coordinates, checking the collision with the edges of the screen and rendering the elements on the screen were divided between the classes Snake, Food and Environment. In the current

implementation all these responsibilities are joined together in the SnakeEnv class which allows easier maintainability.

```
44  class SnakeEnv:
45
46      def __init__(self, display, width, height):...
82
83      # reset environment
84      def reset(self):...
97
98      # render game window
99      def render(self, episode, e):...
142
143     # get player move from keyboard
144     def player_move(self):...
170
171     # play game step
172     def play_step(self, action=None, episode=None, e=None):...
236
237     # create new food coordinates
238     def update_food(self):...
257
258     # detect collision with wall or body
259     def check_collision(self, point=None):...
283
284     # calculate distance from danger
285     def check_danger(self, pt, pt_danger, x_axis=None, y_axis=None):...
304
305     # get environment state
306     def get_state(self):...
354
```

The initial implementation also did not allow an agent to play the game and refactoring of the code was needed. The play_step method is the crucial part of the application and was implemented to move the snake in the direction decided by the player or take an action as a parameter and control the snake remotely by an agent. The episode and e(psilon) parameters are only for user display.

```
178         # if agent provides action
179         if action is not None:
180             possible_dir = ['right', 'down', 'left', 'up']
181             curr_dir = possible_dir.index(list(directions.keys())[list(directions.values()).index(self.snake_dir)])
182
183             if numpy.array_equal(action, (1, 0, 0)):
184                 # no change snake goes straight
185                 new_dir = self.snake_dir
186             elif numpy.array_equal(action, (0, 1, 0)):
187                 # turn right
188                 new_dir = directions[possible_dir[(curr_dir + 1) % 4]]
189             else:
190                 # turn left
191                 new_dir = directions[possible_dir[(curr_dir - 1) % 4]]
192
193             self.snake_dir = new_dir
```

The actions that can be provided by an agent are three:

- 1,0,0 (straight)
- 0,1,0 (right)

- 0,0,1 (left)

Based on the current direction of the snake and on the action received the code above will update the current direction of the snake in the following way:

- 1,0,0: No change
- 0,1,0: RIGHT -> DOWN -> LEFT -> UP -> RIGHT -> …
- 0,0,1: RIGHT -> UP -> LEFT -> DOWN -> RIGHT -> …

4.3 A.I. Agent

The agent was implemented using Q learning, a value-based algorithm, and the Q table was implemented using a default dictionary. The default value of the dictionary is zero, which means that all the keys which are looked up in the dictionary and do not exist will not produce an error, but they are initialised at zero. The keys of the Q table are tuples of two elements, state and action and the values of such keys are called Q values. As the agent plays more episodes, the Q values will be updated based on the rewards received from the environment. The agent will always perform the action with a higher Q value in a specific state when exploiting the current knowledge. The Q value is updated using the Bellman equation.

```
92              # Bellman equation
93              optimal_action = self.q_table[(new_state, self.max_q_value(new_state))]
94              target_q = reward + self.discount_factor * optimal_action
95              self.q_table[(state, action)] += self.learning_rate * (target_q - self.q_table[(state, action)])
```

Important notions of this equation are target Q value, delta Q value, learning rate and discount factor. The discount factor was set at 0.9, which means the agent gives high consideration to future rewards instead of immediate rewards. The learning rate was set to 0.01, which defines the magnitude of the step towards the target Q or toward the solution.

The policy implemented by the agent is the ε-greedy in which the agent randomly explores with probability ε while taking the optimal action with probability 1 – ε (with $0 < \varepsilon < 1$).

```
        for step in range(self.max_steps):
            # Policy implementation: get action to play considering exploration and exploitation tradeoff
            tradeoff = random.uniform(0, 1)

            if tradeoff > self.epsilon:
                # exploitation
                action = self.max_q_value(state)
            else:
                # exploration (random action)
                action = random.choice(self.actions)
```

It can be seen above that the tradeoff is randomly calculated for each step, and if the value is higher than the value of ε, then the agent exploits its current knowledge. At the same time, if the opposite happens, the agent takes a random move and explores the environment. The ε

value starts at one and gradually decreases every episode, meaning that as the agent plays more episodes, it will also exploit more of its knowledge.

In order to make use of the knowledge accumulated, the Q tables are serialised and saved into files using the pickle module. The name of the Q table files depends on the current game's active rules.

```
39      def save_table(self, name):
40          # save q_table
41          with open(name + 'table.pkl', 'wb') as pickled_table:
42              pickle.dump(self.q_table, pickled_table)
43
44      def load_table(self, name):
45          # load q_table
46          try:
47              with open(name + 'table.pkl', 'rb') as pickled_table:
48                  self.q_table = pickle.load(pickled_table)
49          except:
50              self.q_table = defaultdict(def_value)
51
```

5. Testing

5.1 Performance Testing

While running performance tests we were watching and analysing the behaviour of the system in different scenarios. We did these tests to ensure that our system is reliable and stable. In doing these tests we refectored our source code a couple of times, each time increasing the performance.

5.2 Smoke Testing

Smoke testing was quick to execute. It’s goal was to give us the assurance that the major features of our system were working as expected. We performed smoke testing after every deployment. This was to make sure that the application was still running properly after changes were made. If any errors were caught we used the trace back errors in the command prompt to fix the problem.

5.3 Integration Testing

Integration testing was done to verify that different modules and functions were working well together. These tests required multiple functions and modules meaning they all had to be finished in order to be tested together. Integration testing was done quite regularly to ensure that newly added modules were working as expected. Errors found while performing

integration testing were mostly to do with the parameter changes so they weren't too hard to fix using the Pycharm debugger.

5.4 User Testing

User testing was very important in our suite of tests because it shows you how real users would perform and act in realistic conditions. From our observations we were able to modify the user interface to make it not only more user friendly but also interactive. To perform user testing we had a few people play the game for a couple rounds of the game, changing the rules to whatever suited them. From this we were able to prevent user errors and increase the usability while keeping the overall design consistent.

6. Problem Resolution

6.1 States & Memory Constraints

One problem we encountered was what relevant information to give the agent about the environment and the impact on memory usage. In earlier versions of our game, the snake needed to get information about the environment. To do this, we used a tuple of 11 binary values to indicate the location of the food, the current direction of the snake and the danger 1 square around the snake. The number of possible states was 6144, this meant that the agent quickly learnt what it should do, and it could implement an essential strategy when the rules of the game were the basic ones (no rules enabled). The snake, though failed, as it grew longer, to recognise the location of its own body and was ineffective with the growth game rule.

We decided to increase the state to 14 binary values, giving the agent information about the danger two squares in front, to the right and the left of the snake. With this information about the environment, the agent took slightly longer to train, but the performances were similar.

Another problem we considered was that the information given about the environment might represent more than one state. Due to the lack of information, the agent treated two states the same even when that was not the case. To try and differentiate, we introduced another piece of information to the environment: the snake's length. The snake's length was no longer a binary value, which meant that every time the snake reached a new length, it had no experience about the environment and had to start learning what to do. The training was prolonged, and because the exploration was constantly decreasing as the snake grew longer, we decided to discard this idea.

We further decided to provide information about the squares of a 9x9 matrix around the head of the snake. The matrix would indicate the squares with 1 the body of the snake, 2 the

dangers (walls or edge of the screen depending on the rules), 3 the food if it was present in that matrix and with 0 for every other square that had no reward. The matrix combined with the current direction of the snake and the location of the food (up, down, right, left), created a state with 89 values. The agent improvement was relatively slow given the number of values in the state and that we were not dealing anymore with binary values. After two days of training, the performances were acceptable and satisfactory, but unfortunately, the Q table reached 1.5GB in size, and it took more than a minute to save and load a table.

We decided to go with a more memory-efficient way to explore the game field for the snake. Instead of using this 9x9 matrix around the head of the snake, we shifted to analysing the distance from the dangers, e.g. the walls and borders. This implementation takes less memory than before, with the Q table only reaching about 100MB. The agent's performance was more or less the same given the amount of training received by the agent with both states.

6.2 Training Time

We trained the agent for long hours, using our laptops, which used the CPUs without having dedicated GPUs to train it. Given that the project had a deadline, it was impossible to train the agent to its full ability before the deadline. We noticed that the snake would sometimes get stuck in a loop, searching around the same area for a long time. Lacking time to train further and fix this issue, we added a rule that if the snake has moved 200 times and has not found the food item, the food item is randomly placed in another area. Doing this avoids these long loops.

6.3 Size of the board

The size of the board on which the agent plays can represent a problem. A large board means more states that the agent can experience, and the training requires more time. Given a large board, there is the danger that the agent interprets two different situations as the same state. Also, an agent trained in a board made to play in a bigger board will experience states never seen before. For these reasons, we decided to use a 16x16 board in which the agent trains and plays.

7. Future Work

From our experience in this project, Q learning is not suited in dealing with large amounts of data. Another aspect where Q learning is limited is when presented with a new state, the agent can not decide what to do based on previous similar experiences but it only learns on a trial and error basis. In the future we hope to develop further this project by using deep reinforcement learning and make use of neural networks to implement algorithms such as monte carlo and genetic algorithm and trying to achieve the maximum score possible.