SNAKE GAME USING Q-LEARNING

CHAPTER 1
INTRODUCTION

1.1 General

Snake game is a computer game, in which the player controls a snake to move around and collect food in a grid. In this project an AI-based agent is developed using reinforcement learning approach namely Q-learning.

In the game, the snake is allowed to pass through all the area around a 2-Dimensional environment or grid which is surrounded by walls. At each distinct interval (a time step), the snake should move forward, turn left, turn right, as the snake requires and the snake cannot stop moving. The game will be generating random piece of food that will be placed anywhere in the environment. When the snake moves towards the food and if the food is eaten then the length of the snake will be increased by one. The goal of the game is to eat as many foods without getting collide to the wall or by itself. The objective of the game is to maximize the score or length of snake. To achieve this an AI-based agent is to be developed in order to maximize our score.

1.2 Objectives

The objectives that are aimed to achieve in this project are as follows:

- To study the snake game and code a working replica of the game.
- Examining different methods that can be used to solve the game.
- Selecting an appropriate method to work on and apply on our version of game.
- Checking the performance of the method used.
- Comparing our method with other existing solutions.

1.3 Structure of Report

The rest of this report is organized as follows:

In the literature survey section, a survey of methods used to solve the snake game are listed and reviewed on their pros and cons of the said approaches. In the Background of project section, a brief description on the game, its technical details are given. Further

reinforcement learning, more specifically Q-learning is discussed. In methodology chapter, we have discussed our approach to the problem statement and its implementation, along with tools used to implement the project. The Results section, consist of the performance measured of our discussed method on the game and parameters used to obtain the results. The report is concluded in the Conclusion section, where our work is briefly summarized.

CHAPTER 2
RELATED METHODS

In this section, different approaches to solve the snake game using both non-machine learning and machine learning methods have been studied and the pros and cons of the same are also discussed.

2.1 Non-ML Techniques

The game of Snake has a trivial, unbeatable solution. Create a cyclic path that goes through every square on the board without crossing itself (this is known as a Hamiltonian Cycle), and then just keep following this cyclical path until the snake's head is as long as the entire path. This will work every time, but it wastes a lot of moves. In an NxN grid, it will take ~N^2 apples to grow a tail long enough to fill the board. If the apples appear randomly, we would expect that the snake will need to pass through half the currently open squares to reach the apple from its current position, or around $N^2/2$ moves at the start of the game. Since this number decreases as the snake gets longer, we expect that on average, the snake will need ~$N^4/4$ moves to beat the game using this strategy. This is about 40,000 moves for a 20x20 board, which will increase exponentially as board size increases.

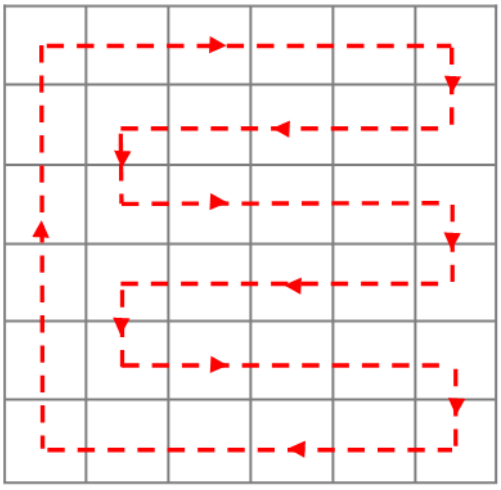

Figure 2.1 Snake Game using Hamiltonian cycle

Several optimizations on this approach, finds ways to cut off bits of the cycle without trapping the snake, so that the apple can be reached in fewer moves. This involves dynamically cutting and restitching the Hamiltonian cycle to reach the apple quickly. There are other non-ML techniques for playing snake, such as using the A* algorithm to find the shortest path to the food, but unlike the Hamiltonian Cycle approach, this is not guaranteed to beat the game.

Pros: Guaranteed to beat the game.

Cons: Is slow for large game boards, requires exponential number of moves.

2.2 Reinforcement Learning

Reinforcement learning is a field of AI where at a very basic level, reinforcement learning involves an agent, an environment, a set of actions that the agent can take, and a reward function that rewards the agent for good actions or punishes the agent for bad actions. As the agent explores the environment, it updates its parameters to maximize its own expected reward. In the case of Snake, the agent is obviously the snake. The environment is the NxN board (with many possible states of this environment depending on where the food and the snake are located). The possible actions are turn left, turn right, and keep going straight.

Figure 2.2: Reinforcement Learning

In this approach we can see its process of exploration and exploitation, happening live. In some games, the snake may die after 5 moves. This incurs a penalty, and the network will update itself to avoid similar moves in the future. In some games, the snake stays

alive for a long time and amasses a long tail, earning lots of reward. Either way, the actions are either positively or negatively reinforced to teach the snake how to play better in the future.

Pros: RL can be applied to many other tasks as well, and doesn't require supervision beyond setting up the environment and reward system. It converges faster than genetic algorithms because it can take advantage of gradient descent rather than mutating randomly.

Cons: The model's performance is dependent on what inputs are available to the network, and more inputs means more model parameters, which means longer to train.

2.3 Genetic Learning

Genetic algorithms are another popular approach to this type of problem. This approach is modelled off of biological evolution and natural selection. A machine learning model maps perceptual inputs to action outputs. An input might be the snake's distance to obstacles in the four main directions (up, down, left, right). The output would be an action like turn left or turn right. Each instance of a model corresponds to an organism in the natural selection analogy, while the model's parameters correspond to the organism's genes.

To start, a bunch of random snakes are initialized in an environment and set loose. Once all the snakes die, a fitness function selects the best individuals from a given generation. In the case of Snake, the fitness function would just pick snakes with the highest scores. A new generation is then bred from the best individuals, with the addition of random mutations. Some of these mutations will hurt, some will not have any effect, and some will be beneficial. Over time, the evolutionary pressure will select for better and better models.

Pros: Once the model is trained, predicting the next move is fast.

Cons: Can be slow to converge because mutations are random. The performance is dependent on the inputs available to the model. If the inputs only describe whether there are obstacles are in the immediate vicinity of the snake, then the snake isn't aware of the "big picture" and is prone to getting trapped inside of its own tail.

CHAPTER 3
BACKGROUND OF THE PROJECT

In the Background of project section, a brief description on the game, its technical details are given. Further reinforcement learning, more specifically Q-learning is discussed.

3.1 Snake Game

Snake game is a computer game, in which the player controls a snake to move around and collect food in a grid. In the game, the snake is allowed to pass through all the area around a 2-Dimensional environment or grid which is surrounded by walls. At each distinct interval (a time step), the snake should move forward, turn left, turn right, as the snake requires and the snake cannot stop moving. The game will be generating random piece of food that will be placed anywhere in the environment. When the snake moves towards the food and if the food is eaten then the length of the snake will be increased by one. The goal of the game is to eat as many foods without getting collide to the wall or by itself. The objective of the game is to maximize the score or length of snake.

Our Snake Game project is written in Python. GUI is implemented using pygame library. In this implementation of the Snake game using Reinforcement Learning, we are going to use Q Learning instead of the Traditional Machine Learning approach as they work on available data but in our case, we don’t know what is the best action to take at each step of the game.

3.2 Reinforcement Learning

Reinforcement Learning is an aspect of Machine learning where an agent learns to behave in an environment, by performing certain actions and observing the rewards/results which it get from those actions.

The Reinforcement learning is one of the most intriguing technology in Machine Learning, which learns the optimal action in any state by trial and error in similar way like we humans learn — by trial and error. As the agent explores the environment, it

updates its parameters to maximize its own expected reward. In the case of Snake, the agent is obviously the snake. The environment is the NxN board (with many possible states of this environment depending on where the food and the snake are located). The possible actions are turn left, turn right, and keep going straight.

Based on our state current S0, the RL agent takes an action A, say — our RL agent moves right. Initially, this is random as it progresses it uses exploration and exploitation trade off in reinforcement learning. Exploration is all about finding more information about an environment, whereas exploitation is exploiting already known information to maximize the rewards. Now, the environment is in a new state S1. The environment gives some reward R or penalty based on this new state S1.

Q-learning

Q-Learning is an example of a reinforcement learning technique used to train an agent to develop an optimal strategy for solving a task, in which an agent tries to learn the optimal policy from its history of interaction with the environment.

In Q-learning we use Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state. In the Q-Table, the columns are the actions and the rows are the states.

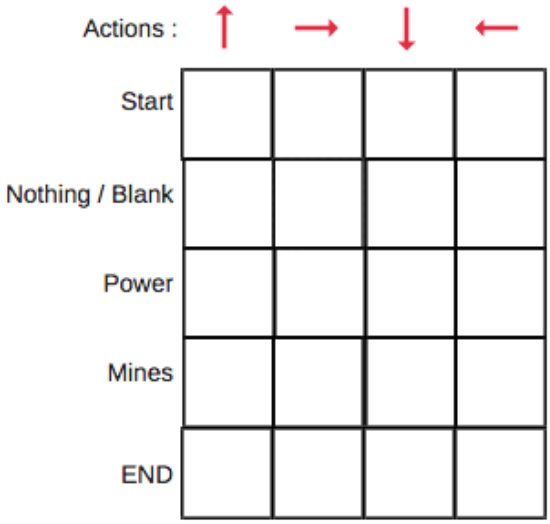

Figure 3.1: Sample Q Table

Each Q-table score will be the maximum expected future reward that the robot will get if it takes that action at that state. This is an iterative process, as we need to improve the Q-Table at each iteration. To learn each value of the Q-table, we make use of the Q-Learning algorithm.

The algorithm, has a function that calculates the quality of a state–action combination:

$$Q: S \times A \rightarrow R$$

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action At, observes a reward Rt, enters a new state St+1 (that may depend on both the previous state St and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)}^{\text{temporal difference}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

where Rt is the reward received when moving from the state St to the state St+1, and α is the learning rate ($0 < \alpha \leq 1$).

Q new (st, at) is the sum of three factors: (1 − α) Q (st , at): the current value weighted by the learning rate. Values of the learning rate near to 1 make the changes in Q more rapid. α rt: the reward rt = r (st , at) to obtain if action at is taken when in state st (weighted by learning rate). αγ max a Q (st + 1 , a): the maximum reward that can be obtained from state st + 1 (weighted by learning rate and discount factor)

An episode of the algorithm ends when state st + 1 is a final or terminal state. However, Q-learning can also learn in non-episodic tasks (as a result of the property of convergent infinite series). If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states sf, Q (sf , a) is never updated, but is set to the reward value r observed for state sf . In most cases, Q (sf , a) can be taken to equal zero.

1. Learning Rate (α)

The learning rate or step size determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

2. Discount factor (γ)

The discount factor γ determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e., rt, while a factor approaching 1 will make it strive for a long-term high reward.

CHAPTER 4
METHODOLOGY

4.1 Tools Used

1. **Python**, a free and open-source object-oriented programming language, draws attention with its simple syntax and dynamic structure. In Python, it's very easy to write code and analyse code. Another advantage is that it has the advantage of extensive documentation (books, internet sites, forums, etc.). In addition to all these advantages, it works in concert with many libraries which "machine learning" applications can be done. In this context, Python3.6has been chosen to be used in this work, because of many of the advantages it provides.

2. **Pygame** is a set of Python modules designed for writing video games. Pygame adds functionality on top of the excellent SDL library. This allows you to create fully featured games and multimedia programs in the python language.

3. **Matplotlib** is a library that runs on Python, allowing visualization of data. This library is used to create graphs used in the study.

4.2 Implementation

Implementing of Snake game using Q-Learning consists of Creating Environment and the agent, Reward and action, defining states, Q-Algorithm, and GUI to visualize the game. These have been discussed further in this section:

Creating Environment and Agent:

The Snake game environment, shown in figure is the game implemented using custom code that closely matches snake game. This allows for full control over the size of the board and rules of play. The reward function was chosen to be 1 million if the snake completes the game. The game ends when the snake collides with its own body or the

walls. Further a time penalty was introduced to avoid loops. In order to facilitate fast evaluation, and to increase the probability of the snake finding food a small game board of 5x5 were used, and required no pre-processing.

PEAS description of Game:

Performance: Score or Max length attained in the game

Environment: 10x10 grid consisting of randomly spawned apple

Actuators: Snake Movement based on Q-table (Left, Right, Forward)

Sensors: Inputs based on state of the snake (e.g., location of food, wall, tail etc.)

Agent: Snake

Rewards: Rewards are the enforcement provided to the agent to train it. That is, if the snake grabs a food, give it a reward, if the snake dies, the reward is negative. The agent will try to maximize the reward and will take action accordingly.

Snake eats an apple	length(snake)^2
Snake does nothing	-time interval
Snake dies (hit the body or wall)	-1000
Snake completes the game	1Million

Actions: Actions are performed by the agent in the environment as in our case it can choose between going forward (1), right (2), or left (0). To maximize the reward, the agent will try to take optimal action based on the algorithm.

Defining Q states:

In our game 2049 number of states are possible which are defined by the percept/vision of the snake as:

Vision 1: presence of a wall/tail in the immediate surroundings (left forward right). For each direction it may have a possible 2^3 = 8 number of combinations.

Vision 2: presence of a tail in 3 directions (left forward right). For each direction it may have a possible 2^3 = 8 number of combinations.

Vision 3: relative apple location in 8 quadrants. For each direction it may have a possible 8 number of combinations.

Vision 4: last turn (left right). For each turn it may have a possible 2 combinations.

Vision 5: length of the snake (short long). Here it may have a possible 2 combinations, defined by if the snake size is more than one-third of grid size or not.

Hence the possible number of states of our game will be 8 * 8 * 8 * 2 * 2 + 1 = 2049 states.

Q-Algorithm:

In Q-learning we use Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state. In the Q-Table, the columns are the actions and the rows are the states.

To learn each value of the Q-table, we make use of the Q-Learning algorithm.

The algorithm, has a function that calculates the quality of a state–action combination:

$$Q: S \times A \rightarrow R$$

Initially, Q is initialized to 0. Then, at each time t the agent selects an action At, observes a reward Rt, enters a new state St+1 (that may depend on both the previous state St and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \Bigg)}^{\text{temporal difference}} \underbrace{- Q(s_t, a_t)}_{\text{old value}}$$

$$\text{new value (temporal difference target)}$$

where Rt is the reward received when moving from the state St to the state St+1, and α is the learning rate ($0 < \alpha \leq 1$).

Algorithm 1: Epsilon-Greedy Q-Learning Algorithm

Data: α: learning rate, γ: discount factor, ϵ: a small number
Result: A Q-table containing Q(S,A) pairs defining estimated optimal policy π^*

```
/* Initialization                                          */
Initialize Q(s,a) arbitrarily, except Q(terminal,.);
Q(terminal,.) ← 0;
/* For each step in each episode, we calculate the
   Q-value and update the Q-table                          */
for each episode do
    /* Initialize state S, usually by resetting the
       environment                                         */
    Initialize state S;
    for each step in episode do
        do
            /* Choose action A from S using epsilon-greedy
               policy derived from Q                       */
            A ← SELECT-ACTION(Q, S, ϵ);
            Take action A, then observe reward R and next state S';
            Q(S, A) ← Q(S, A) + α [ R + γ max_a Q(S', a) - Q(S, A)];
            S ← S';
        while S is not terminal;
    end
end
```

Figure: Pseudo Code of Q-learning Algorithm

Learning Rate (α)

The learning rate or step size determines to what extent newly acquired information overrides old information. Alpha is a real number between zero and one. If we set alpha to zero, the agent learns nothing from new actions. Conversely, if we set alpha to 1, the agent completely ignores prior knowledge and only values the most recent information. Higher alpha values make Q-values change faster.

Discount factor (γ)

The discount factor γ determines the importance of future rewards. Gamma is a real number between 0 and 1. If we set gamma to zero, the agent completely ignores the future

rewards. Such agents only consider current rewards. On the other hand, if we set gamma to 1, the algorithm would look for high rewards in the long term. A high gamma value might prevent conversion: summing up non-discounted rewards leads to having high Q-values.

Epsilon Decay

The epsilon parameter introduces randomness into the algorithm, forcing us to try different actions. This helps not getting stuck in a local optimum. If epsilon is set to 0, we never explore but always exploit the knowledge we already have. On the contrary, having the epsilon set to 1 force the algorithm to always take random actions and never use past knowledge. Usually, epsilon is selected as a small number close to 0.

Action Selection

The target of a reinforcement learning algorithm is to teach the agent how to behave under different circumstances. The agent discovers which actions to take during the training process. The agent can choose to explore by selecting an action with an unknown outcome, to get more information about the environment. Or, it can choose to exploit and choose an action based on its prior knowledge of the environment to get a good reward.

The concept of exploiting what the agent already knows versus exploring a random action is called the exploration-exploitation trade-off. When the agent explores, it can improve its current knowledge and gain better rewards in the long run. However, when it exploits, it gets more reward immediately, even if it is a sub-optimal behavior. As the agent can’t do both at the same time, there is a trade-off.

CHAPTER 5
RESULTS

The Results section, consist of the performance of our discussed method on the game and parameters used to obtain the results.

Parameter Selection:

The parameters alpha, gamma, and epsilon were selected by performing the training on a set of learning rate, discount factor, epsilon decay parameters, over specific intervals, for a grid size of 5*5 and 10*10.

According to the training results, we settled on $\alpha = 0.2$, $\gamma = 0.8$, $\epsilon = 0.2$, ran for epoch=100, batch size=100, i.e., 10 thousand iterations.

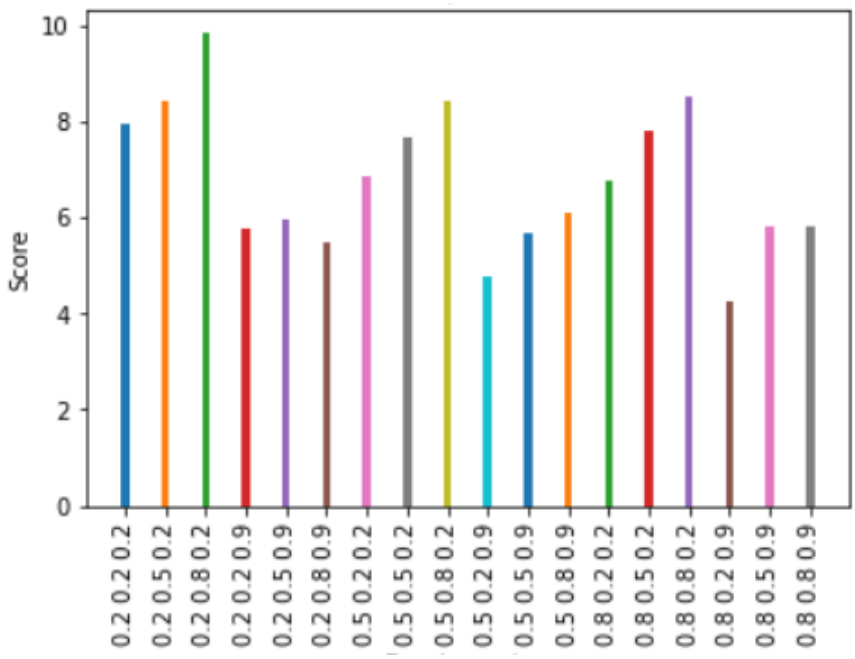

Figure: Average training score vs parameters for 5x5 grid

The above graph shows the training average score obtained by the snake over their parameters used during training. The value [0.2 0.2 0.2] represents the parameters alpha, gamma and epsilon respectively. This process was done for alpha = [0.2,0.5,0.8],

gamma= [0.2,0.5,0.8] and epsilon= [0.2,0.9] giving us 18 possible combinations of parameters.

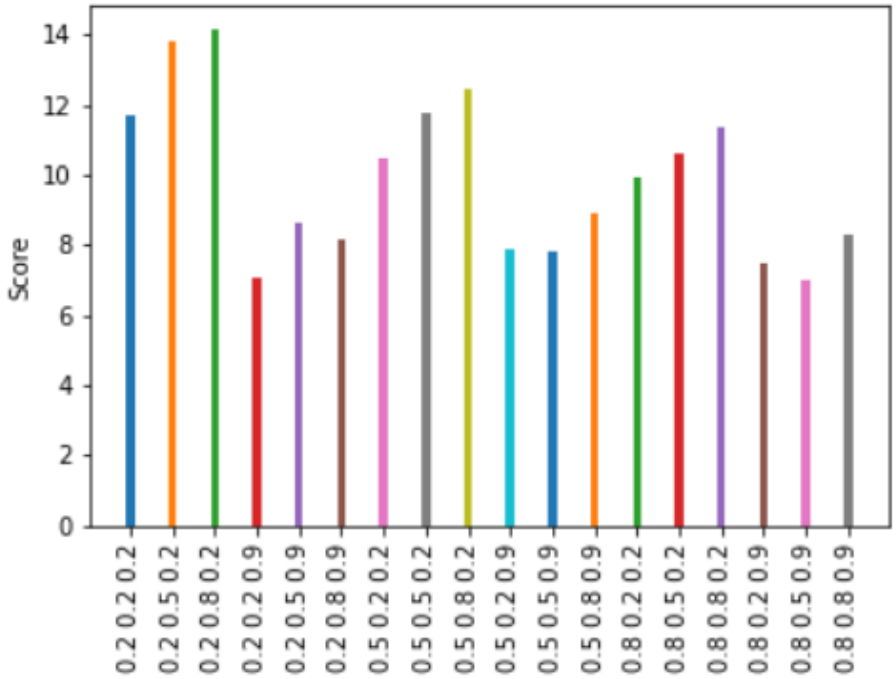

Figure: Average training score vs parameters for 10x10 grid

The best average score can be seen for parameter alpha=0.2, gamma=0.8, epsilon=0.2, and its corresponding training graph is shown below.

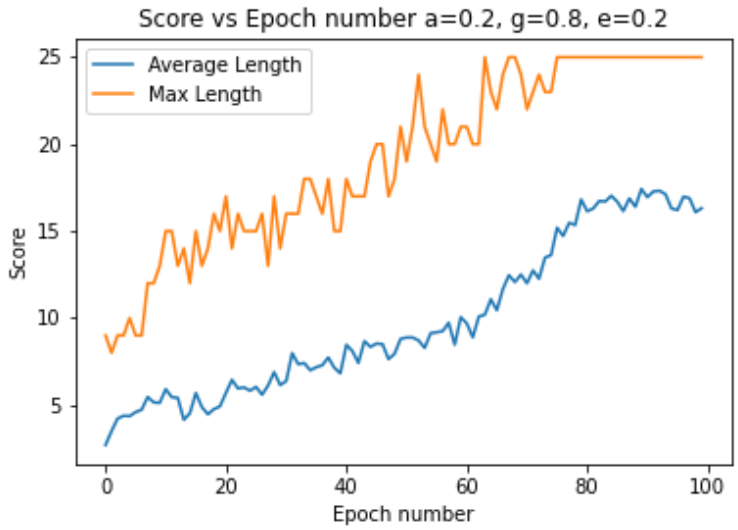

Figure: Score vs Epoch for alpha=0.2, gamma=0.8, e=0.2 (5x5)

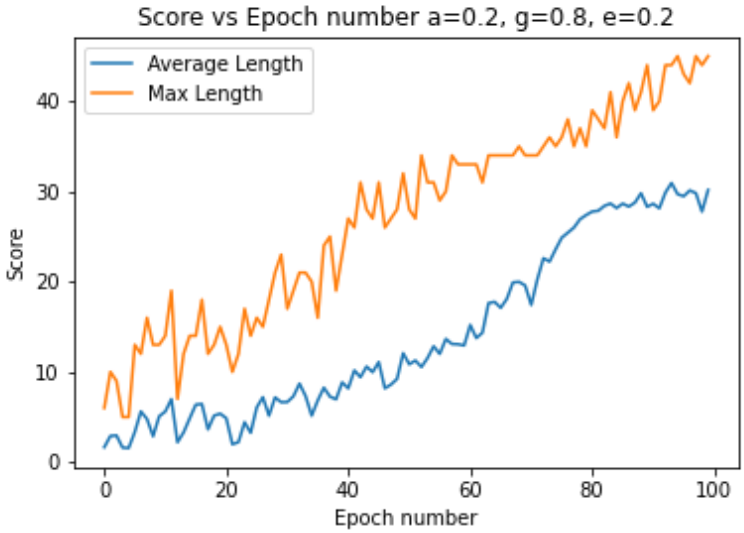

Figure: Score vs Epoch for alpha=0.2, gamma=0.8, e=0.2 (10x10)

Output:

The graphical interface for visualising the snake game was done using pygame library for python, which allowed us to demonstrate the game for the trained model.

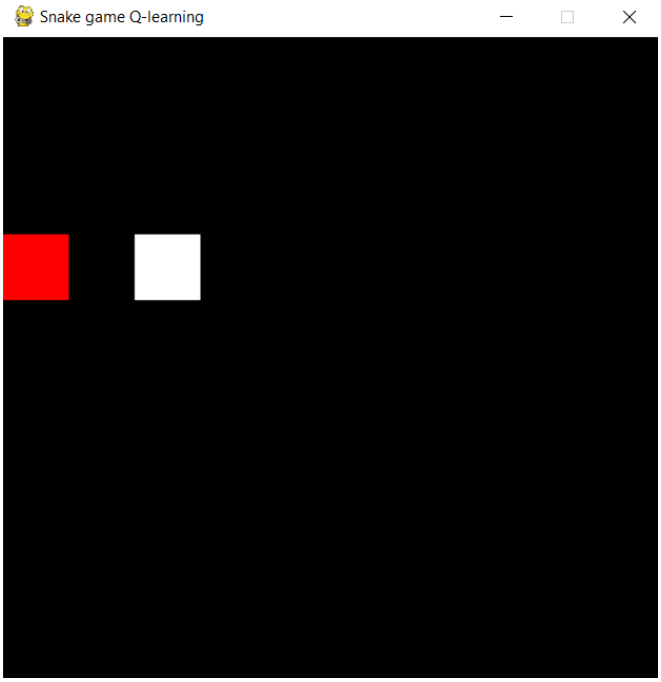

Figure: Snake game graphical interface

The output screen consists of a square game board of defined size, in our case 5x5 board was used, the apple or food denoted by red coloured dot and snake of increasing length denoted by white boxes.

The testing for the above defined parameters was done for 50 iterations and the scores were tabulated as shown:

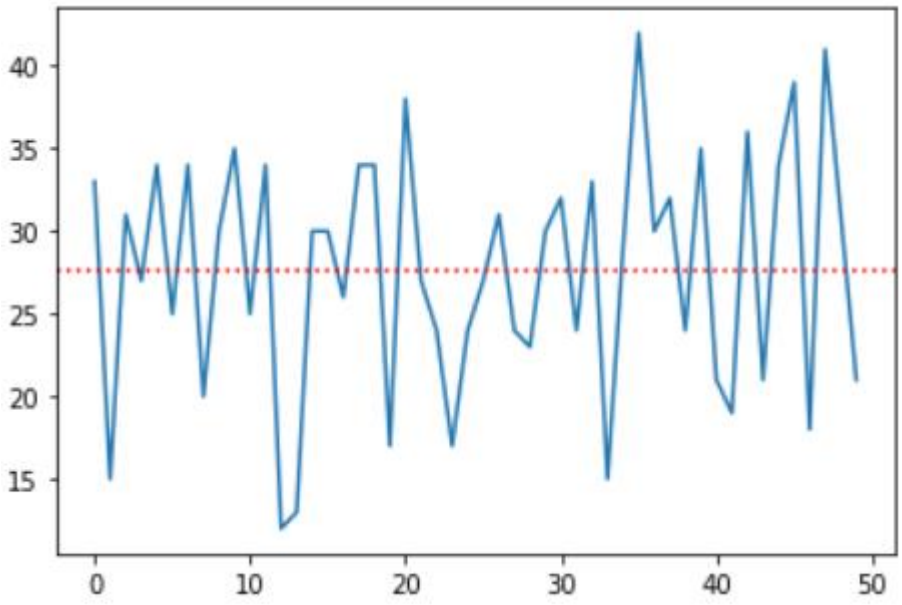

Figure: Game scores for testing

CHAPTER 6
CONCLUSION

The project, Snake game using Q-learning was aimed to make an AI agent that can play the game intelligently, by making using of reinforcement learning technique called Q-learning. We studied the different aspects of the q-learning and the application of it in a game environment. For, this we used the pygame library to create a graphical interface of the game, where the agent was trained on over a different hyperparameters and the results were plotted.

Through this project we were able to use our knowledge on reinforcement learning and use it to solve a game-based problem. In future we can implement deep learning-based reinforcement techniques, and also explore other recent algorithms, to solve the game.

CHAPTER 7
REFERENCES

Reference Links:

Almalki, Ali Jaber, and Pawel Wocjan. "Exploration of Reinforcement Learning to Play Snake Game." In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 377-381. IEEE, 2019.

Ma, Bowei, Meng Tang, and Jun Zhang. "Exploration of Reinforcement Learning to SNAKE." (2016).

https://en.wikipedia.org/wiki/Q-learning

https://towardsdatascience.com/teaching-a-computer-how-to-play-snake-with-q-learning-93d0a316ddc0

Pygame: https://www.pygame.org/docs/

Code Link:

https://drive.google.com/file/d/1if_IpBSiZjsBgVjR4M7awAislBqrE0TC/view?usp=sharing