Aircraft Traffic Control with Reinforcement Learning

Name:	**Abhinav Sethi & Hritik Bana**
Registration No./Roll No.:	19006 & 19141
Institute/University Name:	IISER Bhopal
Program/Stream:	DSE

1 Introduction

In the modern world, the pace of life in general has increased quite significantly, and to match up with that pace, and save the time spent in travelling large distances, air travel is by far the most convenient mode available to most people. This increased demand has in turn decreased the airfare by a significant amount and thus a sort of virtuous cycle has been formed.

With more and more people getting access to travel to different places by flight, either for work or vacation, it implies a larger burden on the Air Traffic Control (ATC) to ensure that all the planes can complete their journey safely.

A plane crash is a very tragic experience in which many lives are lost. In 2021 alone, 134 deaths were reported from plane crashes worldwide. Thus, it shows that even with having highly skilled people managing the ATC, it is necessary to build a robust system that can take in all the complex data and guide the planes to the destination via a safe route.

2 Problem Statement

The task at hand is to simulate a real-life situation in which multiple planes are trying to reach a certain common location from various directions almost at the same time.

Given this situation, the reinforcement learning agent should be able to keep track of all planes to ensure that no two planes collide. The game environment [1] has to be modified to be represented as a Markov Decision Process (MDP), and then the agent (the aeroplanes) is trained on the SARSA algorithm to maneuver it's way to the goal (destination).

3 Approach taken

Our approach can be broken into four steps:

1. Understand how to modify the game built using PyGame, into an MDP:
    - Firstly, we noticed that we didn't need the main menu and high scores as they would cause discontinuity in episodes. The game also needed to be modified so that it has to be iteratively stepped through in order to play. This allowed us to generate and execute our desired actions at every timestep of the game.
    - We stripped the menu and made the game continuous over multiple episodes through the file game_ai.py (modification of game.py) by changing the while loop to a function, and commenting out the demo-game code inside the 'step' function.
    - To get the current list of active aircraft, potential colliders, and current status (reached destination etc) at each timestep, the function 'getPlaneDict' was made in game_ai.py.

- To obtain all pairs of planes in the game that are within the radius of potential collision from each other the function 'getCollidingAircraft' was made in game_ai.py. Further, we made the function 'getRewards' that utilises the information of the MDP (current state, action taken, new state), to generate the reward. All these constitute the information our MDP needs to make decisions about the game state.
- Now that we have the information the MDP needs, following the approach of the paper, we made an interface enabling us to modify the default game configurations, such as the number of planes, destinations (restricted to 1 in our experiments, which is in line with the paper provided to us), and obstacles per game (restricted to 0, same as the paper). Although we can handle many planes and destinations, we have removed the termination because of the obstacles.
- For training the agent, we made a file 'mdp.py', in which we first initialised the hyperparameters of the SARSA algorithm, such as learning rate, discount factor, and exploration probability (since we use an epsilon-greedy policy), and we have also provided a function 'setExplore' to change the exploration probability in between the game, which has been used to slowly diminish its value so that once the optimal state-action value estimate has been made, we can exploit the obtained knowledge and gain maximum return.
- The interface steps through the game, receiving the state information from each game timestep, processing the data, and generating an action using the function 'chooseAction' that follows the epsilon-greedy policy.
- Once we arrive at any state, whilst keeping track of the previous state and the action taken to reach here, we use the 'chooseAction' to select the action that is to be taken (following SARSA) and use the function 'update' to see if the two states were observed earlier or not; if not, then it assigns 0 q-value to each action corresponding to that state, and passes these values to the function 'updateQ', which uses the formula for SARSA for performing the update.
- To incorporate all the changes and reflect them in the pygame environment, we created a file 'game2rl.py'. In each timestep, the interface uses the 'getState' function to get the state of an aeroplane A, given it is in the colliding radius of some aeroplane B, and determines the best action for every active plane and modifies the game plane objects to add any necessary waypoints by using the function 'queueAction', before initiating a new timestep to continue the game. The file is highly related to mdp.py as it uses its functions to select the action and finally uses the 'trainSARSA' function to perform the updation along with the preliminary steps.

2. Since looking at the state space for random spawns and keeping track of those whilst trying to manoeuvre around collisions is very computationally expensive; thus the problem was broken into two major stages:
   - If there are other planes within a specified radius (the authors used 50 pixels, so we stuck to that as well), deploy a collision-avoidance agent to the closest two planes until they are clear of each other or a plane reaches a destination.
   - Else, follow the current trajectory (straight line towards the destination).
3. A few points to note:
   - We followed the paper's definitions for the state space, action space, and reward model.
   - In the original paper, the authors mentioned that they will be using SARSA but have given the algorithm for Q-Learning. Still, we decided to implement their original plan, i.e., SARSA.
4. The changes made have been detailed below:
   (a) aircraft.py: (Base code: aircraft.py)
      - getDistanceToGo(): This function calculates the distance for a plane that needs to be covered to reach the destination. Added this function to the original file.

(b) mdp.py: (New file)

- __init__(): Defined the SARSA class. Specified d, rho and theta, which are the state variables for each plane as mentioned in paper. Specified alpha (learning rate), lamda (discount factor), and explore (epsilon). A $n\_state \times n\_actions$ q-table, storing the predefined or intermediate step q-values for each state-action pair.
- setExplore(): This function is used to change the explore value during episodes. Used to define new exploration probability (epsilon).
- chooseAction(): At any given state it chooses an action by following the epsilon greedy policy.
- updateQ(): Given prevState, prevAction, reward, state, action; it finds the new Q-value for (prevState, prevAction) using SARSA.
- saveQ(): Saves the qTable at any point of the algorithm.
- loadQ(): Loads the stored qTable.
- update(): Given prevState, prevAction, State, reward, it sees if the two states were observed earlier or not; if not, then it assigns 0 q-value to each action corresponding to that state, otherwise it skips this step and utilizes 'chooseAction' to choose an action at present state and passes these values to updateQ.

(c) game2rl.py: (Base code: main.py)

- Setup the default/initial values of learning rate, lambda, and epsilon as 0.5, 0.9 and 0.1, respectively.
- Minor changes: Updating the epsilon every 10 episodes. Decided to save the Q-table after every 25 episodes, so as to checkpoint progress in relatively well-spaced intervals.
- trainSarsa(): Implementing the Sarsa on our agent.
- getState(): Get state of Plane 1 given the state of Plane 2, a potential collider. Necessary to help in rerouting.
- queueAction(): A function that carries out the chosen action.

(d) game_ai.py: (Base code: game.py)

- step(): Converted the game into a step-through game from a continuous one. Modification - Changed the while loop to a function, and commented out the demo-game code.
- getPlaneDict(): Added a function to keep track of all planes. To keep tab on all planes, to test for potential colliders, and current status (reached destination etc).
- getCollidingAircraft(): Obtain all pairs of planes in the game that are within the radius of potential collision from each other. A necessary addition, without which it would be impossible to make an agent learn.
- getRewards(): Made a function that utilises the information of the MDP, to generate the reward. A fairly simple implementation in which after all the checks, a reward is added for reaching the destination, and for the distance left to travel to reach the destination.

4 Results

What was achieved:

- We successfully converted the PyGame environment into an MDP (the most challenging part of the project, in our opinion, as we had very little exposure to PyGame and its functionalities).
- We were able to iterate through the game in a step-wise manner which enabled us to get the required information for the MDP: state, action, and reward.
- We were able to implement SARSA and obtained results strikingly similar to the paper, signalling that they had a typo in the algorithm given, and they had applied SARSA itself.

- We were able to reproduce the plots shown in the paper (Figure 5):

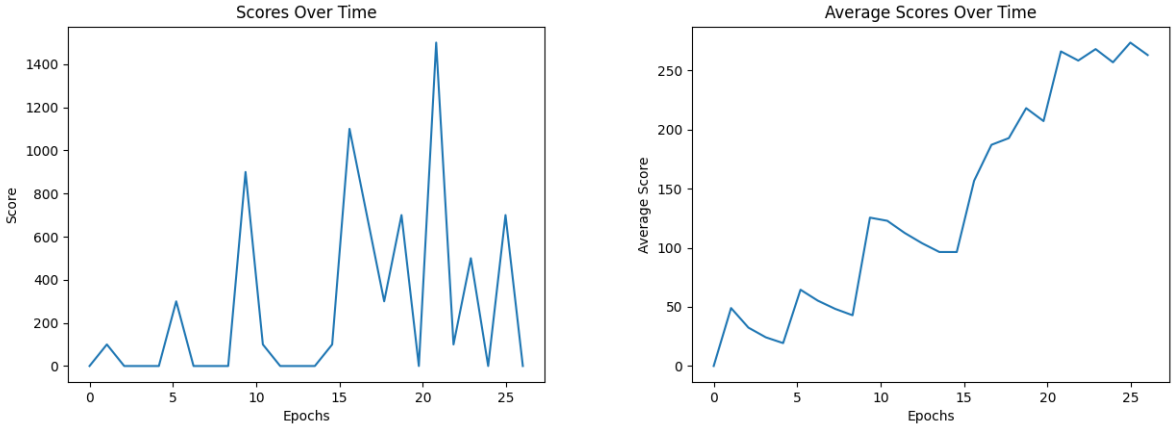

Figure 1 - Plots showing the per episode score, and average score of the SARSA algorithm uptil 25 episodes.

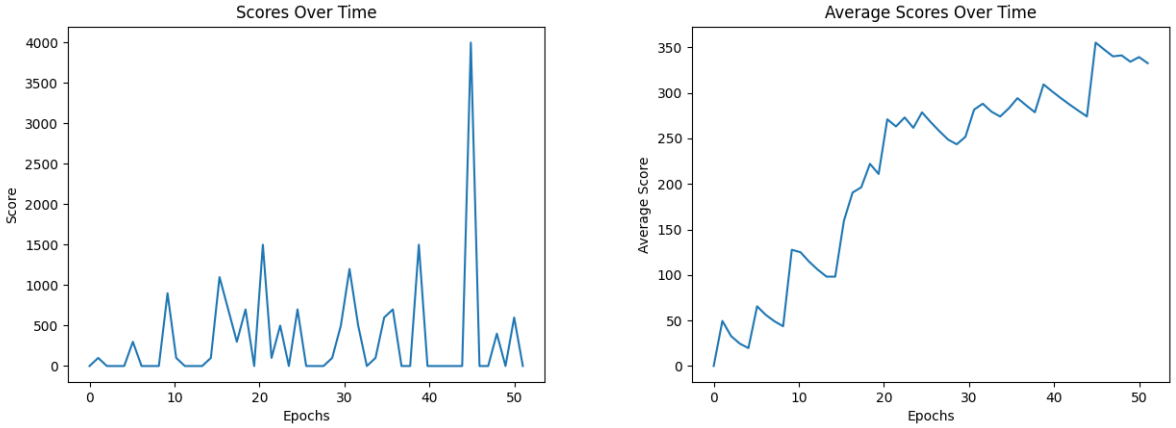

Figure 2 - Plots showing the per episode score, and average score of the SARSA algorithm uptil 50 episodes.

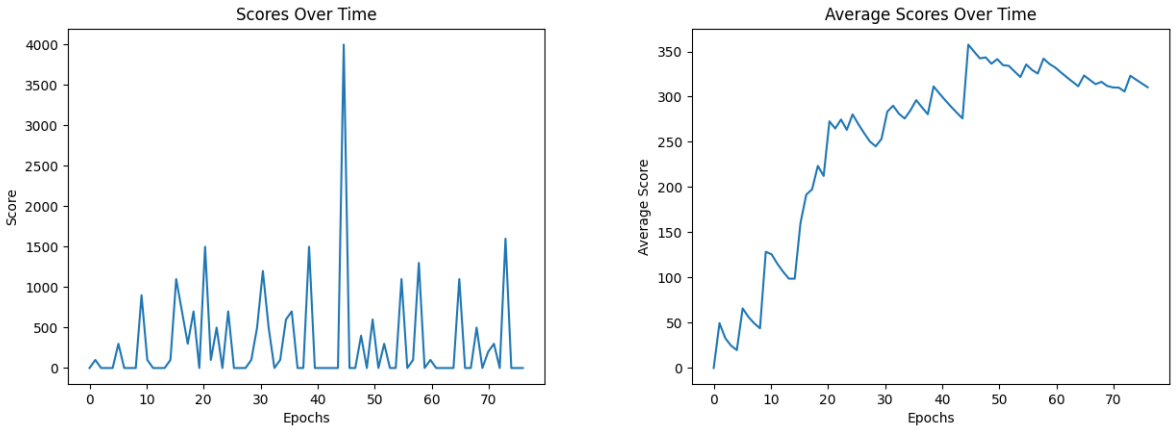

Figure 3 - Plots showing the per episode score, and average score of the SARSA algorithm uptil 75 episodes.

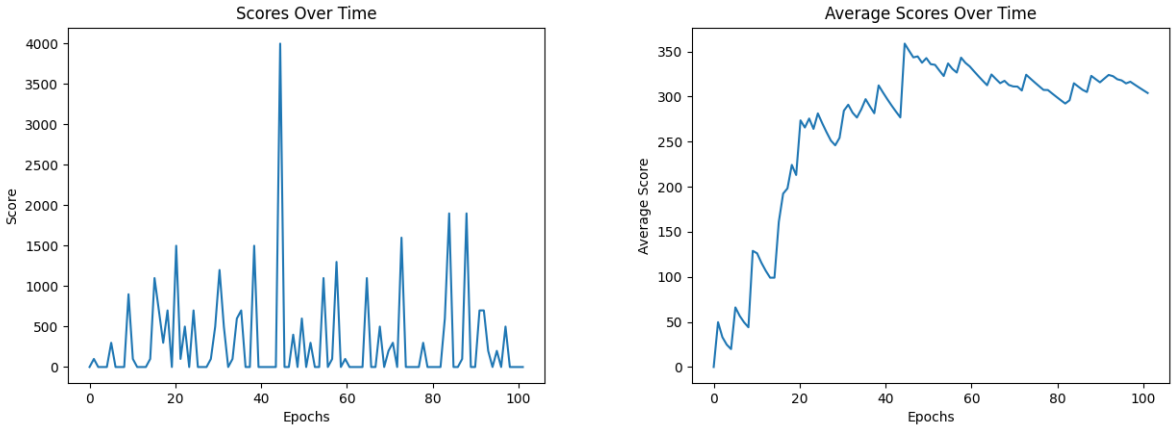

Figure 4 - Plots showing the per episode score, and average score of the SARSA algorithm uptil 100 episodes.

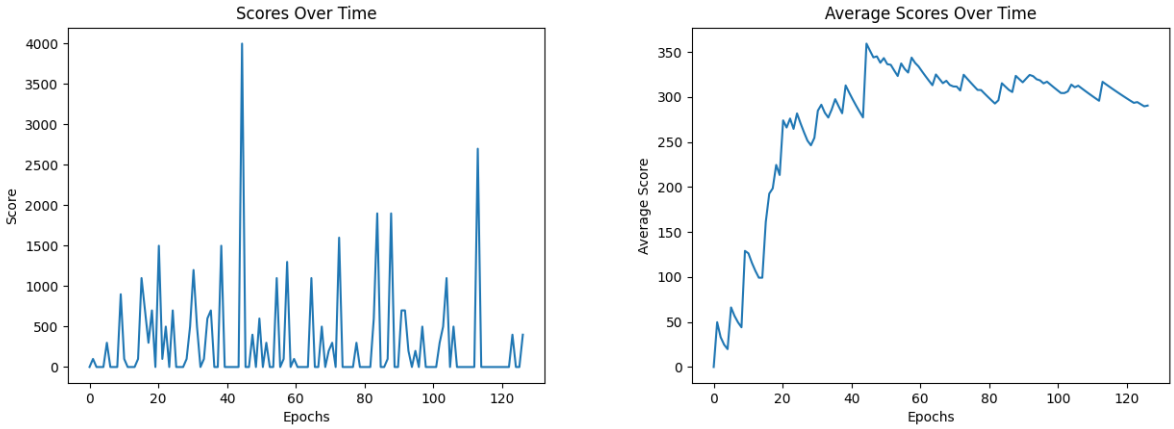

Figure 5 - Plots showing the per episode score, and average score of the SARSA algorithm uptil 125 episodes.