UNIVERSITY OF EXETER

Computer Science Department

B.Sc. Computer Science
Computer Science Department

Navigating the Skies: Reinforcement Learning with Double Deep Q-Networks for Air Traffic Control

Candidate

Oskar Aaron Oramus

Student ID

Supervisor

Prof Richard Everson

University of Exeter

Co-supervisor

Academic Year
2022/2023

Abstract

This paper proposes a novel approach to air traffic control using the Deep Double Q-Learning algorithm, in a multi-agent reinforcement learning scenario using a distributed system. The approach aims to improve conflict resolution problems in air traffic systems. The performance was evaluated using a swap position scenario 3-dimensional scenario, with 2 aircraft and was analysed based on collisions, fuel efficiency, safety and reliability. Results show that the approach was able to get to the target 87% of the time, with 41ms of inference time, 3% crash rate and 4% of the time it would enter the wrong sector. The resulting distance flown on average was around 1.7% more than the optimal distance, indicating that this approach could guide aircraft along efficient flight paths, whilst taking collision avoidance into consideration. The proposed approach has the potential to contribute to more efficient and effective air traffic control systems, benefiting airlines, passengers and the environment.

	Yes	No
I certify that all material in this dissertation which is not my own work has been identified.	☑	☐
I give the permission to the Department of Computer Science of the University of Exeter to include this manuscript in the institutional repository, exclusively for academic purposes.	☑	☐

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

ATC Air Traffic Control

MDP Markov Decision Process

RL Reinforcement Learning

DQN Deep Q-Network

DDQN Double Deep Q-Network

MARL Multi-Agent Reinforcement Learning

TD Temporal Difference

1 Introduction

Air traffic control (ATC) is a crucial aspect of aviation safety and efficiency. It involves managing the movement of aircraft in both air and ground to ensure planes arrive at their destinations safely and efficiently. Effective ATC systems are essential for passenger and crew safety, as well as for minimising delays which in turn will maximize the available space in the airport. The way ATC does this is by having a lot of controllers communicate with other members of the aviation industry, such as ground crew, pilots and maintenance personnel to coordinate the safe movement of the aircraft. This involves monitoring of the aircraft using radars and making adjustments to their flight paths as needed. The job requires rigorous training to ensure that the trainee is qualified and fit enough for the job. ATC applicants must be 30 years old or younger even to be considered [Everyday Aviation]. The role of an air traffic controller can be incredibly demanding and stressful, a fatal mistake could result in many lives lost. A NASA study reveals that almost 20% of controllers made significant errors due to persistent fatigue or sleep deprivation [NASA, 2011]. A minor error caused by exhaustion almost resulted in a fatal accident in France in July 2020, with the aircraft narrowly avoiding a collision by 100 meters [BEA Aero, 2020]. In busy airport environments, there may be dozens or even hundreds of planes taking off every hour, controllers must be able to accurately and quickly assess complex situations and make decisions that keep everyone safe. With the predicted annual growth of 3.6% in air traffic, the job is not getting any easier [Airbus, 2022]. Additionally, all air traffic controllers must be able to speak English proficiently and have a clear accent, which can limit the pool of potential candidates [Raga, 2019].

A well-designed system could be capable of being more efficient than humans in flight paths, saving fuel and reducing carbon emissions. At first, the automated systems could focus on solving the simpler scenarios automatically and leave the more complex ones to humans. These kinds of systems could also be used in other areas or industries that deal with aerial vehicles.

This research will provide critical insights which could contribute to the development of new systems that could help air traffic controllers make better decisions. It primarily focuses on reinforcement learning as the method, which is capable of self-learning and adapting to unseen situations, which could be viable in sudden environmental changes like the weather or turbulence.

2

Literature Review

This section provides an overview of the existing research on double deep Q-learning and its predecessors in reinforcement learning. In the next section it will describe how this literature is used to conduct research on air traffic control, and how the environments are built.

Reinforcement learning (RL) is a branch of machine learning that specializes in general-purpose automated decision-making. RL algorithm can learn from experience by interacting with its environment and receiving feedback in the form of reward signals. By using RL, the algorithm can adapt to changing conditions and figure out the optimal decision-making process that yields the best outcomes. Given the correct training process, it can generalise the problems very well which in succession will perform well even on unseen problems.

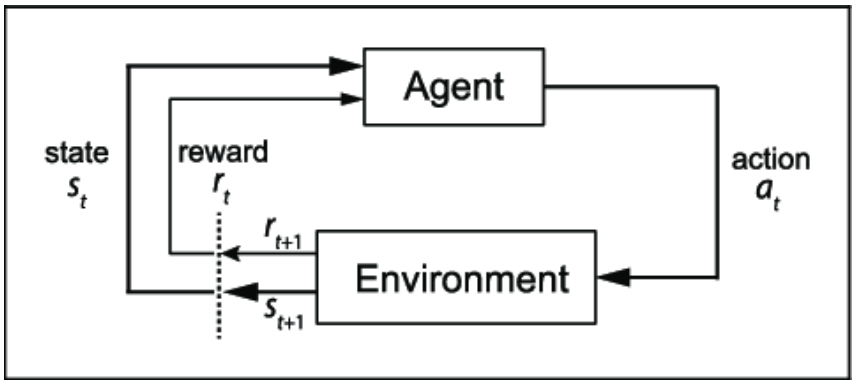

Figure 2.1: Learning cycle of a reinforcement learning agent

Figure 2.1 shows a learning cycle of a reinforcement learning agent. An agent at time t will take an action a_t, the environment will calculate the new state of the simulation and output a reward r_{t+1} and new state s_{t+1}. The agent will receive the new state as input and the whole cycle repeats until the simulation is either terminated or completed.

2.0.1 Markov Decision Process

The environment always respects the Markov Decision Process (MDP), i.e. it has a memoryless property of a stochastic process. This means that history does not affect the current move action, i.e. it does not matter how we got into this situation, what matters is how we get out of it. It is usually denoted by tuple $\langle S, A, P, R, \gamma \rangle$, where S is the set of possible states in the environment, A is the set of actions that we can perform, P is the probability transition function from one state to another. The conditional distribution $P(s'|s, a)$ of the next states is

given by the current state and actions we can take (s' refers to the state during the next time step). In a deterministic environment like chess, one of the actions will have a probability of 1, and the others 0. This is because we have full information and can always perform the perfect action. However, in a stochastic environment, our conditional distribution will have a variety of probabilities as we may have to take a risk in some cases.

R_t is the immediate reward at time t and $\gamma \in [0, 1]$ is the discount factor. Which tells the agent how much to care about future rewards. The discount factor is bounded to be smaller than 1 as a trick to make the infinite sum finite. The cumulative reward is the discounted sum of the reward accumulated in the given episode. The return that the agent receives from time step t is represented by $G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. The expected value is represented by $E[G_t] = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...]$.

Policy π is the agent's strategy, it is a map from state to action. An optimal policy is denoted by π_* and maximizes the discounted reward in the episode. There may exist more than one optimal policy, but we are guaranteed at least one [Silver, 2015].

We want a policy that maximizes $R = \sum_{t=0}^{n} \gamma^t r_{t+1}$. $V_\pi(s)$ is our state value function. It will give us the expected returns starting from state s and following policy π. $V_\pi(s) = E_\pi\{G_t | s_t = s\}$. Let's assume $V_{\pi*}(s)$ is our optimal value function. From it, it is possible to get the optimal policy. At s_t, choose an action that maximizes the value function of $V_{\pi*}$ using greedy behaviour. The optimal policy is then the greedy policy with respect to $V_{\pi*}$.

$$\pi^*(s) \text{ chooses } a \text{ s.t. } a = arg\max_a [E_\pi(r_t + \gamma V(s_{t+1}) | s_t = s, a_t = a)] \tag{2.1}$$

2.0.2 Q-Learning

Q-learning is a model-free, off-policy algorithm that solves for $\pi*$. Off-policy algorithms tend to find the best actions based on the state, by coming up with their own rules that may operate outside the policy given. This means there is no need for an actual policy, hence it's called off-policy [Banoula, 2023]. Q-Value $Q(S, A)$ determines how good it is to take action A at a particular state S. When the Q function converges to an optimal value, we can solve for $\pi*$ by picking the highest Q value action at state s_t, using greedy evaluation.

The optimal action-value function satisfies the Bellman Optimality Equation:

$$q_*(S, A) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s', a')\right] \tag{2.2}$$

Where q_* denotes the true value of action a at state s [Sutton and Barto, 2018][33]. The equation tells us that at time t, for any state-action pair (s, a) the expected return is the expected reward R_{t+1} and the maximum of expected discounted return [Dave, 2021].

Temporal difference (TD), is a reinforcement learning method, which uses a difference between its current understanding of the environment and the actual outcome. For example, if the agent predicts 0.5 as the Q-value of an action, but the actual value was 0.7, it would calculate the error $0.7 - 0.5 = 0.2$, and update its value by a learning rate α.
The update equation for q-learning is [Banoula, 2023]

$$NewQ(s,a) = Q(s,a) + \alpha\left[R(s,a) + \gamma \max Q'(s',a') - Q(s,a)\right] \tag{2.3}$$

The algorithm essentially keeps track of a table with all possible states and actions, it then tries a lot of things during the exploration stage and notes the rewards. It is then saved in the table using the formula 2.3. The learning rate ensures that if the environment is random, the values won't explode out of proportion, but instead will slowly be changed. The issue with the basic Q-learning algorithm is that the space complexity is $O(nm)$ with n states and m actions. It also only deals with discrete values which makes some environments very difficult to solve. Another issue is that Q-Learning can only decide the Q-value of (s,a) if it explores that tuple. Deep Q-Learning (DQN) can make estimated guesses even for states we have never encountered.
DQN builds on top of the Q-Learning, it replaces a table with a neural network that estimates all of the q-values for any given state-action tuple. Action is simply the highest q-value that the network returns. Update formula changes, we now use the mean squared error (MSE) of the predicted Q-value, and actual. This is given by $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. For the training, we use an ϵ-greedy strategy, where we choose a random action with probability ϵ and the policy action otherwise. ϵ usually starts at 1, and goes down to 0.1 [van Hasselt et al., 2015]. The q-value update formula becomes:

$$Q_{target} = R + \gamma \max(Q_{next}) \tag{2.4}$$

Where Q_{next} is predicted Q-values for the next state. However, the formula always chooses the maximum Q-value to update itself. If our environment is noisy (has random/uncertain rewards), or the neural network's initial state coincidentally has a high value, our Q-values will be over-estimated as they pick up all of the noise and take a maximum of it. To combat this, Double Deep Q-Network has been created.

2.0.3 Double Deep Q-Network (DDQN)

DDQN is a model-free off-policy algorithm. Model-free algorithms focus on the consequences of their actions and attempt to minimise them. Off policy here refers to a method of using two policies to solve a problem: a behaviour policy and a target policy. DDQN is able to use every experience as much as possible, making it sample efficient. This is quite important in the context of ATC as gathering experience data for learning can be time-consuming. It also employs an exploration-exploitation strategy using an ϵ-greedy approach, meaning it can adapt to dynamic environments. The optimal policy will change depending on many factors in the real world like congestion, the number of aircraft in the air, weather conditions and other

factors. The hyperparameters are also very easy to tweak, making it easier to focus on the reward function.

DDQN has been successfully applied in a variety of fields. For instance, it has been used to optimize wireless network access configuration, allowing for more efficient network usage [Wang et al., 2021]. In healthcare, DDQN has been employed for hemodynamic management in sepsis patients, improving patient outcomes [Lu et al.]. It has also been used to enhance the precision of airdropping using UAVs [Ouyang et al., 2022]. The success of DDQN in various fields will make a strong foundation for this project.

Algorithm 1 Double Deep Q-Network with Replay Memory

Initialize replay memory R with capacity N
Initialize model Q with random weights: M
Initialize target model Q' with random weights: M_T
for each episode **do**
  Observe the first state s_1
  repeat
    Choose action A_t using policy, using ϵ-greedy approach
    Take action A_t
    Observe reward R_t and state S_{t+1}
    Store (S_t, A_t, R_t, S_{t+1}) in R
    Calculate target:

$$T_i = \begin{cases} R_i & \text{if } S_{i+1} \text{ is terminal,} \\ R_i + \gamma \max_a Q(S_{j+1}, argmax_a Q(S_{j+1}, a|M)|M_T) & \text{otherwise} \end{cases}$$

    Update network parameters M with loss $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
    Update $M_T \leftarrow M_T \cdot (1-\tau) + M \cdot \tau$
  until terminated or done
end for

The basic idea is to split the predicted value and the target value between two neural networks. It is much harder to learn when the target keeps moving, as in the case of DQN where the $Q(S, A)$ and $Q(S', A')$ keep changing, which moves our loss [van Hasselt et al., 2015].

Q_{θ^-} denotes the target network, Q_θ is the model network.

In DDQN, we instead have one neural network estimate the target $r + \gamma \max Q(S', A')$ and another one estimates the prediction $Q(S, A)$. To keep the target network up to date, we update it once in a while. There are two ways, either hard copy every n^{th} episode (for example 100), or soft τ update, each episode update $Q_{\theta^-} = Q_{\theta^-}(1-\tau) + Q_\theta\tau$. With $\tau \in [0, 1]$ (typically $\tau = 0.01$).

The neural network that is used in the DDQN algorithm, uses ReLU as its activation function. It is defined as $f(x) = max(0, x)$. ReLU activation solves the problem of "Vanishing Gradient", which can occur when using other activation functions like Sigmoid or Tanh. The vanishing gradient happens when gradients become very small. When using the derivative of Sigmoid or

Tanh, there is a chance the gradients could become tiny, and the network would stop learning because changes are too small to have any effect [Basodi et al., 2020]. However, with ReLU, our gradients are always either 0 or 1. The final layer uses a linear activation, as we need the network to predict the action value, which could be any real value. Both networks will be using the Adaptive Moment Estimation (ADAM), which uses the estimations of the first and second moments of the gradient to adapt the learning rate.

2.0.4 Experience Replay

To improve the learning speed of DDQN, we use experience replay which is used to store agent's experiences $\langle S, A, R, S' \rangle$. Neural networks require a lot of data to converge, instead of learning as we experience observations, we instead store them in a list and the neural networks use those as batches to update their gradients. A typical experience replay size is around 250 000, depending on the problem. This also helps in experiences that happen rarely, for example, if our agent has accidentally found a solution in the first few episodes, the experience replay will make sure the neural network realises that this was a good action.

Prioritised Experience Replay (PER)

Experience replay samples each experience at the same frequency as it was originally experienced [Schaul et al., 2016]. We can score each experience based on how useful it was and choose the more useful ones more often. Our error is given by $error = Q(S, A; Q_\theta) - Q(S, A; Q_{\theta^-})$. To avoid division by zero when sampling probabilities, our probability becomes $p = |error| + 0.1$. $Pr(i) = \frac{p_i^a}{\sum p^a}$ where a is our priority sampling factor, with $a = 0$ becoming normal experience replay, and $a = 1$ using full priority sampling. To avoid the gradient calculations becoming too big on commonly sampled experiences, our weight update is $w_i = (\frac{1}{N} \times \frac{1}{Pr(i)})^b$. Exponent b starts off as 0, and slowly goes to 1 as the agent learns as we want to reduce the step size for higher probability experiences to avoid excessive update steps in the network.

2.0.5 Multi Agent Reinforcement Learning (MARL)

MARL refers to an environment where multiple agents can interact. For example, in a game of pong, we can have two agents learn how to get better by utilizing self-play. This technique was used in 2016 by AlphaGO. The algorithm was able to win most of the games, given that the complexity of Go is much bigger than chess (10^{360} vs 10^{111}), this makes the algorithm very capable of solving extremely hard problems [Silver et al., 2016] [Koch, 2016].

MARL can be done either using distributed or centralised systems. Distributed methods involve having one neural network that controls each agent individually. The main advantage of this method is scalability, in theory, if we train 2 agents we can add any amount and the system should still work. However, it can be harder to solve a problem when the neural network has no idea about the other agent's thought process. For example, if we are about to collide, we need to predict what the other agent will do, to avoid the collision.

The centralised method includes one network controlling all of the agents. The main advantage is that the whole system can act much smarter and come up with potentially better solutions, however, if we add or remove any number of agents, everything needs to be retrained.

In some scenarios, the centralised method might involve fewer parameters making it faster to train. In the case of aeroplanes, the distributed method might require additional information like other planes' flight direction, speed or distance. But with the centralised method we can input the most basic information like $\langle position(x, y, z), rotation(x, y, z), target(x, y, z)\rangle$, the neural network should be capable of figuring out how to use this data to solve the environment.

There is also a hybrid version, where we can train a centralised solution for 1, 2 and 3 agents. Then in the simulation, we perform centralised execution on the local group of planes. Giving us the advantages of both methods.

The study by Brittain and Wei [Brittain and Wei, 2019] proposes an autonomous air traffic control system that ensures safe separation between aircraft in high-density air space sectors. Their framework uses the Advantage Actor-Critic (A2C) model, which uses a loss function from Proximal-Policy Optimisation (PPO). Additionally, they use centralised training but a decentralised execution scheme, which makes the solution scalable. They report impressive results, with their framework being able to resolve 99.97% of conflicts at intersections, and 100% at merging points. Their solution consists of a simple piece-wise reward function:

$$f(x) = \begin{cases} -1 & d_{co} < 3 \\ -\alpha + \beta \cdot d_{co} & if 3 \leq d_{co} < 10 \\ 0 & otherwise \end{cases} \tag{2.5}$$

With α and β being small values. However, the planes always fly a predetermined path, and the only actions they can take are to speed up, slow down or maintain the current speed. This system could be employed in conjunction with another model that controls the movement of planes. The tests were also ensured to have a possible solution, which didn't test the framework to its limits. The study has only shown results on two hard-coded flight path scenarios, which may not represent all possible scenarios.

Another study by [Mukherjee et al., 2022] has explored using DDQN to assist air traffic control operators by resolving conflicts and finding alternate flight paths. The study has considered using algorithms like Asynchronous Advantage Actor Critic (A3C) and PPO but found that they "suffer from the problem of sample complexity" [Mukherjee et al., 2022][2]. Whereas the Deep Deterministic Policy Gradient (DDPG) is sample efficient but suffers from hyper-parameter sensitivity. Ultimately DDQN was chosen for its simplicity. They have found that agents were able to keep a minimum separation distance, but performance dropped as the aircraft distance got smaller.

This study will use a modified version of reward functions presented in [Ghosh et al., 2021] to fit our specific research context and experimental setup. Their study has proposed a set of rewards that have shown promising results in multi-agent reinforcement learning for air traffic control. The action space in the paper is similar to that of [Brittain and Wei, 2019], i.e. ⟨speed

up, slow down, keep speed⟩. This project will focus on giving more freedom to the agents, allowing them to control the steering angles as well as the speed.

2.0.6 Project specifications and objectives

The project will focus on implementing DDQN in order to automate air traffic control. This will involve the development of a simulation environment to model air traffic scenarios and test the performance of the DDQN algorithm. All of the specifications are outlined here.

The application will be designed to work with the terminal, as it will be run on the University machines using SSH. There will be a general config file which will be used, this will make it easier to run specific experiments. The three main files will be "main.py" which is responsible for training the agent, "player.py" which visually renders the agent, does not train the neural network and uses $\epsilon = 0$, and finally "benchmark.py" which will take information returned by the environment and work out averages for the performance.

The objectives of the research will be to create a system capable of fully manoeuvring the aircraft in a way that does not conflict with other aircraft and is able to efficiently get to the target position within the time frame. The highest priority in this research will be safety and reliability. The performance of the agents will be evaluated on a pre-trained model. Metrics such as the number of collisions, whether the target was reached, the flight efficiency and the number of times an aircraft leaves its designated sector, will be recorded and then analysed. Flight efficiency will be calculated using $\frac{\text{distance flown}}{\text{optimal distance}}$, which will show how much excess flying the agent has done, the closer to 1 the better its flying is. All environment scenarios will be trained and tested against a "swap position" scenario, where the agents need to land in each other's starting positions. This will force them to resolve a collision, and we can be sure that if the distance is minimised, the collision has been resolved in some way. Since the RL algorithms rely on randomness for the exploration, (known as ϵ-greedy exploration), the benchmark model will use $\epsilon = 0$. The solution will be tested many times on random scenarios to ensure consistency and reliability, the average will then be taken of all the metrics and analysed.

The solution will first be programmed to solve a two-dimensional drone environment, allowing for fast iteration (due to the number of dimensions being much smaller in 2D than in 3D). This will also allow for faster iteration of the experimental design of the reward function. Once the drone simulation show progress, the project will be moved towards the three-dimensional aircraft environment. It will first be designed to accommodate one plane reaching its target. After that, another plane will be added, and transfer learning will be utilised to speed up the learning. Transfer learning refers to teaching an agent that can perform a simple task, by transferring the old knowledge to the new (usually harder) task.

According to [TypeIA and rbp], planes need to maintain a separation distance of at least 3 miles laterally or 1000 (304.80 meters) feet vertically. This distance will be represented in the reward function, encouraging planes to keep a minimum separation distance from each other.

There are risks that could impact the project's objectives. Technical difficulties or challenges in the implementation of the DDQN algorithm would be minimised by thoroughly researching

the methods, carefully looking at the pseudo-code proposed in the original DDQN paper (or similar) and also performing baseline experiments that can be compared with other people. In this study, the cart pole and lunar lander are the baseline experiments used to compare results with other, already established solutions. Changes in project requirements or scope could impact the final result. To avoid this, a detailed project schedule with realistic timelines and milestones will be laid out. It will allow for flexibility in case of unforeseen circumstances.

3 Methodology

This section explains how the DDQN algorithm was implemented and tested on air traffic control scenarios. The algorithm has been tested on baseline problems (cart pole and lunar lander) to ensure consistency with results and convergence. It then showcases a drone environment which was then modified into an air traffic control environment. The drone and plane environment uses a custom physics engine which has improved the learning times for the agent. All of the solutions here use the distributed method.

3.0.1 Hyperparameters

The base experiments (cart pole and lunar lander) are using the same set of hyperparameters outlined in the table 3.1. Where t is the episode number, starting from 1.

α	γ	Batches	Memory	τ	Network Size	Epsilon
5×10^{-4}	0.99	64	250,000	0.05	$(obs_n, 64, 64, action_n)$	$0.1 + 0.9 \times e^{-0.01t}$

Table 3.1: Hyperparameters used for base experiments

α refers to the learning rate used by the neural network optimiser. All neural networks in this project use the ADAM optimiser. Batch size here refers to the amount of data that the neural network will use in one training step. The bigger the batch size the better the gradient can be approximated, but the training step will take longer. The best performance is achieved with a batch size of 512 [Stooke and Abbeel, 2019], however, it makes the training process much longer, and the performance improvement isn't always worth the extra time spent training. Most of the suggested hyperparameters here, are the default values used for DDQN applications. The original DDQN paper [van Hasselt et al., 2015] suggests using min epsilon as 0.1, this keeps exploration high even when training for a longer time.

3.0.2 Base Line - Cart Pole Environment

Cart pole problem summary

A pole is attached to a cart, the agent has to balance the pole. The environment is in 2 dimensions. Every frame that the pole is upright (12 degrees to the vertical), gets a +1 reward. The episode is terminated if we reach 200 simulation ticks, the pole falls or the cart goes off the screen. The agent receives 4 observations, ⟨ cart position, cart velocity, pole angle, pole angular velocity ⟩. Its action space (2) is ⟨ Push the cart to the left, push the cart to the right ⟩. It always has to be moving the cart making the problem a little bit more difficult. Once 200 points are reached, the environment is solved. This is a one-dimensional motion as the cart can only move to the side.

Lunar lander problem summary

Once the cart pole problem was solved, it was also tested on the lunar lander to ensure it wasn't just coincidentally getting the right actions. The lunar lander is a 2-dimensional rocket trajectory optimisation problem, it involves landing a spaceship on the landing pad without crashing, by controlling one of the 3 engines, the main engine (located at the bottom of the ship), or the side engines. It has many more dimensions than the cart pole problem (8 vs 4), which greatly increases the complexity of the problem. The observation space (8) includes ⟨ lander coordinates (x,y), lander linear velocity (x,y), angle, angular velocity, left leg contact, right leg contact ⟩. Action space (4) is ⟨ do nothing, fire left engine, fire main engine, fire right engine ⟩. Solving means reaching 200 points. To reach 200 points, the agent needs to land on the pad and come to rest efficiently. The reward for moving from the top of the screen to the landing pad with zero speed gives a 100-140 reward (based on speed). Each leg ground contact is +10, if it comes to rest on the landing pad that's +100. It also gets receives -100 for crashing, -0.3 for firing the main engine, and -0.03 for firing the side engines, this represents using fuel.

The algorithm was tested on the basic cart pole problem. This was done for three main reasons. Firstly we want to make sure the algorithm converges to an optimal policy, which can be easily verified. It is also much easier to debug knowing that the reward function works and that it's most likely a bug in the code. Secondly, there are a lot of results available online for the cart pole problem, making it easy to compare the training times and hyperparameters used. Finally, it will be quickly clear if the algorithm was implemented correctly (results within the first ≈200 episodes).

3.0.3 Baseline Results

The results are shown in figure 3.1 and 3.2. We can see that during the exploration stage, the results are completely random. However, once the neural network has enough data to learn from, we can see a sudden jump in performance. All of the results are noisy due to the epsilon never going below 0.1, which was suggested in the [van Hasselt et al., 2015] DDQN paper. Also

due to the random nature of the problems, some scenarios can be solved quicker or easier than others, causing more noise in the rewards. This is inevitable if we want our agents to learn how to generalise problems better. In compliance with standard results, both environments are considered "solved" when reaching a consistent reward of 200, which has been reached in both cases.

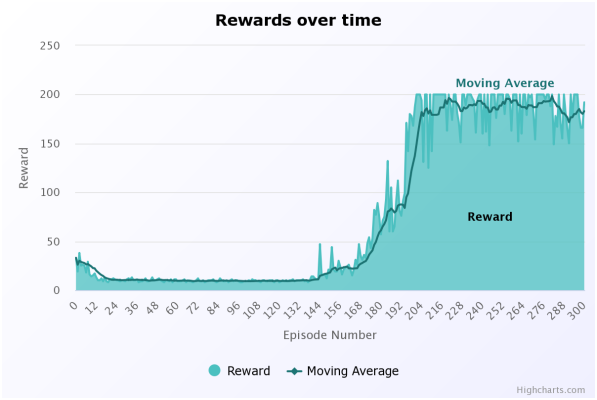

Figure 3.1: Cart pole rewards over time.

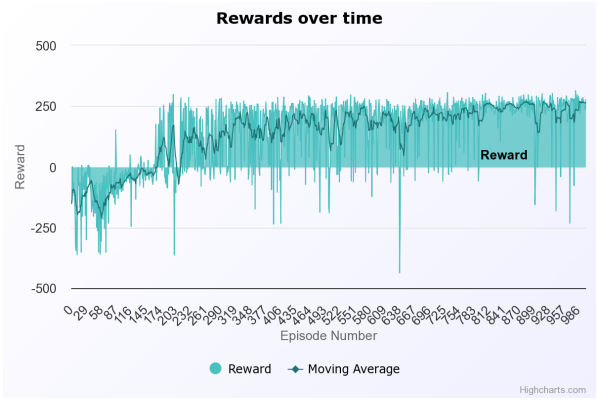

Figure 3.2: Lunar lander rewards over time.

3.0.4 Potential Issues with DDQN

DDQN, as well as other machine learning models, can suffer from "catastrophic inference" [Zhang et al., 2022]. Due to the nature of constantly changing data in reinforcement learning, when a model is doing very well, it might "forget" what failure is, causing a big dip in performance. There are some proposed solutions like inference-aware deep Q-learning (IQ). The simplest solution (and the one used for this study) is to select the best-performing model out of the ones available.

3.0.5 Drone Environment

The DDQN algorithm was first tested on a custom 2D drone environment problem, which made it easy to start off and add complexity. The drone is positioned at a random location inside the 300×300 simulation environment, with at least 30 units away from the edges. The target is also placed with the same rules, at least 60 units away. The action space (5) allows the agent to accelerate ⟨ up, down, left, right,⟩ or do nothing. The observation space (7) is ⟨position(x,y), target position(x,y), velocity(x,y), distance to target⟩ .The acceleration value is 2 units per frame, with a max speed of 10 units. Air resistance is also applied to the agent every frame, slowing the drone down by 10% each frame. The episode terminates whenever the agent leaves the map boundary or 200 frames pass. This experiment used 7 - 64 (ReLU) - 64 (ReLU) - 5 (Linear) neural network model structure.

α	γ	Batches	Memory	τ	Network Size	Epsilon
0.001	0.99	128	250,000	0.05	$(obs_n, 32, 32, action_n)$	$0.01 + 0.99 \times e^{-0.02t}$

Table 3.2: Hyperparameters used for the drone environment

$$f(x) = \begin{cases} 10 & \text{distance} < 10 \\ -1000 & \text{leaves bounds} \\ \max(0.01, 1 - d_0) \times 5 & \Delta \text{ distance} > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.1}$$

The equation 3.1. shows the reward used to train the drone, with $d_0 \in [0, 1]$ being the normalized current distance to the target, and Δ distance is the difference between the distance last frame, and the current distance.

The system has been trained for 2000 episodes, and the performance was evaluated by running 100 tests. Benchmarks show that the agent reached the target 100% of the time, with an average distance of 2.7 units at the end of the episode. The drone has a very high acceleration so cannot reach the exact middle of the target, but it still achieves an impressive level of accuracy.

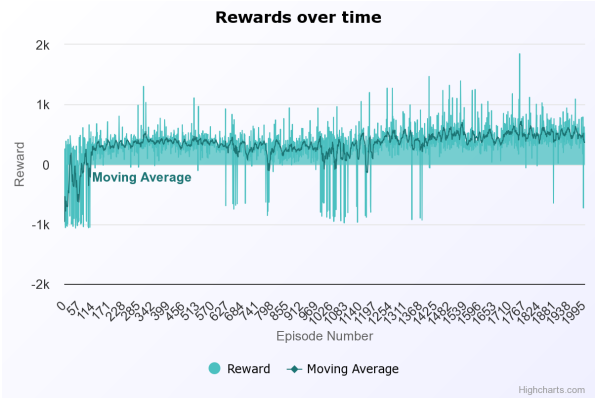

Figure 3.3: Drone rewards over time

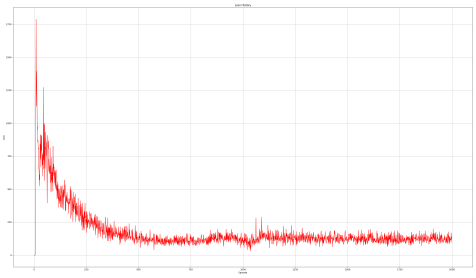

Figure 3.4: Drone loss over time

In the results, we can see that around episode 120, the agent quickly learns the reward function. The epsilon is small enough to let the agent maximize the reward in the environment. The loss has a downward trend, meaning the agent is getting better at predicting future rewards based on the current state and actions available. The loss stabilises at around ≈ 150, it may seem high but the higher the rewards, the higher the loss will be as it tries to predict the Q-values. If our action has a Q-value of 500, but estimated is 520, that is not that much off, but using MSE loss we will get a loss of $(500 - 520)^2 = 400$.

3.0.6 MARL Drone Environment

This environment builds on top of the drone environment, the physics remains the same. We now have two drones that are tasked with swapping positions. This is the hardest 2-dimensional scenario for collision avoidance, as both drones want to ideally travel in a straight line to minimise the delay, but can't because of the collision. They need to find the optimal solution and to aid them, the reward is now a combination of 5 functions. This environment uses the same hyperparameters as outlined in table 3.2.

$$R_i = \alpha \times c(r_{cg}) + \beta \times c(r_{cf}) + \Gamma \times s(x) + b(x) + t(x) \tag{3.2}$$

$c(x)$ returns $-n$ where n is the number of agents in the radius x, excluding self. $s(x) = -1 + d_{t-1} - d_t$ is the "schedule" reward, where d_t is the distance to target at time t. $b(x) = -20 \times e^{-0.1 \times d_e}$ is the boundary penalty, where d_e is the distance to the closest edge.

$$t(x) = \begin{cases} 10 & d_t < 10 \\ 0 & otherwise \end{cases} \tag{3.3}$$

α	β	Γ	r_{cg}	r_{cf}
1.00	1.00	0.01	75	25

Table 3.3: Parameters used in the reward function, where "cg" and "cf" are congestion and conflict variables respectively.

The observations (11) are ⟨position(x,y), target position(x,y), velocity(x,y), distance to target, distance to closest agent, bearing to closest agent, velocity of closest agent(x,y)⟩. The actions allow the agent to accelerate in four directions, or do nothing.

Goal reached	78%
Crashed with agent	19%
Left bounds	3%

Table 3.4: Performance of the model, tested on 100 random swap-position scenarios.

The results can be compared to another study performed at the University of Barcelona. They have used a convolutional neural network (CNN) and graph convolution reinforcement learning together with DDQN to model multi-UAV conflict resolution [Isufaj et al., 2022]. While the methodology of their approach is different to the one performed in this study, it is still valuable to compare the results to understand the strengths and limits of each method. Out of 10,000 episodes, their agent was able to resolve conflicts 56.5% of the time. It is important to note that their agents were working with a CNN, which may have resulted in imperfect information. A CNN system would more accurately represent the real-world scenario where perfect information might not be available, and the drones would need to rely on the camera frames as input. This study aims to test the theoretical best performance achievable with perfect information and the agents acting on their own.

The goal counts as "reached" when a drone is within 20 units of the target, and a crash occurs when the distance between agents is less than 10 units. The results demonstrate that the agent is capable of avoiding collisions and reaching the target in the majority of hard 2-dimensional scenarios. Further testing and refinement may be necessary to improve the performance of the model. Results suggest that the approach could be a viable solution for managing air traffic

control in real-time and real-world scenarios. We can expect the number of collisions to go down when we move into 3 dimensions as we will have more options for avoiding a collision.

3.0.7 MARL Planes Environment

The plane environment is represented in 3-dimensional space, with X going to the side, Y up and Z forward. The environment size is 1000×1000, each plane is spawned with a margin of 100, and making sure targets are instantiated at least 400 units away (this is calculated only in X and Z axis, this gives extra freedom in choosing arbitrary Y). The max environment time is set to 200, and the starting speed for planes is 2 units / second. This is done to make the training faster and to help the agents focus on avoiding collisions and then landing, rather than the flight itself. An episode terminates whenever an agent leaves the bounds, 200 frames have passed, and any agent crashes with the ground or another agent. The collision is modelled using a cylinder of height 10 and radius 32. When the plane reaches its target, it gets "frozen" into place, which simulates the plane leaving its sector (the target reward is still applied per environment tick).

The training scenario has the planes attempting to swap positions, similar to the drone environment. If we train using the hardest scenario possible, and the agent succeeds, that means much simpler scenarios can also be solved.

α	γ	Batches	Memory	τ	Network Size	Epsilon
5×10^{-4}	0.99	64	500,000	0.005	$(obs_n, 128, 128, action_n)$	$0.1 + 0.9 \times e^{-0.01t}$

Table 3.5: Hyperparameters used for the MARL plane environment

The hyperparameters are outlined in table 3.5. The replay size has been increased due to the simulation becoming more complicated. This dictates how far back we want an agent to look, if we have a small value only recent experiences are considered and we may encounter "catastrophic forgetting". Too large and we consider experiences which happened a long time ago that may be irrelevant at this time. This can be fixed by the prioritised replay buffer by considering the valuable experiences more.

Similarly, there is a balance with the τ value. Too large and we update the network too frequently making it hard for agents to learn. Too small and training takes much longer.

Each plane agent has a yaw and pitch rotation. Yaw is the rotation around the Y-axis, and pitch makes the plane rotate up or down. Both of the rotations are done with respect to the global axis, meaning changing yaw is not affected by the pitch. The plane forward is calculated by:

$$\langle cos(yaw) \times cos(pitch), sin(pitch), sin(yaw) \times cos(pitch) \rangle \tag{3.4}$$

Similarly, the right is calculated using

$$\langle cos(yaw + 90), 0, sin(yaw + 90) \rangle \tag{3.5}$$

the up is calculated using $up = forward \times right$ where $\times$ is the cross product between the two vectors. The "90" degree offsets are due to numpy arctan2 being 0 degrees at $\langle 1, 0 \rangle$, and the simulation treating 0 degrees as $\langle 0, 1, 0 \rangle$.

The plane is able to accelerate/decelerate, and will always move with respect to the forward. When on the ground ($y \leq 0$), the plane's max speed is set to 10% of the flight speed, and a 10% friction penalty is applied per frame. A crash with the ground occurs when the plane's vertical velocity is ≤ -2 or the normalized forward vector mostly points towards the ground ($forward \cdot \langle 0, -1, 0 \rangle \geq 0.5$). The simulation also includes gravity which encourages the planes to go fast. It equals $\min(0, speed - 2)$. The observation state (28) contains:

Basic Information	Target Information	Closest Agent (CA)	Angles
position	position	position	$\phi(A_{\text{fwd}}, \text{target}_{\text{dir}}, A_{\text{up}})$
velocity	$\text{target}_{\text{dir}}$	velocity	$\phi(A_{\text{fwd}}, \text{target}_{\text{dir}}, A_{\text{right}})$
forward	dist(A, target)	forward	$A_{\text{fwd}} \cdot \text{target}_{\text{dir}}$
		yaw	
		pitch	
		relative position	
		dist(A, CA)	

Table 3.6: Observation space for each agent

$$\phi(a, b, v) = \left(arccos\left(\frac{a \cdot b}{|a||b|}\right) \times sgn((a \times b) \cdot v \right) \tag{3.6}$$

Where

$$sgn(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \\ 0 & otherwise \end{cases} \tag{3.7}$$

All observations in the table 3.6 are in three dimensions unless it's an angle or distance. A is the agent, $target$ is their own target, $distance(x, y)$ is the Pythagorean distance between x and y (in three dimensions), $\phi(\vec{a}, \vec{b}, axis)$ is the signed angle between direction vectors $\vec{a}$ and $\vec{b}$ with respect to an axis (shown in equation 3.6.), CA refers to the closest agent to self. All subscripts are the variables of the corresponding objects, for example, A_{fwd} means forward facing vector of agent. $target_{dir}$ is the direction to the target, i.e. $target_{pos} - A_{pos}$. All of the above vectors were normalized to have a length of 1, i.e. $|\vec{a}| = 1$, and everything else is mapped into the $[-1, 1]$ range. The $\cdot$ symbol represents the scalar product between two vectors.

$$\alpha \times c(r_{cg}) + \beta \times c(r_{cf}) + \Gamma \times s(x) + b(x) + t(x) \tag{3.8}$$

The reward has been built on top of the drone environment reward, but it includes some variables. $c(x)$ is the number of agents in the radius, excluding self. r_{cg} is the congestion radius, and is set to 50, r_{cf} is the conflict radius and is set to 25. $s(x)$ is the schedule function which

guides the agent towards the target. It is identical to the one used in the drone environment. $b(x)$ is the boundary penalty, defined as $-100 \times e^{-0.2*d_e}$, where d_e is the distance to the closest edge (excluding Y). $t(x)$ is the target reward and encourages the agent to stay on it. When distance is less than 100, it returns $10 \times (1 - \frac{distance}{100})^2$, otherwise 0. $\alpha = 1$, $\beta = 20$ and $\Gamma = 1$ were used for this environment.

Planes have an action space (7) which consists of ⟨ do nothing, accelerate by 0.25, decelerate by 0.25, yaw + 4, yaw - 4, pitch + 4, pitch - 4 ⟩. The values have been chosen arbitrarily, yaw and pitch controls are quite high as the planes can fly extremely quickly in the simulation (usually reaching the target in about ≈ 50 ticks).

Transfer training has been used here, a single-plane environment was used that just forced it to land in the target. Once the plane was comfortably reaching the target, the second plane was added together with additional rewards and collision logic. This has greatly improved the training efficiency. When MARL is trained from scratch, it takes around 5000 episodes to get to the near-optimal policy. With transfer training, it took around 1700 episodes to reach a near-optimal policy. The results have been shown in figure 3.5.

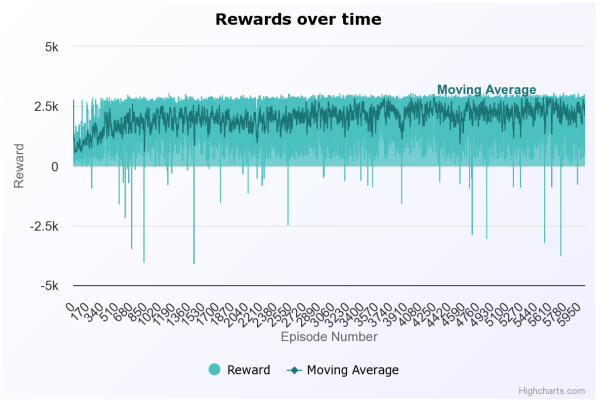

Figure 3.5: Moving average (window=10) of MARL plane transfer learning, 6019 episodes (9,019 total)

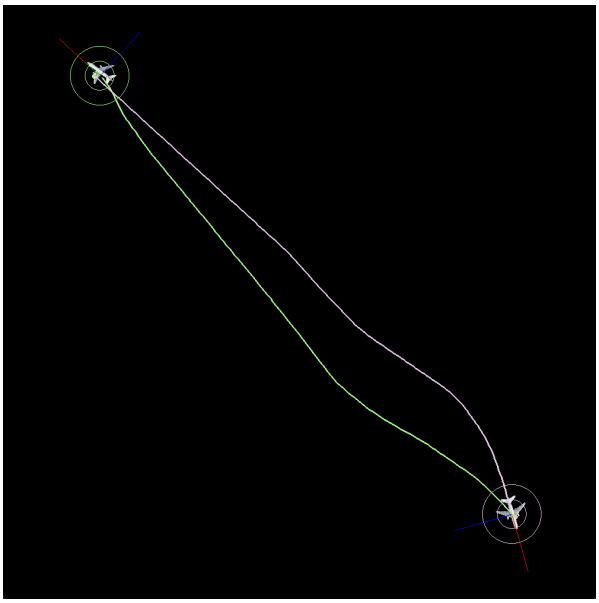

Figure 3.6: A successfully resolved conflict by the agents

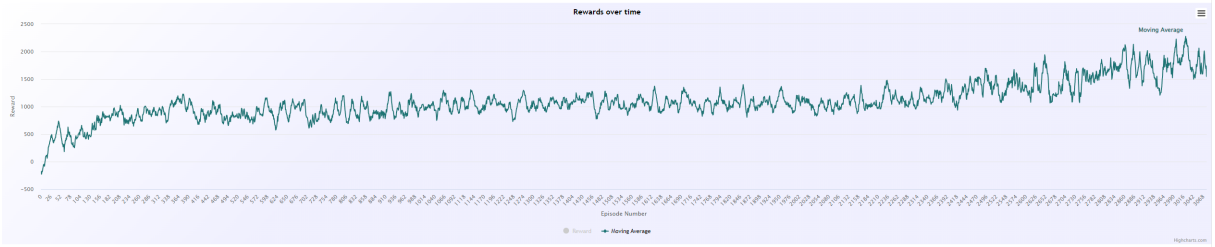

Figure 3.7: Moving average (window=10) of the rewards for the plane, 3000 episodes.

From the results in figure 3.7, we can see that the agents quickly learn a "decent" policy and then get stuck for about 2000 episodes. After an additional 1000 episodes, the agents have managed to improve their solution. Figure 3.5 shows the agent rewards when training in the conflict resolution scenario using transfer learning. Figure 3.6 shows a path taken by the agents, which resulted in a successful conflict resolution and both agents have landed in their corresponding targets. The red line indicates the plane's forward, blue is the local right, and green is the local up. This is used to make 3-dimensional visualisation a bit easier, as the simulation is run in 3 dimensions, but drawn in 2 dimensions. The circles indicate the congestion and conflict radius, where congestion is the bigger circle.

Goal reached	92%	184 / 200
Crashed with agent	3%	3 / 100
Left bounds	4%	8 / 200
Crashed with ground	6%	12 / 200
Flight Path Efficiency	101.65%	184 / 200
Smallest separation distance	39.7 units	184 / 200

Table 3.7: Performance of the model, tested on 100 random swap-position scenarios.

It is important to note that the results here showcase the worst-case scenario. Two planes swapping a position, in a distributed system is a very difficult task.

We can see that in the benchmark the percentages do not add up to 100%. This is because if a plane has not reached the target, but also did not leave the sector or did not crash, it would count as 0 towards all statistics.

The system was also tested against the speed of inference, the time it takes to compute an action in 10,000 different scenarios was averaged, and the result was $\approx 41.22ms$ computed on the 3060Ti Nvidia GPU, without using tensor cores. This is important because if this was used in a real-life scenario, the system needs to be able to react to environmental changes quickly, due to the speed of aeroplanes. This could potentially be reduced if we decreased the number of parameters in the system.

One of the key metrics used in this study is the flight path efficiency. It simply was calculated using a percentage based on theoretical optimum distance $\frac{\text{actual distance flown}}{\text{optimal distance}}$. The closer to 1 we are, the more perfect the agent's flight path is. This metric was only calculated when the plane has successfully avoided a collision and "landed" at the corresponding target, it was also calculated for both planes and in some scenarios, one plane could land but not the other. This metric is desirable in air traffic control systems as it can reduce flight times, fuel consumption, carbon emissions and possibly ticket prices. In this experiment, we have reached a value of 1.0165 which means on average agent flew about 1.65% more than needed (on average). During testing it was found that one of the planes would prefer to dodge very minimally, mostly going in a near-straight line, and the other plane was focused on dodging it, meaning one plane would have an efficiency of near 100%, and the other of around 103%.

Other key metrics we want to focus on are safety and reliability. In the benchmark we

have observed that about 3% of the simulations have resulted in aircraft crashing, this shows promising results for future research, but also that there is room for improvement. Additionally about 4% of the time the agent has left the bounds. This was caused by the plane narrowly missing the target, and then lacking the controls to recover, the environment was very strict and required very precise accuracy when hitting the target.

The recommended distance for planes is 3 miles laterally or 1,000 (304.80 meters) feet vertically. This project has simulated this separation distance using $r_{cg} = 50$ variable. The smallest separation distance is the all-time minimum distance between planes during an episode. We can see that it reaches about 39.7 units on average. With the closest reaching 21.1 and further reaching 86.4. This suggests that the congestion penalty wasn't high enough for the agent to maintain that congestion distance at all times. The congestion importance value (α) was set to 1, which in comparison to other values wasn't big enough for the agent to consider that penalty as strictly. The conflict value r_{cf} was set to 25, and we can see that the agent really avoids going closer to the plane since the penalty for r_{cf} was $\beta = 20$.

Overall, while there is an improvement to be made in specific areas, the study shows promising results, particularly in terms of flight path efficiency and crash rate. To comply with the Federal Aviation Administration (FAA) rules, planes should be able to maintain an arbitrary distance (set in the simulation). This would require reward balancing to make the agents strictly abide by the rules. The reward function is made to be flexible, simply changing the coefficients could improve the behaviour.

4

Discussion and Summary

Experiments in this study showcase the feasibility of using reinforcement learning in a complicated stochastic environment like air traffic control. They were designed to test a two-agent swap position scenario, which is a very hard scenario to resolve a collision in. This section summarises the results and points out areas to improve and future work that could be done.

In this study, we proposed a novel approach to air traffic control using a multi-agent distributed Deep Double Q-Learning algorithm and a custom-built physics engine for aircraft. Our approach is aimed at addressing collision detection, fuel efficiency and safety. The paper showcases how experiments have evolved, starting from a simple drone simulation that has been used as a base, and then build into an air traffic simulator.

The evaluated performance of our approach has used a swap position scenario, which is the hardest air traffic problem in a distributed system. The results have shown that our approach

was able to safely fly the agents to their corresponding location, with a 3% crash rate. We have also achieved a flight path efficiency of 101.65%, indicating that the aircraft was able to resolve collisions efficiently when reaching their destinations. The aircraft movement was limited just to speed control and discrete pitch/yaw change in 3 dimensions, resulting in an action space of 7.

Our approach has used a stochastic environment to generalise the training process, making agents smarter in unseen situations. The proposed system takes around 41.22ms for inference.

4.0.1 Ethical Concerns

The use of DDQN for air traffic control raises several ethical concerns, such as "Who is responsible if a human-AI system makes a mistake?", "How can a system be built that is trusted by humans?" and "How can the need for both safety and efficiency be balanced?" [The Alan Turing Institute, n.d.]. One way to address these concerns is to design experiments that involve real controllers to understand how the AI makes its decisions. Another way is to create a machine-learning system that simply aids in decision-making, but requires human intervention. This could lower the entry requirements for air traffic controllers and allow a bigger pool of candidates to be considered for the job. For a fully automated system, the tests could be done in a controlled environment where no person can be harmed. All of the research conducted in this dissertation was entirely simulated and did not involve any real-world air traffic control scenarios. Hence any potential ethical concerns associated with this project were not an issue in this research.

4.0.2 Further research & improvements

An important improvement would be to use a system that does not have perfect information. This study used perfect information, meaning the plane knew exactly where each plane was and where it was headed. In the real world, radars could malfunction or have a delay, which would affect the agent's ability to take a non-risk action. An agent would need to get much better at predicting observations without taking risks. This could be handled by another neural network or some algorithm that works together with reinforcement learning to create more predictable observations when information is uncertain or missing.

Further research could be conducted that would test distributed vs centralised systems, and whether the performance would improve. Using a "hybrid" version, where a centralised network is trained for 1, 2 and 3 agents. Then the system would resolve conflicts in each plane group. This would utilise the scalability of distributed method and the potential higher accuracy of a centralised system. All 3 methods could be analysed and compared to see which combination is best.

Previous versions of the plane environment made attempts on making planes land, but agents constantly crashed due to the need of going fast. Lowering the initial position of planes did not help. In one of the experiments the collision with the ground was completely turned off, and the plane was given a reward based on the dot product of itself and the highway strip direction. However, the agent still refused to align itself with the highway. This is something

that could be looked into in further research.

Another area of research that could be conducted is if we can reduce the number of dimensions, which in turn would allow us to have many more variables and increase training times. In machine learning the bigger the dimensions of the network, the more computational power is needed to train the network (grows exponentially). This is known as the "curse of dimensionality" [Schumann and Aragón, 2021]. When discretizing space, we want to give a lot of freedom to the agent. For this example, let's assume we have 3 different actions ⟨ change pitch, change yaw, change speed ⟩. Where pitch and yaw refer to the 3-dimensional rotation. We want our agent to have the ability to change those at different rates. Typical solutions would duplicate actions with different rates of change ⟨ -3 pitch, -2 pitch, ..., +2 yaw, +3 yaw, ... +3 speed ⟩. For a actions, and n sensitivity dials, this gives us space complexity of $a \times n$. This paper tests an alternative method of letting the agent select the sensitivity and then choosing an action, reducing our space complexity to $a + n$ ⟨ -3, -2, ..., +2, +3, ..., pitch, yaw, speed ⟩. The downside is that the agent would be behind 1 simulation tick as it has to keep changing the sensitivity.

Another way to make the results more accurate would be to simulate physics more accurately, this could include using some physics engine or using NASA's SimuPy Flight Vehicle Toolkit [Ames, 2020]. This would take much longer to train because of more complex physics calculations but would provide much more accurate results in the real world. A simplified version of this would have the agent control each of the 3 axis rotations locally (pitch, yaw, roll), this would make it much harder to solve the problem, but also more realistic to the real world.

An additional improvement that could be made is using a real-life metric like 3Di [NATS] or similar. It has been designed to *... deliver 600,000 tonnes of CO2 savings over the next 3 years...* With most flights having a score of around 15 and 35 in 2012 [Airport Watch, 2012]. This would help with a direct comparison of the method, and if implemented accurately, would better represent how efficient the system is.

Imitation learning could also be a way to quickly accelerate learning in the real world, where a mistake could be very costly. It involves having an experienced human playing in the environment, the agent will learn how to mimic expert behaviour. This does not require a reward function which could make it more accessible to the real world. We could also experiment with using data from real-life air traffic control situations. This approach could also be used as a base for the agent, transfer learning could then be used for a harder scenario.

In conclusion, this research has demonstrated the potential of using multi-agent reinforcement learning using a distributed system to improve air traffic control using DDQN. The results show that the proposed solution is able to learn and make optimal decisions in complex, dynamic and stochastic environments. This has the potential to significantly improve the safety and efficiency of air traffic control, ultimately benefiting the airlines, passengers and the environment. Further research is required to fully explore the capabilities of this approach and to develop practical implementations for real-world air traffic control systems.

REFERENCES

Airbus. Global market forecast. https://www.airbus.com/en/products-services/commercial-aircraft/market/global-market-forecast, 2022. Accessed: 27/04/2023.

Airport Watch. NATS has data on its first 6 months of new flight efficiency metric, 3Di. https://www.airportwatch.org.uk/2012/08/heathrow-third-runway-blues/, 2012. Accessed: 02/05/2023.

N. Ames. Simupy-flight. https://github.com/nasa/SimuPy-Flight, 2020. Accessed: 01/05/2023.

M. Banoula. What Is Q-Learning: The Best Guide To Understand Q-Learning. 2023. URL https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning. Accessed: 21/04/2023.

S. Basodi, C. Ji, H. Zhang, and Y. Pan. Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3):196–207, 2020. doi: 10.26599/BDMA.2020.9020004. Accessed: 26/04/2023.

BEA Aero. https://bea.aero/en/investigation-reports/notified-events/detail/serious-incident-to-the-boeing-787-registered-n16009-operated-by-united-and-to-the-airbus-a320-registered-oe-ijf-operated-by-easyjet-on-20-07-2020-at-paris-charles-de-gaulle-ad/, 2020. Accessed: 01/05/2023.

M. Brittain and P. Wei. Autonomous air traffic controller: A deep multi-agent reinforcement learning approach. *arXiv preprint arXiv:1905.01303*, 2019. Accessed: 14/04/2023.

H. Dave. Understanding the Bellman Optimality Equation in Reinforcement Learning. 2021. URL https://www.analyticsvidhya.com/blog/2021/02/understanding-the-bellman-optimality-equation-in-reinforcement-learning/. Accessed: 21/04/2023.

Everyday Aviation. Air traffic controller age requirements. https://everydayaviation.com/air-traffic-controller-age-requirements/. Accessed: 22/03/2023.

S. Ghosh, S. Laguna, S. H. Lim, L. Wynter, and H. Poonawala. A deep ensemble method for multi-agent reinforcement learning: A case study on air traffic control. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):468–476, May 2021. doi: 10.1609/icaps.v31i1.15993. URL https://ojs.aaai.org/index.php/ICAPS/article/view/15993. Accessed: 14/04/2023.

R. Isufaj, M. Omeri, and M. A. Piera. Multi-UAV Conflict Resolution with Graph Convolutional Reinforcement Learning. *Applied Sciences*, 12(2):610, 2022. Accessed: 20/04/2023.

C. Koch. How the computer beat the go master. https://www.scientificamerican.com/article/how-the-computer-beat-the-go-master/, 2016. Accessed: 29/04/2023.

M. Lu, Z. Shahn, D. Sow, F. Doshi-Velez, and L.-W. H. Lehman. Is Deep Reinforcement Learning Ready for Practical Applications in Healthcare? A Sensitivity Analysis of Duel-DDQN for Hemodynamic Management in Sepsis Patients.

A. Mukherjee, Y. Guleria, and S. Alam. Deep Reinforcement Learning for Air Traffic Conflict Resolution Under Traffic Uncertainties. *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*, pages 1–8, 2022. doi: 10.1109/DASC55683.2022.9925772.

NASA. Faa releases nasa study of air traffic controller fatigue. `https://aasm.org/faa-releases-nasa-study-of-air-traffic-controller-fatigue`, 2011. Accessed: 27/04/2023.

NATS. Airspace efficiency. `https://www.nats.aero/environment/airspace-efficiency/`. Accessed: 22/03/2023.

Y. Ouyang, X. Wang, R. Hu, and H. Xu. APER-DDQN: UAV Precise Airdrop Method Based on Deep Reinforcement Learning. *IEEE Access*, 10:50878–50891, 2022. doi: 10.1109/ACCESS.2022.3174105.

S. Raga. 8 secrets of air traffic controllers. `https://www.mentalfloss.com/article/502032/8-secrets-air-traffic-controllers`, 2019. Accessed: 27/04/2023.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2016. Accessed: 21/04/2023.

J. F. Schumann and A. M. Aragón. A machine learning approach for fighting the curse of dimensionality in global optimization. *arXiv preprint arXiv:2110.14985*, 2021. URL `https://arxiv.org/abs/2110.14985`. Accessed: 14/04/2023.

D. Silver. Rl course by david silver - lecture 2: Markov decision process. 2015. URL `https://www.youtube.com/watch?v=lfHX2hHRMVQ`. Visited on 21 Oct 2022.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: https://doi.org/10.1038/nature16961. Accessed: 18/04/2023.

A. Stooke and P. Abbeel. Accelerated Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1803.02811*, 2019.

R. S. Sutton and A. G. Barto. Reinforcement Learning, An Introduction, Second Edition. In *Reinforcement Learning, An Introduction, Second Edition*, pages 1–71. Malaysia; Pearson Education Limited, 2018. Accessed: 20/04/2023.

The Alan Turing Institute. Project bluebird: An ai system for air traffic control. `https://www.turing.ac.uk/research/research-projects/project-bluebird-ai-system-air-traffic-control`, n.d. Accessed: 02/05/2023.

TypeIA and rbp. How much is the minimum safe distance between two planes in flight? https://aviation.stackexchange.com/questions/2806/how-much-is-the-minimum-safe-distance-between-two-planes-in-flight. Accessed: 04/05/2023.

H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv preprint arXiv:1509.06461*, 2015. Accessed: 02/05/2023.

C. Wang, H. Yan, and F. Li. Self-selection Protocol Algorithm for Wireless Networks based on DDQN. *Journal of Physics: Conference Series*, 1871(1):012134, 2021. doi: 10.1088/1742-6596/1871/1/012134. Accessed: 16/04/2023.

T. Zhang, X. Wang, B. Liang, and B. Yuan. Catastrophic Interference in Reinforcement Learning: A Solution Based on Context Division and Knowledge Distillation. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2022. doi: 10.1109/tnnls.2022.3162241. URL https://doi.org/10.1109%2Ftnnls.2022.3162241. Accessed: 25/04/2023.

Acknowledgments