Aircraft Traffic Control with Reinforcement Learning

*A project for the course DSE 314: Reinforcement Learning

Akash Kumar Singh
Roll No.: 19023
Department: Data Science and Engineering

Alli Khadga Jyoth
Roll No.: 19024
Department: Data Science and Engineering

Indian Institute of Science Education and Research Bhopal

Motivation

Air traffic control (ATC) is a service provided by ground-based air traffic controllers who direct aircraft on the ground and through a section of controlled airspace as well as providing advising services to aircraft flying in non-controlled airspace. Since the early 1920s, ATC has been an important aspect of the aviation business. It was initially designed for military aircraft, but as passenger and freight flights became more common, air traffic rose, and ATC evolved into its modern form. The air traffic control system's day-to-day issues are mostly tied to the amount of air traffic demand placed on the system and the weather. The volume of traffic that may land at an airport in a given amount of time is determined by several factors.

Airplanes travel along established routes called **airways**, even though they are not physical constructions. They are defined by a particular width and altitudes, which separate air traffic moving in opposite directions along the same airway. Unlike on highways in which a collision threat is nearly always apparent, here a pilot is much more concerned with monitoring aircraft status than looking around for nearby planes.

Air traffic controllers use radar to track planes in their allotted airspace and communicate with pilots through radio. ATC enforces traffic separation standards to prevent collisions by ensuring that each aircraft has a minimum amount of empty space around it at all times. Air traffic control errors occur when the separation (either vertical or horizontal) between airborne aircraft falls below the minimum prescribed separation set by the US Federal Aviation Administration (FAA). The primary purpose of ATC is to prevent collisions, organize and expedite the flow of air traffic, etc. between origin and destination airports. The pilot in command has final authority over the aircraft's safe operation and may, in an emergency, divert from ATC directives to the extent necessary to keep their aircraft safe.

In 1981, the FAA developed TCAS (Traffic Alert and Collision Avoidance System), a mid-air collision avoidance system in response to rising aviation traffic and incidents of mid-air collisions. A new system called ACAS-Xu (Airborne Collision Avoidance System) is currently being developed which utilizes advancements in computer science and dynamic programming to generate alerts.

Problem Statement

A critical component of ATC is the task of directing airplanes to take-off and land between origin and destination airports without collisions. It's a difficult task that necessitates a series of difficult judgments. We describe a method for training an agent to autonomously direct flights to their destinations without colliding, in a 2D version of ATC. The problem is framed as a Markov decision process, and the SARSA reinforcement learning algorithm is used to control the planes' trajectories so that they arrive at their desired destinations without colliding. Our environment contains uncertainties in the form of spawn location fluctuation, plane number, plane starting heading direction, and unknown destinations.

Our general approach is to send planes on straight-line trajectories towards their destinations, with an ACAS-like, SARSA-based agent that takes over when two planes are in danger of collision.

Methodology

- ## ENVIRONMENT

  Each plane begins with a straight-line trajectory to its destination airport but has the ability for waypoints to be added along that trajectory to ensure that no planes collide into other planes or static objects. The goal of the game is to ensure that all planes can reach their destination without any collisions. We haven't used any existing environment instead we develop our own environment from scratch. In our environment the user can decide number of planes, number of destination airports, grid size, threshold for collision, in each episode whether the environment initializes to fixed states or randomly generated states (i.e seed).

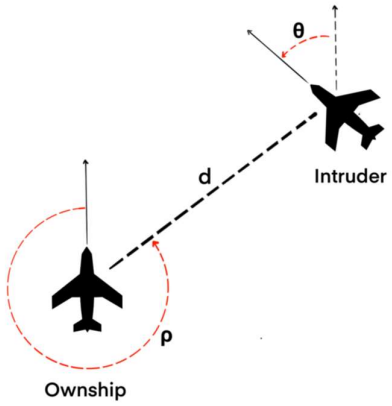
- ## MDP MODEL

  MDP Architecture : Similar to the ACAS/TCAS system, our task is to generate the optimal policy that a plane could follow to avoid a collision with an intruder aircraft nearby. The MDP formulation for SARSA requires a definition of the state space, action space, and reward model, which we will describe next.

  STATE Space : Our state space formulation is inspired by the ACAS-Xu system and measurements, real aircraft might receive about other nearby aircraft. We obtain these 'measurements' in real time and update the agent's states accordingly.
  We have discretized the measurements into buckets to reduce the size of our state space and compress the look-up table. The angle to intruder and relative heading of intruder are each 36 buckets of 10-degree intervals, representing the full 360 degrees. The distance to the intruder is measured in pixels, where 50 is the threshold radius for deploying the agent. The full size of our state space is 64,800, which is a compact representation compared to the 120 million states needed for ACAS-Xu. Table I describes our state space.

State variable	Description	No. of elements
d	Distance to intruder	50
ρ	Angle to intruder	36
θ	Relative heading of intruder	36

TABLE I: STATE Space



ACTION Space : We have chosen to limit our action space to 5. Our action space consists of going straight (N), taking a medium turn (ML, MR), or taking a hard turn (HL, HR). In pixel space, N corresponds to setting the next direction vector of the aircraft to the top centre pixel. In a similar fashion, ML and MR correspond to the top left and top right pixels, while HL and HR correspond to the left centre and right centre pixels. Table II describes our actions.

Action	Description	Actions name in our environment
N	Maintain course	UP
HL	Hard left	Left
ML	Medium Left	Front Left

MR	Medium Right	Front Right
HR	Hard Right	Right

TABLE II: ACTION Space

REWARD Model : Our reward model is dependent on the state variables. We decided to focus on two components, distance and destination.

For distance, we want the negative reward to increase quadratically as intruder planes became closer to our own ship. This means our algorithm would be less likely to choose actions that resulted in a decrease of distance between any two planes. Equation (1) describes the function used.

$$\text{intruder reward} = \frac{-(radius^2 - closest\_distance^2)}{(radius^2/500)} \quad (1)$$

where, radius is the threshold distance of 50 for deploying the agent and closest_distance is the distance between any plane and the plane it is closest to. By using the scaling factor in the denominator, it allows us to vary the negative reward from 0 when the planes are 50 pixels away, to 500 when the planes are about to collide.

We also chose to create a positive reward for reaching a destination so that the agent would be motivated to route the plane to a destination despite a nearby intruder. To handle this, we created a function called _get_info (), that returns the distance from the current plane to its destination as one of its outputs which is stored in the variable distanceToGo. Equation (2) describes our reward for approaching a destination.

$$destination\ reward = Maxdistance - distanceToGo \quad ----(2)$$

Where, $Maxdistance$ is the length of the diagonal of the grid space.
Both rewards are distributed to all agents per timestep in the update function of our learning algorithm. Additionally, to account for planes that have already reached the destination, we propagate the reward of reaching the terminal state back throughout the look-up table.

But in the research paper it's not clear how the authors integrated these 2 rewards into the SARSA Algorithm. Since SARSA only takes one reward in each timestep. Due to this ambiguity, we used a reward function as described below:

Reward = intruder reward + destination reward

If closest_distance > 50, then intruder reward = 0

With this our reward takes both the rewards into account. And motivates the agent to get maximum reward i.e., trying to avoid collisions but also avoid in such a way which maximizes the destination reward.

- # IMPLEMENTATION

However, in this project, we are only focusing on avoiding collisions and could not predict which actions would lead to collisions and which ones would not We chose a model-free learning technique utilising Temporal Difference because we wanted our agent (planes) to learn how to avoid collisions from actual experience rather than an unknown collision model (TD). Temporal Difference algorithms allow an agent to learn via each action it performs by updating the agent's knowledge at each time step rather than at the end of each episode (achieving the goal/end state). This knowledge update is the update of the estimate of the utility given by:

New = Old + α (Target - Old) ------- (3)

Here the new estimate is proportional to the difference in target utility and the previous estimate. α in the above equation is called the learning rate and is representative of the step size applied in the update with values between 0 and 1.

SARSA

We chose SARSA as our reinforcement learning algorithm as SARSA uses the actual action taken at S_{t+1} to update Q values instead of maximizing over all possible actions as done in Q-learning. SARSA is an on-policy learning algorithm, meaning it learns the value of the policy being carried out by the agent, including the exploration steps, while the policy continues to be followed.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Our SARSA algorithm takes values of learning rate (α), discount factor (γ), exploration probability (ε) and its decay rate from users.

SARSA Implementation

Below points describe how this algorithm helps to train our agents to avoid collision between each other:

1. We initialize the Q table with zero.
2. The first episode starts alongside the start of the game with multiple planes spawning in random locations heading towards a destination airport (each episode represents one instance of training without collision).
3. For the next episode we use the Q-table updated till last episodes.
4. We choose an action based on an already updated Q-table.
5. We run the game in timesteps permitted by the environment.
6. We train our MDP for planes which are within the collision radius and choose an action (turning in a particular direction) using an exploration vs. exploitation strategy.
7. We observe the next state and reward gained after taking the previous action and ending up in the next state.
8. Based on the previous action, state and reward we update our Q - values.
9. Finally, we update the state and action of the planes.
10. We repeat this until the planes reach their destination (goal state) or until there is a collision.
11. We train the model until the desired number of episodes, and the learned policy can be saved in the form of a Q-table.

We have implemented **SARSA + Greedy** as our main algorithm. This is implemented as: for the planes lying in the threshold distance here we took its value as 50, we choose action according to SARSA and for the planes having distance > 50 we choose their action greedily to reach their destination airports.

We have implemented our algorithm (SARSA + Greedy) for two cases:
1). With single destination 2). With multiple destination and in each case, we used two methods for choosing action by ε-greedy namely: a). without reducing ε i.e Constant Temperature SARSA b). with reducing ε by the rate of decay factor i.e Temperature Controlled Exploration.

Results

To measure the performance of our RL agent (SARSA + Greedy), we have trained our agent for 1000 episodes with 25 planes and other parameters are: for

a) **single destination with Constant Temperature SARSA**
   α = 0.1, γ = 0.9, ε = 0.1

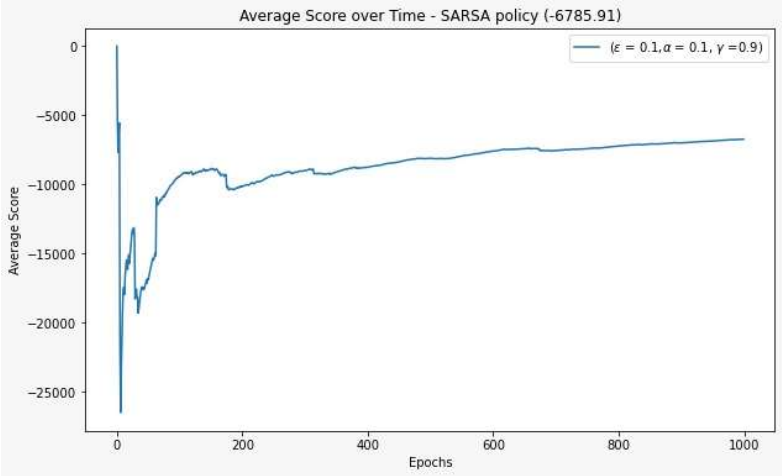


Fig.1

b) single destination with Temperature Controlled Exploration

α = 0.5, γ = 0.9, ε = 0.5, decay factor = 10% reduce every 10 episodes

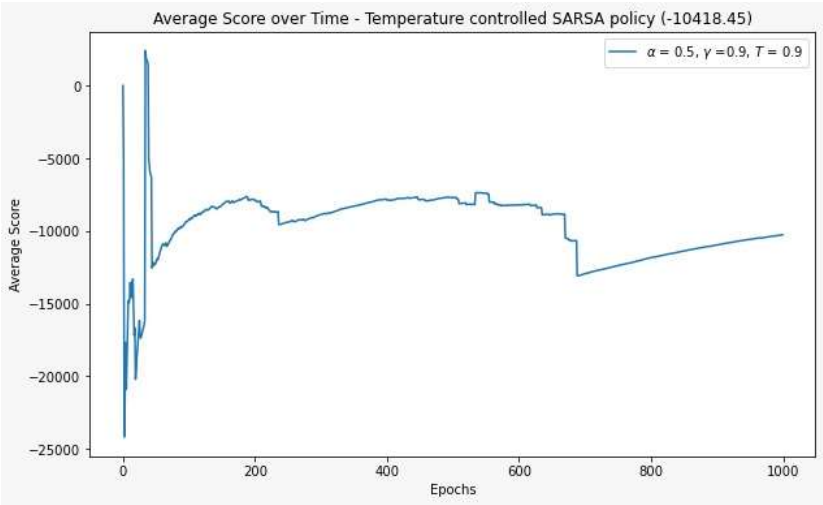


Fig.2

c) multiple destinations with Constant Temperature SARSA

α = 0.5, γ = 0.9, ε = 0.5, the no. of destination airports = 25, no. of episodes = 500

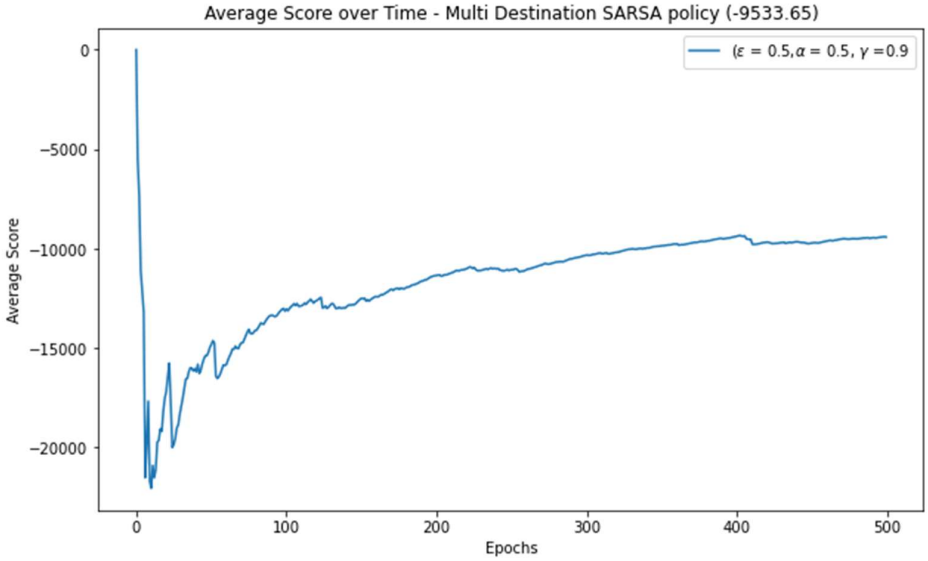


Fig.3

d) multiple destinations with Temperature Controlled Exploration

$\alpha = 0.5$, $\gamma = 0.9$, $\varepsilon = 0.5$, decay factor = 10% reduce every 10 episodes, the no. of destination airports = 25, no. of episodes = 500

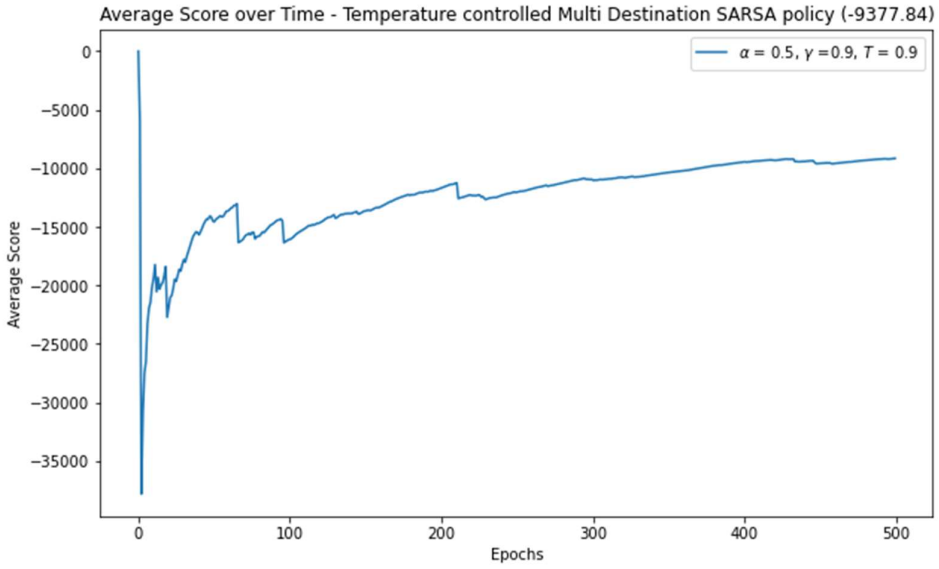


Fig. 4

We compared the performance of our agent against two benchmarks - a random policy agent and a straight line following agent (Greedy Agent). Figures 1-7 show the comparison between the performance of the agents. Average score is calculated as:

avg_reward = sum of all rewards of all agents averaged over no. of agents in each episode

Average Score = sum of avg_reward averaged over no. of episodes

1) **Random Policy Agent:** This agent takes one of the possible actions at random to avoid collisions when near other planes. The figure below shows the plot of average scores vs no. of episodes.

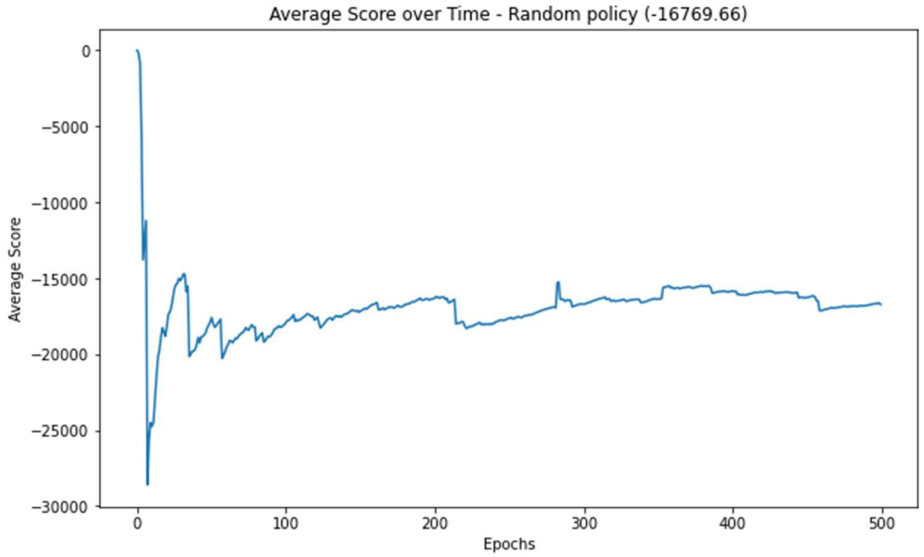


Fig.5

2) **Straight Line Following Greedy Agent:** This agent takes the straight-line trajectory to their destination regardless of the impending collisions. The figure below shows the plot of average scores vs no. of episodes.

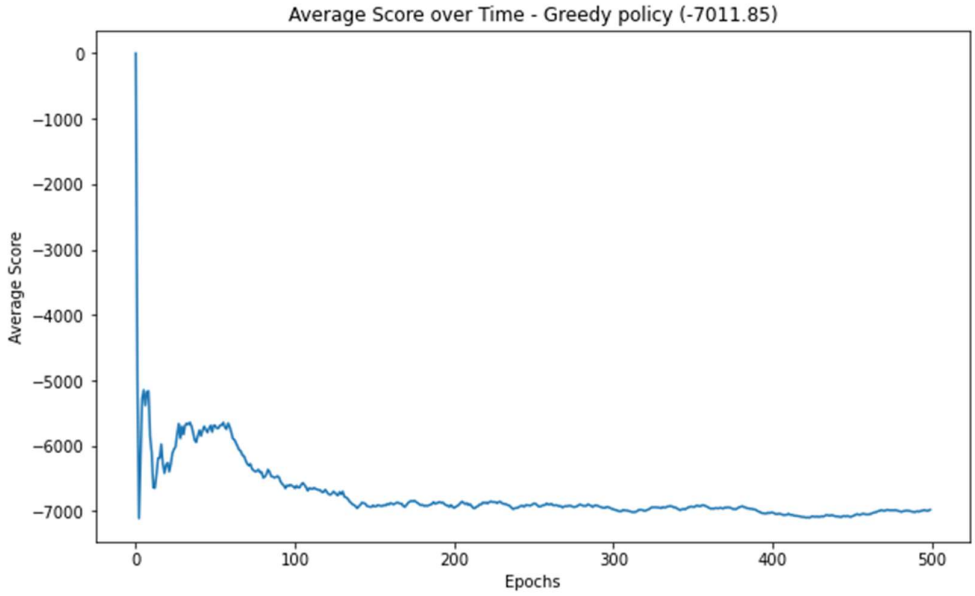


Fig.6

3) **SARSA + Greedy Agent with Constant Temperature SARSA :** The figure below shows the plot of average scores vs no. of episodes.

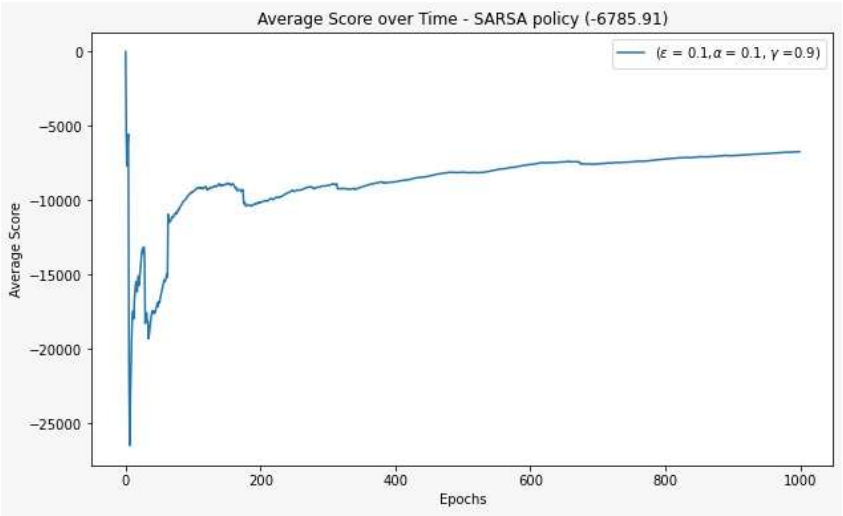


Fig.7

TABLE III represents the comparison of average scores of these 3 algorithms.

Agent	Average Score
baseline - random agent	-16770
baseline - greedy agent	-7012
SARSA + Greedy	-6786

TABLE III: AVERAGE SCORES OF VARIOUS AGENTS

The figure below shows the comparison between all the agents:

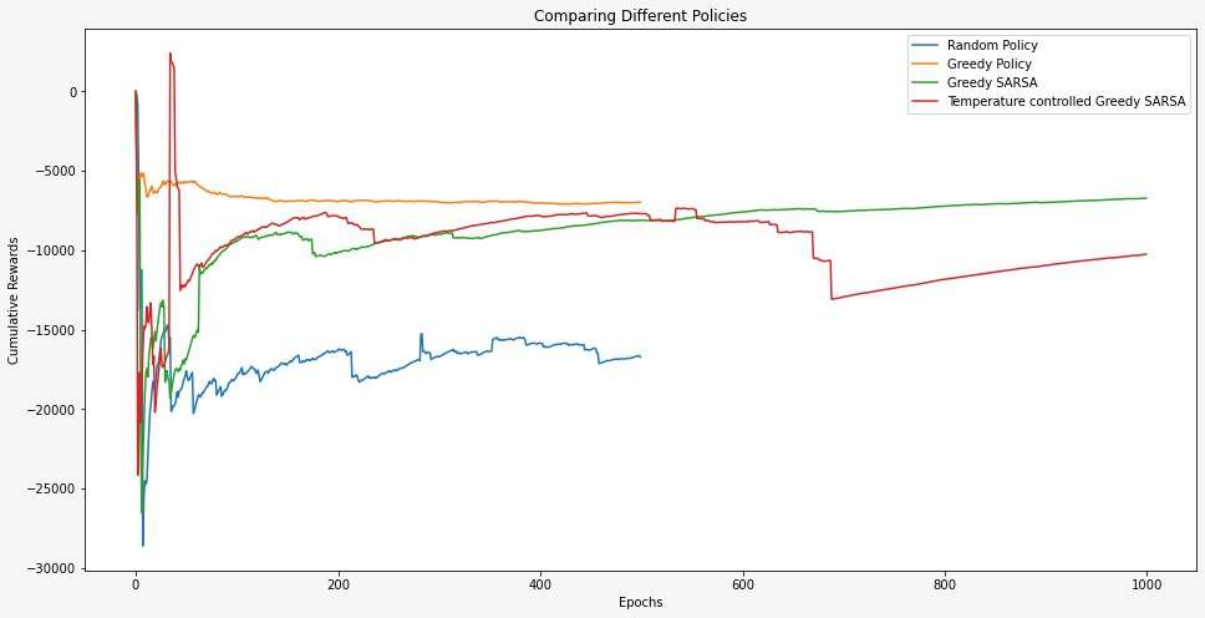


Fig. 8

As you can see in Fig. 8, our agent has an average score of around -6786, which is 3 times better than the average score for the random agent and better than greedy agent.

Conclusion

Our SARSA agent was able to achieve reasonable success in collision avoidance compared to our baseline agents. Our RL agent had consistently better scores once it had learned from its initial episodes.

The authors suggested to implement the SARSA with multiple airport destinations with often crossing paths. And we were successfully able to implement that with our environment thanks to our custom environment code, which we wrote from the ground-up. In our environment we can select how many destinations we would like to have. It can range from 1, where all the planes land at same destination (same as implemented in the paper), to number of planes, where each plane lands at a different destination.

Limitations

We were unable to visualize our environment using Pygame.