REPORT

Submitted by:
Asmita Rani(M22CS059)
Jash Patel(M22CS061)

Advanced AI Assignment 4: Deep RL

Part 1:

The given 4 codes represent a simple game that consists of an agent, environment, actions, and observations.

1) Agent:

The agent in this game is a player-controlled entity, represented by the `PlayerEntity` object in the `GameState` class. The goal of the agent is to reach the goal location while avoiding enemies that move randomly around the environment.

2) Environment:

The game environment is represented by the `GameState` class, which maintains the current state of the game. The environment includes the player entity, enemy entities, the goal location, and the game boundaries.

3)Actions:

The actions available to the agent are represented by the `GameActions` enum, which includes `No_action`, `Up`, `Down`, `Left`, and `Right`. The agent can choose to move up, down, left, or right, or choose to take no action.

4)Observations:

The observations available to the agent are represented by the `GameObservation` enum, which includes `Nothing`, `Enemy_Attacked`, and `Reached_Goal`. The agent receives an observation after each action, indicating whether it has collided with an enemy or reached the goal.

A suitable reward function for this game could be as follows:

If the agent collides with an enemy, it receives a negative reward of -64.

If the agent reaches the goal, it receives a positive reward of +32.

Otherwise, the agent receives a small positive reward of +10 to encourage it to reach the goal quickly.

This reward system works well in this game since it encourages the player to avoid obstacles and get to the objective as quickly as possible. While the positive reward for achieving the goal motivates the agent to keep moving in that direction, the negative reward for colliding with an opponent encourages the agent from taking unnecessary risks. The agent is motivated to prioritize achieving the goal above taking unrelated activities by the small negative payoff for failing to do so quickly enough.

Part 2:

- For game seed we take

  ```
  GAME_SEED = 1322319061 # M22CS061 --> M 13, C 3, S 19 = 1322319061
  ```

- Here we use Q Learning algorithm for learning
- The GetAction method is responsible for selecting the best action at a given state. It implements an epsilon-greedy exploration policy. If a randomly generated number is less than epsilon, the algorithm selects a random action. Otherwise, it selects the action with the highest Q-value.
- The TrainModel method trains the Q-learning model using the Q-learning algorithm. The method loops through a fixed number of epochs, and for each epoch, it initializes a new game state object, sets the done flag to False, and loops until the game is over. It then selects the current action using the GetAction method, updates the game state based on the action taken, determines the reward for the current state-action pair, and updates the Q-value for the previous state-action pair using the Q-learning update rule.
- We train model for 3000 epochs and 0.0001 learning rate and 0.6464 as gamma.

Result:

Terminal

Welcome to AI Assignment 4 program!

1. Run game on your AI model

2. Try the game by controlling with keyboard

Enter your choice: 1

AI controller initialized!

Now training...

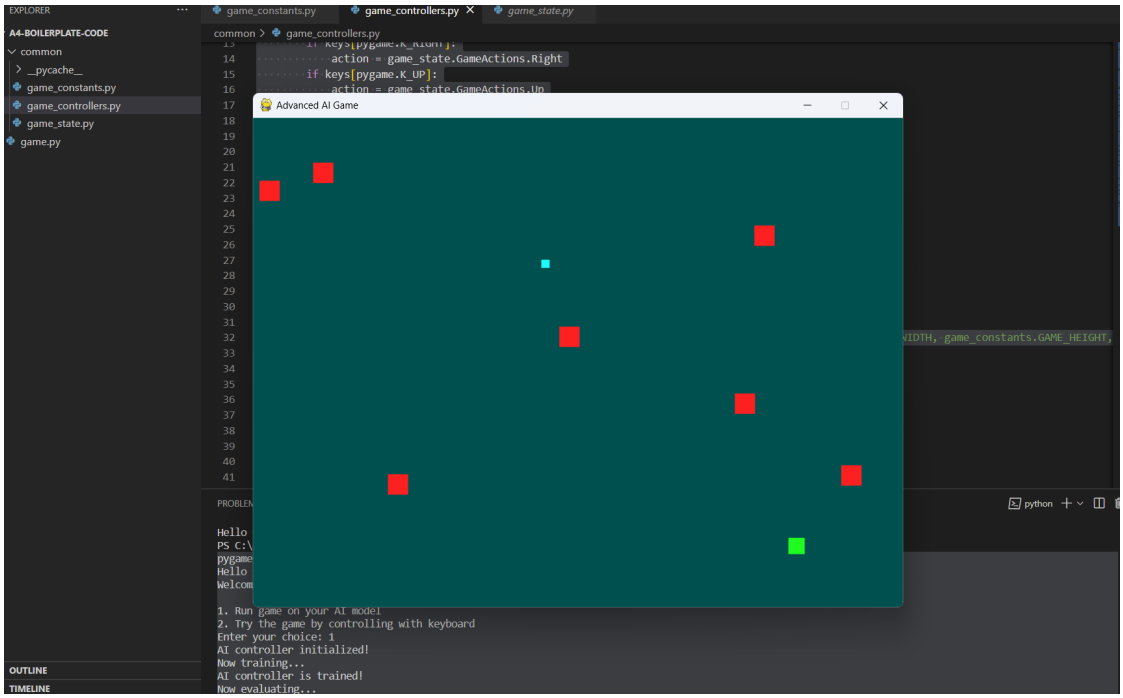
AI controller is trained!

Now evaluating...

On Evaluation, player died 3132 times, while reaching the goal 11 times

Would you like to see how this model performs on the game (y/n)?y

Game window



Part 3:

GAME_SEED select randomly

- Here we use a random module for game_seed to select randomly and observe tree results.
    - On Evaluation, player died 3010 times, while reaching the goal 9 times
    - On Evaluation, player died 2713 times, while reaching the goal 9 times
    - On Evaluation, player died 2945 times, while reaching the goal 11 times
- And roll number as game_seed we get
    - On Evaluation, player died 3132 times, while reaching the goal 11 times

Taking game seeds randomly affects player performance. Sometimes it is better and sometimes it is worse than roll number game seed. For best results in each random seed we need to perform hyperparameter tuning.

Change the dimension of game

GAME_WIDTH = 800 to 1000

GAME_HEIGHT = 600 to 800

On Evaluation, player died 1787 times, while reaching the goal 6 times

GAME_WIDTH = 800 to 600

GAME_HEIGHT = 600 to 800

On Evaluation, player died 3128 times, while reaching the goal 7 times

In dimension change we get poor performance. To get better we need to change hyperparameters.

Change in Goal size

Goal size 10 to 18

On Evaluation, player died 3342 times, while reaching the goal 16 times

Goal size 10 to 32

On Evaluation, player died 2903 times, while reaching the goal 15 times

For a smaller number of goals it performs well and for large it gives low goal reaching states.

Change in Enemy count

ENEMY_COUNT 7 to 3

On Evaluation, player died 1293 times, while reaching the goal 7 times

ENEMY_COUNT 7 to 24

On Evaluation, player died 9518 times, while reaching the goal 13 times

Here we completely show that for more enemy count it give very poor results.

Change in Game Friction

Game Friction 0.05 to 0.01

On Evaluation, player died 2785 times, while reaching the goal 19 times

Game Friction 0.05 to 0.1

On Evaluation, player died 3096 times, while reaching the goal 4 times

For low game fiction it gives best performance but for high value of game friction it performs poorly. Therefore, low game friction game performance is best.

Change in FPS

FPS 30 to 15

On Evaluation, player died 3132 times, while reaching the goal 11 times

FPS 30 to 60

On Evaluation, player died 3132 times, while reaching the goal 11 times

FPS 30 to 120

On Evaluation, player died 3132 times, while reaching the goal 11 times

FPS 30 to 10

On Evaluation, player died 3132 times, while reaching the goal 11 times

There is no change in game performance with respect to FPS.