Assignment 4 - Deep RL

- Chakshu Anup Dhannawat(B20AI006)
- Priya Sahu (B20AI031)

We have set the game seeds according to our roll numbers.

<u>Part I:</u> Describe the game and reward function.

The game is a variation of the classic snake game, where an agent is required to move towards a randomly generated food item while avoiding randomly moving enemies on the game board. The game is played on a two-dimensional grid, with the agent represented as a single cell and the enemies as multiple cells moving randomly across the board. The agent's objective is to consume the food item, represented by a single cell, to increase its score. However, if the agent collides with an enemy, the game is over.

Below is a detailed description:

Game:

- The game is a variation of the classic snake game.
- The objective is for the agent to move towards a randomly generated food item while avoiding randomly moving enemies on the game board.
- The game is played on a two-dimensional grid.

Agent:

- The agent is represented as a single cell on the game board.
- The agent's goal is to maximize its score by consuming as many food items as possible while avoiding collisions with enemies.
- The agent has four possible actions to choose from: move up, move down, move left, or move right, and do nothing.
- The agent's actions consist of moving in one of five possible directions.

Environment:

- The environment is represented as a two-dimensional grid.
- Each cell on the grid can be empty, occupied by the agent, an enemy, or a food item.
- The enemies move randomly in any direction on the game board.

- The food item is generated randomly on the game board, but not within the agent or enemy cells.

Actions:

- The agent has five actions: move up, move down, move left, move right, and do nothing.
- The agent can choose one action at each time step to determine the direction of its movement.
- If the agent collides with an enemy agent or goes out of bounds, the game ends.

Observations:

- At each time step, the agent receives observations about the game state.
- There are three possible observations:
    1. If the agent has reached the goal, represented by 1.
    2. If the agent collides with an enemy, represented by -1.
    3. If nothing happens, represented by 0.
- This grid contains information about the state of the game, including the agent's position, the enemy positions, and the location of the food item.

Reward:

For the Q-table implementation, the rewards were set as follows:

- **If enemy attacks: Reward of -200**
- **If goal is reached: Reward of 100**
- **If neither of these two occur: Reward of -1**

For the Deep Q Network implementation, the rewards were set as follows:

- **If enemy attacks: Reward of -0.1**
- **If goal is reached: Reward of 1**

- **If neither of these two occur:**

  If the distance between the goal and player/agent is less than 1000 unit sq.: Reward = 1000/dist

  If the distance is greater than 1000 unit sq. : Reward = -(dist/100000)

This kind of reward function for the DQN architecture was followed to encourage the agent to always be close to the goal.

We are lurking the agent to get to the goal by giving it a large reward. But also penalizing it when it gets attacked. Though not penalizing it much, otherwise it may only focus on dodging the enemies and not getting to the goal. We are also considering distance as a factor because we want our agent to get close to the goal. As the Q values returned by DQN are between o and 1, we chose our rewards comparable to it.

Part II: Implementing the Deep RL algorithm

We've implemented the AIController class to control the behavior of an AI player in a game. To make the AI player intelligent, I've used a DQN neural network that employs the TD(0) loss function.

The AIController class is responsible for training the DQN network using the game data, and for using the trained network to make decisions during gameplay. During training, the class takes in the current state of the game and predicts the Q-values for each possible action using the DQN. These Q-values represent the expected reward for each action, given the current state of the game. The class then selects the action with the highest predicted Q-value and executes it.

During gameplay, the AIController class takes in the current state of the game and uses the trained DQN to predict the Q-values for each possible action. The class then selects the action with the highest predicted Q-value and executes it. This process is repeated for each game state until the game ends.

Overall, the AIController class provides an intelligent AI player that can learn from past gameplay experiences and make informed decisions during gameplay using the DQN neural network with TD(0) loss.

****

On evaluating the model, we were facing many issues, since the agent was not performing well. The code was working fine, but the agent was not learning to escape from the enemies. We tried many reward functions, Used various values of epsilon, in the epsilon greedy policy to control how the agent explores the environment, tried various learning rates, but the agent was not performing well.

There can be many possible reasons for it. Maybe the epochs were not enough. Maybe the reward function is still not good. Though we tried many learning rates but still there may be some possibilities.
Thus, we also implemented the Q-Learning algorithm.

****

Additional work :
We are also attaching the game_controllers_qtable.py file in which we have implemented the algorithm.
This implementation uses a Q-learning algorithm with a q-table to learn the optimal policy. The q-table is initialized with zeros and is of shape (GAME_WIDTH, GAME_HEIGHT, len(GameActions)). The agent updates the q-value for each state-action pair based on the observed reward and next state. However, the q-table only considers the current location of the agent and not the goal location due to memory constraints, which may affect the learning performance.

<u>Part III:</u> Changes in game parameters

Q) On the basis of your selected algorithm, briefly explain if after training your model, would it still score well in this game (without further retraining) on changing the following game constants ?

We tried to do the following changes with our learned model but as our model was not working properly, the results were unchanged for the following cases. Yet we tried to give theoretical answers according to our understanding.

- If GAME_SEED is selected randomly, rather than keeping it a constant number.

  It would produce the game with a different random seed, changing the initial adversary and goal positions, making it tougher for the agent to traverse and score.

  Keeping it consistent would train the agent on a fixed initial adversary and objective position and behavior.

- If the dimensions of the game, i.e GAME_WIDTH and GAME_HEIGHT are changed.

  Since the agent is trained on a fixed game width and height, changing either might result in poor performance because the agent may not be able to generalize over the new dimensions.

- If the variable GOAL_SIZE is changed.

Increase or reduce GOAL_SIZE. Increased accessibility would help the agent reach the objective state more often. However, decreasing the goal size would make the game harder for the agent since it would have to be more exact in its decision, which would take more training.

- If the variable ENEMY_COUNT is changed.

  If the ENEMY_COUNT is increased, the agent will have additional limitations and elements to contend with, making it harder to attain the goal state without retraining. If the ENEMY_COUNT is reduced, even the pre-trained agent will operate well because it was taught to more enemies (more limitations).

- If the variable GAME_FRICTION is changed.

  Changing the friction in the game would affect the dynamics of the entities, and the pre-trained agent would not be able to generalize to the new physics of the environment and the entities, therefore either raising or reducing the friction in the game would produce bad results without additional training. Changing the friction would also have the same effect.

- If the variable FPS is changed.

  If the FPS were increased, the amount of time it would take an agent to make a choice would decrease, which would necessitate further training in relation to the higher FPS. Because the agent has already been trained on a higher FPS, its speed of decision making is likely to be far faster than what may be necessary for the new FPS. As a result, lowering the FPS may still produce good results.

----------END----------