Aircraft Traffic Control with Reinforcement Learning

Pushpendra Dhakar 19229 and Udaybhan Rathore 19328

13th May 2022

1 Motivation

Air traffic control is a real-time safety-critical decision-making process is highly dynamic and stochastic environments. The development of automatic control systems has played an essential role in civil and military aviation growth. Modern aircraft include a variety of automated control system that aids the flight crew in navigation, flight management, and augmenting the stability characteristics of the airplane. A human air traffic controller monitors and directs many aircraft flying through its designated airspace sector in today's aviation practice. With the fast-growing air traffic complexity in traditional (commercial airliners) and low-altitude (drones) airspace, an autonomous air traffic control system is needed to accommodate high-density air traffic and ensure safe separation between aircraft.
We propose a deep multi-agent reinforcement learning framework to identify and resolve conflicts between aircraft in a high-density, stochastic, and dynamic en-route sector with multiple intersections and merging points. The proposed framework utilizes an actor-critic model, A2C, that incorporates the loss function from Proximal Policy Optimization (PPO) to help stabilize the learning process. In addition, we use a centralized learning, decentralized execution scheme where one neural network is learned and shared by all agents in the environment. We show that our framework is scalable and efficient for many incoming aircraft to achieve high traffic throughput with a safety guarantee. We evaluate our model via extensive simulations in the Blue Sky environment.

2 Problem statement

Air traffic control (ATC) is the task of directing airplanes to take off and land at airports without collisions. It is a complex task involving a sequence of difficult decisions. This project aims to train a neural network to per for primary Air Traffic Control Tasks through reinforcement learning. This project is an example of typical game play where multiple planes try to reach multiple destinations while avoiding obstacles.Each plane begins with straight-line trajectory to its

destination airport but has the ability for waypoints to be added along that trajectory to ensure that no aircraft collide with other planes or static objects.

3 Approach Taken

3.1 structure

Open-AI Gym Environment is used as the interface between the simulation and Reinforcement.

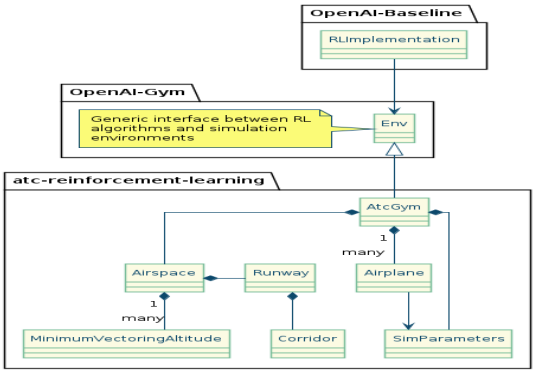

Figure 1: Model pipeline

3.2 learning algorithm Create Custom Simulation

The task can be simulated, and later further complexities can be added to the simulation, the basic features for a minimal viable simulation would be as follows.

3.2.1 State space

State space is measured measurements real aircraft might receive about other nearby aircraft. We obtain these 'measurements' in real time and update the agent's states accordingly

3.2.2 Actions

Our action space consists of going straight (N), taking a medium turn (ML, MR), or taking a hard turn (HL, HR). 1.Assign a heading to the airplane. 2.Assign an altitude to the airplane.

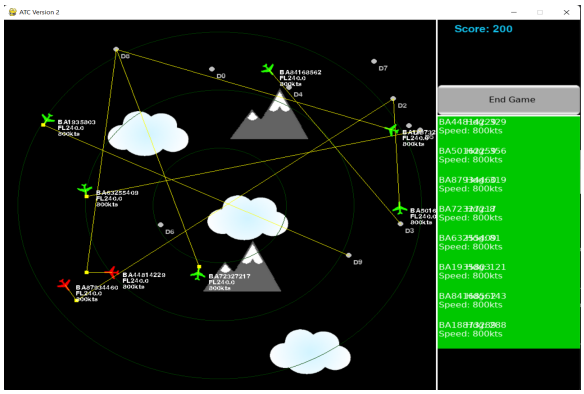

Figure 2: Visualization Env

3.2.3 Rewards

Bringing the aircraft to the Final Approach Fix (FAF) at the correct altitude and within the right angle wins the simulation. Every step the airplane has not reached the FAF yet carries a minor penalty. Descending the airplane below the Minimum Vectoring Altitude carries a heavy penalty.Not maintaining sufficient separation (e.g., 3nm and 1000ft) between two aircraft carries a heavy penalty. Having the aircraft leave the sector carries a heavy penalty.All heavy penalties could also be simulation end/game overstate.

3.3 Environment

Simulate a minimum vectoring altitude. This is a coarse approximation of the terrain. At least one aircraft. Initially, a single plane could be simulated. Runway with a final approach course Final approach fix. This would be the target point to which the plane should move at or below a specific altitude. Basic aircraft model with fixed turn rate and descend/climb rate.we are using Pygame Module. PyGame is a set of Python modules used for programming video games.

3.4 Markov-Decision Process

The project vision was to produce the optimal policy that a plane could follow to avoid a collision with an intruder aircraft nearby. The Markov-Decision Process formulation for SARSA requires a definition of the state space, action space,Reward Model.

3.5 General Approach

: The problem of optimally routing planes without collisions while taking into account uncertainty of new plane spawns can become a complicated and com-

putationally expensive task. To simplify the problem, we have instead chosen to use a two pronged approach:

1) If there are other planes within a specified radius, deploy a collision-avoidance agent to closest two planes until they are clear of each other, or a plane reaches a destination.
2) Otherwise the path is clear, therefore make a beeline for the destination.

4 Implementations

The Sarsa algorithm is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name Sarsa actually comes from the fact that the updates are done using the quintuple Q(s, a, r, s', a'). Where: s, a are the original state and action, r is the reward observed in the following state and s', a' are the new state-action pair. The procedural form of Sarsa algorithm is comparable to that of Q-Learning: We chose SARSA as our reinforcement learning algorithm over Q-learning as SARSA uses the actual action taken at st+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.

4.0.1 SARSA Algorithm

:
1 Initialize Q(s,a) for all s S and a A

2 t ← 0

3 repeat for each episode

4 s0 ← initial state

5 Choose A from S based on policy derived from Q

(-greedy)

6 repeat for each step of episode

7 Choose action at based on policy derived from

Q (-greedy)

8 Observe new state st+1 and reward rt

9 Q(st, at) ← Q(st, at) +

10 (rt + maxaQ(st+1, at) Q(st, at))

11 s ← st+1; a ← at+1

12 until until s reaches goal state;

13 until episodes over;

4.0.2 SARSA Implementations

This aims to achieve an optimal Q value and not just the optimal value of the state. which is the learning rate needs to be initialized, which determines how quickly we make changes to the Q function.

- Q (St, At) is initialized where, Si be the state, At be the action. The Q-table can be initialized with '0' for every state-action pair. This is the agent's estimate of its discounted feature reward, starting from state St and doing an action At.
- Choose an action At from state Si based on epsilon greedy strategy for which we get a reward rt. Epsilon greedy is a strategy to maximize the Q(St, At) value with a probability of 1- or apply a random action at at state Si with a probability of , where is a small number
- The Q (St, At) is calculated using the formula: Q (St, At)) = Q (St, At)) + (rt+1 + * Q (St+1, At+1) – Q (St, At))
- Jump to the next state and perform the next action
- Repeat from step 2 iteratively as i=i+1 for all the states in that episode

Repeat from step 2 iteratively as i=i+1 for all the states in that episode.

4.0.3 SARSA

We implemented the Markov-Decision Process structure.We will briefly describe how this algorithm helps to train our agents to avoid collision between each other (the ordered list correspond to line numbers in the above algorithm):

- Firstly We initialize Q table(Empty Dictionary)

- The first episode starts along side the start of the game with many planes spawning in different locations heading towards a destination airport (each episode represents one instance of training no collision).

- We used an offline policy look-up table (Q table) after simulation of every 25 episodes which may be used later.we can below figure

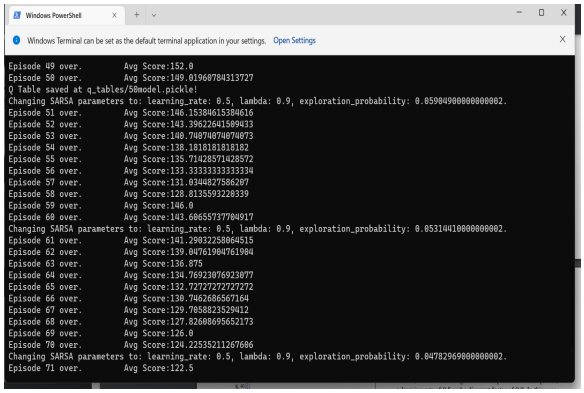

Figure 3: Avg.episodes

- We initialize all unknown states with a zero vector

- We choose an action based before trained look-up Q table saved.

- We run the game in time steps permitted by the environment

- 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.

- We execute the next state and reward (low reward for maneuvers, which might lead to collision) gained after taking the previous action and ending up in the next state.

- Based on the pevious action, state and reward. we update our Q - values.

- Finally, we update all the state and action of the planes

- We repeat this until the planes reach their main destination or goal state or until there is a collision.

- We train the model until the desired number of episodes, and the learned policy can be saved in the form of a Q table.

4.0.4 Simulation Configuration

We have modified our simulation environment such that it can accept various parameters. Thus, we can run our simulation with various configuration parameters to train our agents under various conditions for regressive training (similar to various difficulty levels for games). Following are the parameters, which can be varied during training:

- Number of planes(Twenty five planes)
- Number of spawnpoints
- Number of destinations(Ten destination)
- Number of obstacles(Five)
- Load a previously trained q-table
- Learning rate
- Discount factor
- Exploration probability

5 Result

5.1 Experiments

We simulated our game environment with the following parameters: 10 airport destinations, five obstacles, 25 planes, and a varying number of episodes. Every episode is a run of the program where the agent accumulates points until two planes collide. We compared the performance of our agent against two benchmarks - a random policy agent and a straight line following agent. Below figure show the comparison between the performance of the agents.

5.2 Result with different algorithms

5.2.1 Straight Line Following Greedy Agent

This agent takes the straight-line trajectory to its destination regardless of the impending collisions. This agent shows some peak scores during its run. This can again be attributed to the burn-in time of our environment. We can see that the average score to around 25 for this agent as this agent is focused on reaching the destination without caring about the collision. Due to this one or two planes in the episodes often reach their destination, raising the score above the random policy agent's score. Thus, this greedy agent aims to maximize reward but fails to do better than our agent because of the failure to learn from the collisions.

5.2.2 Random Policy Agent

This agent takes one of the possible actions at random to avoid collisions when near other planes. This agent shows a peak score in the beginning. Since the airplane spawn events are unexpected, this introduces some chance events with very sparse plane density. This might be an explanation for an alienated spike seen in the beginning. We count this as the "burn-in" time of our environment. We can see that the scores average to around 30 for the random policy event, as random policy should produce similar scores when averaged over many episodes. Moreover, these low scores are because the accidental policy agent doesn't learn to improve its results over time.

5.2.3 SARSA Agent

We trained our SARSA agent with 25 planes to follow a single airport destination. Table III below illustrates the different parameters used while preparing the SARSA agent. First, we trained the agent for 125 episodes with 25 planes, a learning rate of 0.5 and a discount factor of 0.9. In this case, we started with an exploration probability of 0.5, which we reduced by 10 present every ten episodes. The reason for doing this is that we wanted to start with more exploration and gradually lessen the investigation to begin exploiting the knowledge gained from our Q-Tables. Our SARSA agent has an average score of around 300, which is ten times better than the average score for the random agent and about three times better than the greedy agent. The average score for SARSA agents is over 125 episodes. Following the previous training instance, we trained the agent for 500 attacks with 25 planes, with a learning rate of 0.5 and a discount factor of 0.9. In this case, we used an exploration probability of 0.1; since we wanted to exploit our learned policy aggressively. Our SARSA agent converges to an average score of about 300 with a steadily increasing trend for scores. This shows again that our SARSA agent is ten times better than the average score for the random agent and about three times better than the greedy agent.

6 Conclusion

In this paper, we presented a reinforcement learning approach to designing an aircraft collision avoidance system similar to what is used in air traffic control. We used a PyGame environment to simulate a simplified version of this scenario and modeled our agent as an MDP running a SARSA algorithm. Our SARSA agent achieved 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. There remains a lot of room for improvement and development for our agent. We can add actions to our agents, such as an option to reduce or increase the speed and change directions. Also, we did not

have enough time to run with various configurations planned with our simulator.

Our current implementation involves simulation-based learning, which we may modify in the future to learn without running the simulation, thus, making the training process faster. Currently, we are limited by the frame rendering rate for running episodes. We could also try other RL algorithms instead of SARSA or implement SARSA-Lambda to increase reward propagation. An exciting learning case could be to train our agent alongside the greedy agent to simultaneously improve our agent's performance against cheap and intelligent agents.