Towards Autonomous Air Traffic Control for Sequencing and Separation - A Deep Reinforcement Learning Approach

Marc Brittain* and Peng Wei†
Iowa State University, Ames, IA, 50011, U.S.A.

In this research, we study the feasibility and solution performance of using an artificial intelligent agent to perform the sequencing and separation task in air traffic control. Our goal is to safely separate the air traffic and minimizing the delay. To do this, we have the options to select the most efficient route and provide speed change commands for each aircraft. We use the NASA Sector 33 as our simulator to demonstrate our proposed method. Our results show that our artificial intelligent agent beats most of the human players in this simulation environment and is able to maintain safe separation and sequencing between the aircraft.

I. Introduction

The pursuit of automated air traffic control (ATC) dates back to the 1970's. Since then, Dr. Heinz Erzberger and his NASA colleagues have laid the foundation for our modern day and future air traffic management technologies [1–4]. Their work has proposed automated conflict detection and automated conflict resolution methods using time-based metering, optimization and real-time trajectory re-planning. Their focus has been in terminal area air traffic. In this paper, we will focus on providing tactical decision support to air traffic controller to select route and change speed for each individual aircraft.

In our research, we use an Air Traffic Controller simulator created by NASA; Sector 33, as an environment to test our reinforcement learning techniques. NASA Sector 33 contains many different problems where there are changes in the number of planes and the number of routes. The objective of the game is to alter the aircraft's velocity, as well as, route in such a way to minimize the arrival delay, maintain a safe distance, and avoid collision. In our implementation, we modify the original environment to end and restart if the planes collide. In the original implementation, the planes will continue to progress. To change the velocity of an aircraft, the user must navigate the mouse to the location of the aircraft and click on it. Only when the user clicks on the aircraft will the velocity menu be displayed, which creates some difficulty when converting to a reinforcement learning environment.

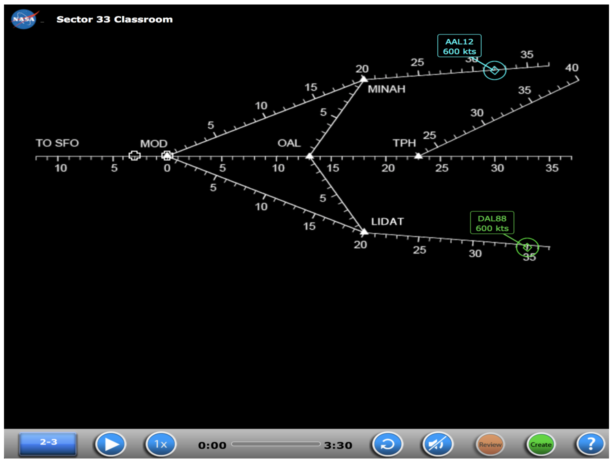


Fig. 1 NASA Sector 33 web-based game. We can see that there are two aircraft's on the right and on the left, there are the two goal positions for each aircraft.

The outline for our paper is as follows: in Section II we include some background on the different reinforcement learning techniques, building up to the technique we implemented. In Section III we discuss our method for formulating this problem and how this problem environment differs from most environments being explored today. In Section IV we describe how we solved the problem with Section V including the results of our numerical experiments. Section VI summarizes the results and provide ideas for future work.

II. Background

In this section, we will provide background on reinforcement learning and a brief survey of current state-of-the-art techniques.

A. Reinforcement Learning

Reinforcement learning is one type of sequential decision making where the goal is to learn how to act optimally in a given environment with unknown dynamics. A reinforcement learning problem involves an environment, an agent, and different actions the agent can take in this environment. The agent is unique to the environment and we assume the agent is only interacting with one environment. Let t represent the current time, then the components that make up a reinforcement learning problem are as follows:

- S - The state space S is a set of all possible states in the environment
- A - The action space A is a set of all actions the agent can take in the environment
- $r(s_t, a_t, s_{t+1})$ - The reward function determines how much reward the agent is able to acquire for a given (s_t, a_t, s_{t+1}) transition
- $\gamma \in [0, 1]$ - A discount factor determines how far in the future to look for rewards. As $\gamma \to 0$, only immediate rewards are considered, whereas, when $\gamma \to 1$, future rewards are getting prioritized.

S contains all information about the environment and each element s_t can be considered a *snapshot* of the environment at time t. The agent accepts s_t and with this, the agent then decides an action, a_t. By taking action a_t, the state is now updated to s_{t+1} and there is an associated reward from making the transition from $s_t \to s_{t+1}$. How the state evolves from $s_t \to s_{t+1}$ given action a_t is dependent upon the dynamics of the system, which is unknown. The reward function is user defined, but needs to be carefully designed to reflect the goal of the agent. Figure 2 shows the progression of a reinforcement learning problem.

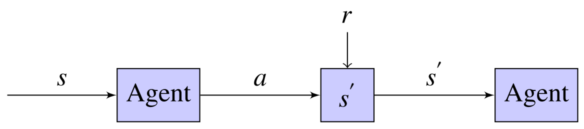


Fig. 2 Progression of a reinforcement learning problem within an environment.

From this framework, the agent is able to learn the optimal decisions in each state of the environment by maximizing a cumulative reward function. We call the sequential actions the agent makes in the environment a *policy*. Let π represent some policy and T represent the total time for a given environment, then the *optimal policy* can be defined as:

$$\pi^* = \arg\max_{\pi} \mathrm{E}[\sum_{t=0}^{T}(r(s_t, a_t, s_{t+1})|\ \pi)]. \tag{1}$$

If we define the reward for actions we deem "optimal" very high, then by maximizing the total reward, we have found the optimal solution to the problem.

B. Q-Learning

One of the most fundamental reinforcement learning algorithms is known as Q-learning. This popular learning algorithm was introduced by Watkins [5] and the goal is to maximize a cumulative reward by selecting an appropriate action in each state. The idea of Q-learning is to estimate a value Q for each state and action pair (s, a) in an environment that directly reflects the future reward associated with taking such an action in this state. By doing this, we can extract the policy that reflects the optimal actions for an agent to take. The policy can be thought of as a mapping or a look-up table, where at each state, the policy tells the agent which action is the best one to take. During each learning iteration, the Q-values are updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)). \tag{2}$$

In equation (2), α represents the learning rate, r represents the reward for a given state and action, and γ represents the discount factor. One can see that in the $\max_{a'} Q(s', a')$ term, the idea is to determine the best possible future reward by taking this action.

C. Deep Q-Network (DQN)

While Q-learning performs well in environments where the state-space is small, as the state-space begins to increase, Q-leaning becomes intractable. It is because there is now a need for more experience (more game episodes to be played) in the environment to allow convergence of the Q-values. To obtain Q-value estimates in environments where the state-space is large, the agent must now generalize from limited experience to states that may have not been visited [6]. One of the most widely used function approximation techniques for Q-learning is deep Q-networks (DQN), which involves using a neural network to approximate the Q-values for all the states. With standard Q-learning, the Q-value was a function of $Q(s, a)$, but with DQN the Q-value is now a function of $Q(s, a, \theta)$, where θ is the parameters of the neural network. Given an n-dimensional state-space with an m-dimensional action space, the neural network creates a map $R^n \longrightarrow R^m$. As mentioned by [7], incorporating a target network and experience replay are the two main ingredients for DQN [8]. The target network with parameters θ^-, is equivalent to the on-line network, but the weights (θ^-) are updated every τ time steps. The target used by DQN can then be written as:

$$Y_t^{DQN} = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_t^-). \tag{3}$$

The idea of experience replay is that for a certain amount of time, observed transitions are stored and then sampled uniformly to update the network. By incorporating the target network, as well as, experience replay, this can drastically improve the performance of the algorithm [8].

D. Double Deep Q-Network (DDQN)

In Q-learning and DQN there is the use of a max operator to select which action results in the largest potential future reward. [7] showed that due to this max operation, the network is more likely to overestimate the values, resulting in overoptimistic Q value estimations. The idea introduced by [9] was to decouple the max operation to prevent this overestimation to create what is called double deep Q-network (DDQN). To decouple the max operator, a second value function must be introduced, including a second network with weights θ'. During each training iteration, one set of weights determines the greedy policy and the other then determine the Q-value associated. Formulating equation (2) as a DDQN problem:

$$Y_t^{DDQN} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_t); \theta_t^{'}). \tag{4}$$

In equation (3), it can be seen that the max operator has been removed and we are now including an argmax function to determine the best action due to the online weights. We then use that action, along with the second set of weights to determine the estimated Q-value.

III. Problem Formulation

A. Problem Statement

In real world practice, the air traffic controllers in en route and terminal sectors are responsible for sequencing and separating aircraft. In our research, we used the NASA Sector 33 web-based application (a video game) as our air traffic control simulator. The application has a set of problems whose solutions are difficult to find. To evaluate the performance of our deep nested agent algorithm, we selected a complex problem with many possible routes for the aircraft to take.

1. Objective

The objective in the NASA Sector 33 game environment is to maintain a safe separation between aircraft, resolve conflict, and minimize delay by making appropriate route changes and providing speed advisories. Ultimately, we want to guide each aircraft quickly flying through the metering fix. Unlike the real world air traffic control scenario, in this game environment there is also a unique “final metered position" right after the metering fix for each aircraft. It is not too difficult for a good human player to obtain a *feasible solution* in the game, where the aircraft are close to their final metered position, safe separation is maintained, and conflicts are resolved. In order to obtain the *optimal solution* in this environment, the aircraft have to maintain safe separation, resolve conflict, and arrive at their final metered position with no delay. However, obtaining the optimal solution in this environment is much more difficult than obtaining the feasible solution, due to the fact that each aircraft has to follow the optimal speed and route at every time-step. If one speed change is made incorrectly, then the optimal solution will not be achieved.

2. Constraints

There are many constraints in the NASA Sector 33 game environment to help resemble a real-world air traffic environment. For each problem, there is a fixed number of aircraft. For some problems there is only two aircraft, some there is three aircraft, and this number increased to a maximum of five aircraft. The next constraint imposed a limit on the number of route changes for a given aircraft. For example, one problem might allow for a single aircraft to change routes, while another problem would allow both aircraft to change routes, thus increasing the complexity. Weather also imposed an additional limit on the number of aircraft route changes. In some of the problems, there is a storm blocking one of the routes, so now there is no option to select the corresponding route anymore. One of the most strict constraints in the game is time. In each problem, there is a timer for how long the episode lasts and there is only enough time for the optimal solution to be obtained. This meant that to obtain the optimal solution, the aircraft has to be at their final metered position when the timer is at 0:00, otherwise it is not an optimal solution. The last constraint in the game is the individual speed of each aircraft. Each aircraft has the ability to fly at six different speeds ranging from 300 knots to 600 knots.

B. Reinforcement Learning Formulation

Here we formulate the NASA Sector 33 environment as a reinforcement learning problem and define the state-space, action-space, and reward functions for the main agent, as well as, the nested agent.

1. State Space

A state contains all the information the AI agent needs to make decisions. The state information was composed of different information for the main agent and nested agent. For the main agent, the information included in the state was a screen-shot of the game screen. For the nested agent, if we let i represent a given aircraft, then the information included in the state was: aircraft positions (x_i, y_i), aircraft velocities v_i, and route information. Route information included the combination of routes for both aircraft, defined as C_j, where j represents a given route combination. From this, we can see that the state-space for the main agent is constant, since it only depends on the number of pixels in the screen-shot. Suppose there are $m \times m$ pixels in the screen-shot and n number of aircraft, then the state-space can be represented with $m \times m$ numbers for the main agent and $2 \times n + n + 1$ for the nested agent. Figure 3 shows an example of a state in the NASA Sector 33 game environment.

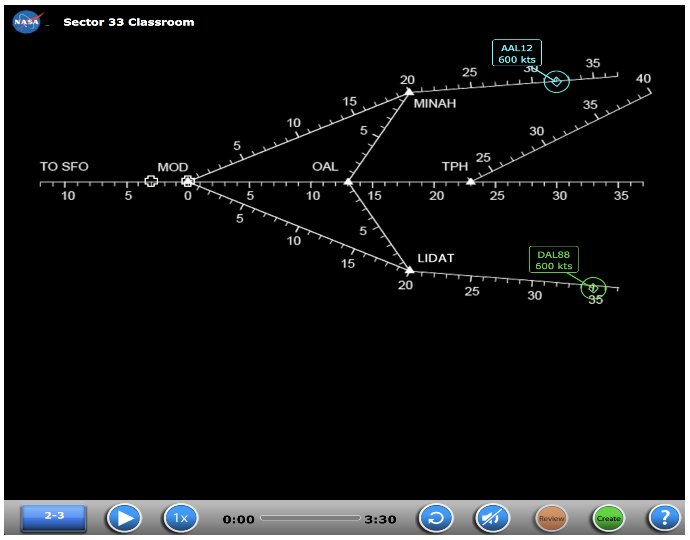


Fig. 3 Example of a state in the NASA Sector 33 game environment.

If we consider Figure 2 as an example, we can acquire all of the state information we need from the game-screen. For the main agent, the state will be represented as follows:

$$S_M = (p_1, p_2, p_3, ..., p_{m\times m}),$$

where p_k represents the intensity tuple of pixel k. For each pixel in the game screen, there is an associated pixel intensity tuple (red, green, blue) that contains the intensity for each color ranging from [0, 255]. For example, the color green is represented as (0, 255, 0), the color blue is represented as (0, 0, 255) and the color red is represented as (255, 0, 0). For the nested agent the state is defined as:

$$S_N = (x_1, y_1, x_2, y_2, v_1, v_2, C_j),$$

where the subscript represents a specific aircraft and j is the combination of route that the aircraft will take.

2. Action Space

At each time-step, the main agent and nested agent can make a decision to change the route of the aircraft and change the speeds of the aircraft, respectively. The only difference is the decision time-step for the main agent and nested agent. The main agent takes one action every episode, where an episode is defined as an entire play through the game. The nested agent takes one action every four seconds within the episode to provide more control over the aircraft once the route combination is determined. The action-space for the main agent can be defined as follows:

$$A_M = (C_1, ..., C_j)\ \forall\ j,$$

where j is the number of route combinations for the aircraft. If we consider the example in Figure 2, the action-space for the main agent will be:

$$A_M = (C_1, C_2, C_3, C_4).$$

This is because each aircraft can take two unique routes, which equates to four unique route combinations.

For the nested agent the action-space is defined as:

$$A_N = (U_1, ..., U_k)\ \forall\ k,$$

where we define U as all of the possible combinations of speeds for the aircraft and k as a unique speed combination.

3. Terminal State

Termination in the episode could be achieved in three different ways:

- Goal reached (optimal) - All aircraft made it to their final metered position, maintained safe separation, and avoided collision. This meant that:

$$|g_{x_i} - x_i| = 0,\ \forall i$$

- Out of time (feasible) - The aircraft did not arrive at their final metered position, but might have given more time.

$$|g_{x_i} - x_i| > 0,\ \forall i$$

- Collision - The aircraft collided with one another.

$$\sqrt{(y_f - y_i)^2 + (x_f - x_i)^2} < \delta, \quad \forall\ i \neq j,$$

where δ is in terms of the pixel distance, and g_{x_i} is defined as the goal position of aircraft i.

By observing the current state, we were able to see if any of the terminal states were obtained. For example, let the current state be defined as $(x_1, y_1, x_2, y_2, v_1, v_2, C_j)$, then if

$$|g_{x_1} - x_1| + |g_{x_2} - x_2| = 0,$$

we have obtained the optimal solution to the problem.

4. Reward Function

The reward function for the main agent and nested agent needed to be designed to reflect the goal of this paper: safe separation, minimizing delay of arriving at final metered position, and choosing the optimal route combination. We were able to capture our goals in the following reward functions for the main agent and nested agent:

Main Agent

$$r = \frac{1}{\sum_{i=1}^{N} |g_{x_i} - x_i|}$$

Nested Agent

$$r = 0.001 \sum_{i=1}^{N} v_i - 0.6,$$

where N is the number of aircraft. In the reward for the nested agent, we included two constants (0.001 and -0.6). The reason for adding the factor of 0.001 is to scale the rewards between [-1, 1]. The addition of -0.6 is to penalize slower aircraft speeds and to reward faster aircraft speeds. These rewards were obtained at each time-step for the main and nested agent. If a terminal state was reached, then there was an additional reward that was added for each scenario: -10 for collision, -3 for out of time, and +10 for optimal solution. With these reward functions, the AI agent will prioritize choosing the route combination that allows the aircraft to arrive as quickly as possible, as well as, choosing the fastest speed for the aircraft without creating any conflict.

IV. Solution Approach

To solve the NASA Sector 33 environment, we designed and developed a novel reinforcement learning algorithm called the *deep nested agent*. In this section we introduce and describe the algorithm, then we explain why this algorithm is needed to solve this game.

A. Deep Nested Agent

To formulate this environment as a reinforcement learning problem, we found that we were unable to formulate this environment as a typical single agent environment due to the non-Markovian property that this problem involves. The route change for an aircraft can only happen during a small window of time, therefore, if formulated as a single agent problem, the action of changing routes would not be chosen nearly as often leading to a very slow convergence time.

To solve this problem we used a main agent who has a second agent *nested* within. The main agent will take an action (changing route) and then the nested agent will control the actions of changing speeds. One of the important differences between the main and nested agents is the state-space. The state-space for the main agent can be represented as screen pixels or in terms of the aircraft positions and speeds. The nested agent has the same state as the main agent if we use the aircraft positions and speeds, but we also add another dimension to the nested agent state. This new dimension we add is the action of the main agent.

We do this to decouple the action sets of changing route and changing velocity. Then we can have the main agent take one action in the beginning of the episode, followed by the nested agent adding this action to it's state and proceeding with the actions of changing speed. Figure 3 provides a diagram of the progression of information from the main agent to the nested agent. We can see the initial state s_M is input into the main agent, then the main agent takes action a. From there, action a is now included into the nested agent state, s_N. From there, the nested agent is able to make all successive decisions with this information. Algorithm 1 provides a pseudo-code for formulating a deep nested agent reinforcement learning problem and Figure 4 shows the progression of information from the main agent to nested agent.

$$\begin{tikzcd} {} \arrow[r, "s_M"] & \boxed{\text{Main Agent}} \arrow[r, "a"] & \boxed{s_N = [s_M, a]} \arrow[r, "s_N"] & \boxed{\text{Nested Agent}} \end{tikzcd}$$

Fig. 4 Progression from main agent to nested agent.

=

Algorithm 1 Deep Nested Agent

```
Initialize: Main Agent
Initialize: Nested Agent
Initialize: $s_M$
reward = 0
number of episodes = $n$
for $i = 1$ to $n$ do
    $a_M = ChooseAction(MainAgent)$
    $s_N = [s_M, a_M]$
    repeat
        $a_N = ChooseAction(NestedAgent)$
        $s', r_N$ = Evolve state based on simulation
        receive main agent reward = $r_M(s_N)$
        $reward = reward + r$
        $update$(Nested Agent)
    until Terminal
    $update$(Main Agent)
end for
```

B. Overall Approach

In our formulation of the deep nested agent, we decided to use the game screen (raw pixels) as input for the main agent. By using the game screen, the main agent can “see" and “comprehend" the air traffic situation by abstracting the important features through the hidden layers of double deep q-network (DDQN), such as relative aircraft positions without explicitly providing the information. The main agent is then able to make route selections for all aircraft at the output layer. In this way, the main agent integrates the conflict detection task (“seeing" and “comprehension") and the first part of conflict resolution task of route selection (“decision making").

With the route combination selected by the main agent, the nested agent will include this information to its current state and proceed with learning how to compute the speed adjustment advisories.

V. Numerical Experiments

A. Interface

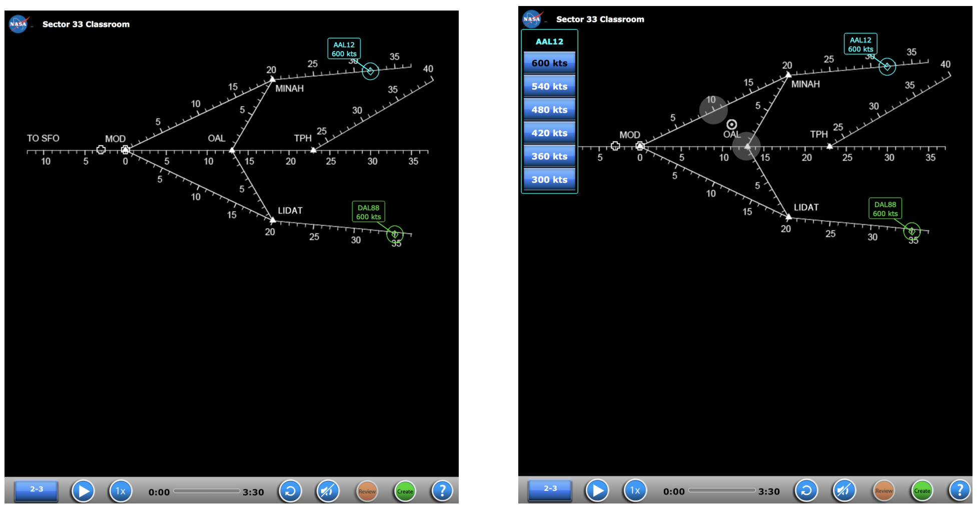
To utilize the reinforcement learning techniques, we needed to convert the NASA Sector 33 game to an environment where we could interact with it and allow the AI agent to learn. With the game itself being web-based, this increased the complexity of the problem, since many games already have built in code simulators to allow for fast computation. To formulate this environment, we created our own interface to communicate between the web-based game and code which allowed the agent to learn. If we let i represent a specific aircraft, the information we needed from the game was:

- Aircraft positions - (x_i, y_i)
- Route information - R_j
- Aircraft Velocities - v_i
- Game controls (start, stop, etc.)
- Goal position - (g_{x_i}, g_{y_i}).

Once we obtained the route information of each aircraft, when then determined the unique combinations of routes that the aircrafts could have, which we denote as C_j, where j is a given route combination. For example, consider figure 2(1). C_1 could represent the blue aircraft on route from MINAH to MOD and the green aircraft on route from LIDAT to MDO. C_2 could be the same route for the blue aircraft, but now the green aircraft is on route from LIDAT to OAL.

The environment information needed to be collected in terms of the screen pixel location, that way we could utilize a python module: *pyautogui* to operate mouse clicks and navigate to locations on the screen. For the game controls, we could hard-code their locations in since they were static. By doing this we could create helper functions to control all of the static game controls to simplify the interface.

Route information needed to be observed by the user and then hard-coded into the interface. For a given aircraft, if one was able to change its route, then there would be a slider that was activated when the user clicked on the aircraft. We then hard-coded the slider location into the interface so that if we wanted to change the route then we could easily do so. Figure 5 (a) and figure 5 (b) show the game screen when the route option is not shown and when it is shown, respectively.

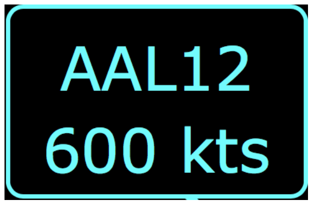


(a) The game screen when the route information is not present. Notice that the velocity information is also hidden.

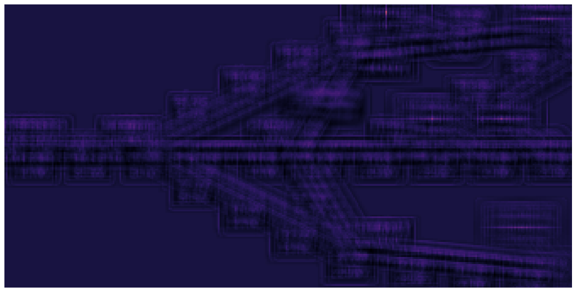
(b) The game screen when the route information is shown. The gray circles are where to drag and drop the route to change it. It is only shown when the user clicks on the aircraft. We can see this also activated the velocity information.

Fig. 5

The most difficult information to obtain was the aircraft positions due to the fact that they could not be hard-coded, since at every time-step they were changing. To find the aircraft's positions, we utilized a module in python called *OpenCV*. By obtaining the images of the individual aircrafts and a screen-shot of the current game image, this module allowed us to use template matching to find where the aircraft was located on the screen. Figure 6 (a) shows one of the individual aircraft images we used and figure 6 (b) shows the result of the template matching. There were many different fine-tuning parameters as different colored aircraft's would respond differently to the template matching techniques. Once this information was obtained, we could easily calculate the distance to the goal, the distance from one aircraft to the next, and also obtain the velocity information.



(a) Individual aircraft image to use for the template matching.



(b) Result of the template matching. We can see that from the template matching, the game screen was overlayed with the aircraft image until it found the most probable match which is at the top of the screen.

Fig. 6

The speed information for the aircrafts are hidden until one clicks on the aircraft itself. With the aircraft's position obtained, we could then access the velocity by making a click on the chosen aircraft to obtain the information.

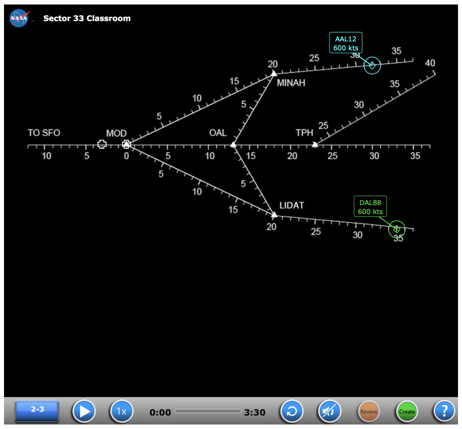
B. Environment Setting

For this problem in NASA Sector 33, we discretized the environment into episodes, where each run through the game counted as one episode. We also introduced a time-step, Δt, so that after the nested agent selected an action, the environment would evolve for Δt seconds until a new decision was made. This was to allow for a noticeable change in state from $s_t \rightarrow s_{t+1}$, as well as, allow the game to be played at different speeds. There were many different parameters that had to be tuned and selected to achieve the optimal solution in this game. We implemented the DDQN concept that was mentioned earlier, with a hidden layer of 64 nodes. We used an epsilon-greedy search strategy for both the main and nested agents. For the nested agent, epsilon started at 1.0 and exponentially decayed until 0.01. For the main agent, epsilon also started at 1.0 and exponentially decayed to 0.15. The reason for the main agent decaying until 0.15 was to allow for the slightly larger probability of exploring the other routes even when many episodes had already been run. In harder game environments, we could obtain a near optimal feasible solution on the non-optimal route. By forcing the lower bound on epsilon, this ensured that we were able to converge to the optimal route. We used a *tanh* activation function for the nested agent and the *ReLU* activation function for the main agent. This was because for the main agent, instead of using a multi-layer perceptron network, we used a convolutional neural network whose input was the current game screen. Our memory length for both the main and the nested agents was set to 500,000 to allow for a large number of the previous transitions to be made available for training.

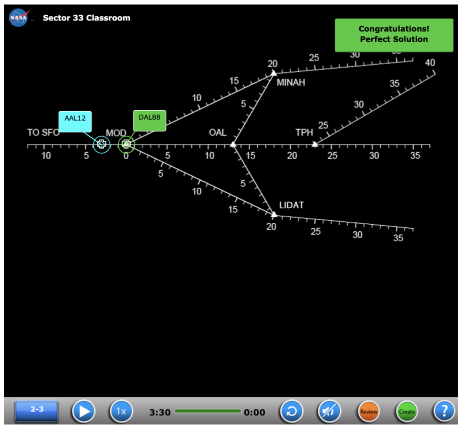
C. Case Study: two aircraft with four routes

In this problem, we once again considered two aircraft except this time both aircraft have the ability to change routes. Figure 7 (a) shows the game screen for this problem. With both aircraft able to change routes, this leads to four different route combinations and greatly increases the complexity of the problem. What also makes this problem interesting is that even if the aircraft take the incorrect route, a near-optimal feasible solution can be obtained which makes choosing the route more difficult.

By training the AI agent on around 2000 episodes and choosing a time-step of four seconds, we were able to obtain the optimal solution for this problem. The number of episodes required to achieve the optimal solution increased from the first problem, but this was due to the complexity of the problem and the fact that different routes could achieve very similar results. It is also important to note that in this problem, if we chose a different route, the AI agent would achieve the optimal solution specific for that route. Therefore, in a real world scenario, if there was a last minute decision to change from the optimal route to a feasible route, the AI agent would be able to maintain safe separation and minimize delay on this new route. Figure 7 (b) shows the game screen after obtaining the optimal solution to this problem.



(a) The game screen for the Case Study. Problem 2-3 in the NASA Sector 33 application.



(b) The game screen after obtaining the optimal solution in the Case Study.

Fig. 7

D. Algorithm Performance

In this section we compare the algorithm's performance on the two aircraft problem with one nested agent and the five aircraft problem with multiple nested agents. Figure 10 shows the score per episode during training for the Case Study.

We can see that throughout training the score tends to oscillate frequently, but is on an increasing trend. This is due to the value of ϵ being close to 1 and as this value decreases through the episodes, the score increases significantly. We define the score in Figure 8 to be the cumulative reward at the end of each episode. The score involves the reward of the nested agent at each time step, as well as, the termination rewards: -10 for collision, -3 for out of time, and +10 for optimal solution.

VI. Conclusion

A novel deep reinforcement learning algorithm with nested agent is proposed in this paper to sequence and separate aircraft as a core component in an autonomous air traffic control system. The problem is formulated in a reinforcement learning framework with the actions of changing aircraft route and adjusting aircraft speed. The problem is then solved by using the deep nested agent algorithm. Numerical experiments show that this proposed algorithm has promising performance to help aircraft maintain safe separation, resolve conflict, and arrive at their final metered positions in different air traffic scenarios. The proposed algorithm provides a potential solution framework to enable autonomous sequencing and separation for a fully automated air traffic control system in a structured airspace.

According to our knowledge, the major contribution of this research is that we are the first research group to investigate the feasibility and performance of autonomous aircraft sequencing and separation with deep reinforcement learning framework to enable an automated, safe and efficient airspace. In addition, we propose the novel hierarchical architecture of deep nested agent, which is demonstrated capable of solving complex online sequential decision making problems. The promising results from our numerical experiments encourage us to conduct future work on more advanced air traffic control simulators that can model operational uncertainties.

VII. Acknowledgments

The authors would like to thank Michelle Eshow for pointing us to the NASA Sector 33 simulation environment. We thank her 35 years service at NASA Ames Research Center and wish her all the best for the retirement life.

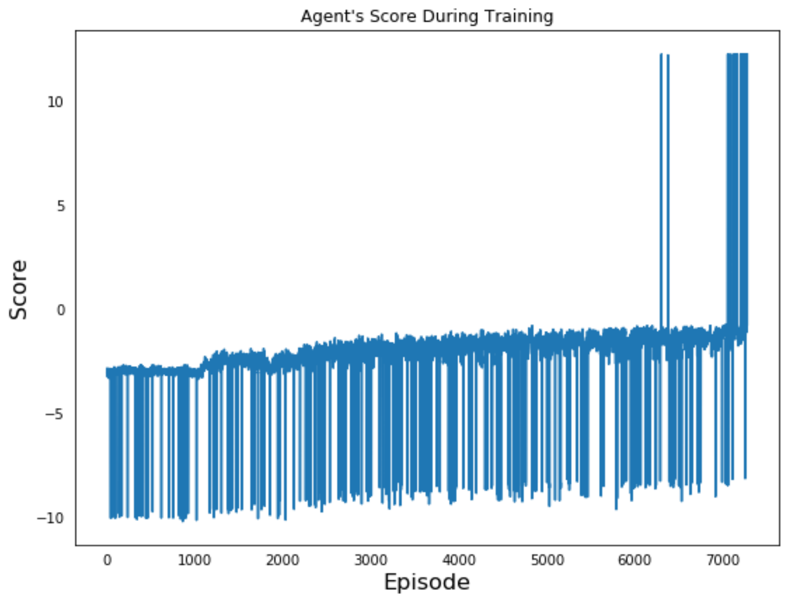


Fig. 8 Agent's score throughout the training process for the Case Study. We can see that as the episode number increases, the score increases as well.

References

[1] Erzberger, H., and Lee, H. Q., "Terminal-Area Guidance Algorithms for Automated Air-Traffic Control," 1972.

[2] Erzberger, H., "ATC automation concepts," 1990.

[3] Erzberger, H., "CTAS: Computer intelligence for air traffic control in the terminal area," 1992.

[4] Erzberger, H., "Automated conflict resolution for air traffic control," 2005.

[5] Watkins, C. J. C. H., "Learning from delayed rewards," Ph.D. thesis, King's College, Cambridge, 1989.

[6] Kochenderfer, M. J., Amato, C., Chowdhary, G., How, J. P., Reynolds, H. J. D., Thornton, J. R., Torres-Carrasquillo, P. A., Üre, N. K., and Vian, J., *Decision Making Under Uncertainty: Theory and Application*, 1st ed., The MIT Press, 2015.

[7] Van Hasselt, H., Guez, A., and Silver, D., "Deep Reinforcement Learning with Double Q-Learning." *AAAI*, Vol. 16, 2016, pp. 2094–2100.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al., "Human-level control through deep reinforcement learning," *Nature*, Vol. 518, No. 7540, 2015, p. 529.

[9] Hasselt, H. V., "Double Q-learning," *Advances in Neural Information Processing Systems*, 2010, pp. 2613–2621.