university of groningen / faculty of science and engineering

Bachelor's thesis

Evaluating the potential of Deep-Q Networks for weather avoidance in aviation

Abstract

Adverse weather conditions pose considerable challenges to aviation, affecting safety, efficiency, and passenger comfort. The current approach to weather avoidance in aviation is dependent on pilots' interpretation of weather forecasts from various channels, potentially causing inconsistencies in the ways the weather is avoided. Deep Reinforcement Learning offers a promising alternative to traditional automation methods, enabling agents to learn optimal actions through repeated interactions with simulated environments. The project adapts the Deep Q-Network algorithm, building on agent architecture originally designed for aviation conflict resolution, to navigate simulated aircraft around inclement weather. During evaluation, the trained agents managed to avoid all adverse weather in the majority of simulation runs. Despite challenges with training stability and occasional unrealistic flight paths, the results demonstrate the potential of Deep Reinforcement Learning—specifically the Deep Q-Network algorithm—in automating aircraft navigation around inclement weather. Ultimately, the project provides a foundation for integrating the presented approach with other domains of air traffic control, such as conflict resolution, weather forecasting, and more.

Author:
Adam Posker

Supervised by:
Dr. Harmen A. de Weerd
Dr. Michael H. F. Wilkinson

July 3, 2025

Acknowledgments

I would like to dedicate the thesis to my dear friends—Giulia, Lorenzo, Mechy, and many others—who have supported me throughout the study, both emotionally and intellectually, and made the entire experience truly unique.

I am thankful to my parents for encouraging me to explore the possibility of studying abroad, and supporting me in pursuing it. I would also like to express deep gratitude toward my grandparents—my grandmother, for always showing me nothing but unconditional love and care; and my grandfather, who taught me that there exists no problem in this world that could not be solved with a 50-kilometer hike, a screwdriver, or a drop of superglue. Real radios do glow in the dark.

A word of thanks also goes to my supervisors, especially Dr. De Weerd, who has been very understanding of my personal peculiarities—our meetings have always helped me stay motivated throughout the project.

Contents

Abbreviations

AI	Artificial Intelligence
CA	Cellular Automata
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
ICAO	International Civil Aviation Organization
MDP	Markov Decision Process
ML	Machine Learning
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
TD	Temporal-difference target

1 Introduction

Navigating around adverse weather conditions is essential for aircraft, as it directly affects safety, operational efficiency, and passenger comfort. Currently, the industry's approach to weather avoidance relies on pilots' interpretation of data presented by the onboard weather radar, as well as information from air traffic control (ICAO, 2016). However, the human factor introduces variability into the decision-making, potentially leading to inconsistent weather avoidance practices.

In recent years, Deep Reinforcement Learning (DRL)—first formalized by Mnih et al. (2015)—has emerged as a promising approach to sequential decision-making problems, such as conflict resolution in air traffic control, offering distinct advantages over traditional methods. Unlike rule-based systems, DRL-based models learn optimal actions through repeated interactions with simulated environments, enabling them to independently adapt to dynamic and unpredictable inputs while accounting for extensive sets of parameters. This aligns closely with the demands of weather avoidance in aviation, where conditions can change rapidly and are influenced by a wide range of variables. Moreover, DRL models can be designed to balance specific objectives based on desired priorities, such as accepting a moderate increase in fuel consumption as a trade-off for safely avoiding severe weather.

Despite the potential of DRL, research into its application in aviation has so far been limited to conflict resolution, without exploring its use for weather avoidance (Wang et al., 2022). This project therefore seeks to make a preliminary investigation into the feasibility of using DRL in this domain. Specifically, the study initially compares two widely-used DRL algorithms—Deep Q-Network (DQN) (Mnih et al., 2015) and Proximal Policy Optimization (PPO) (Schulman et al., 2017)—to then assess the efficacy of the selected DQN algorithm in navigating simulated aircraft around adverse weather. Aside from evaluating the feasibility of DRL for weather avoidance in aviation, the project provides insights into the effectiveness and limitations of the chosen algorithm, serving as a basis for more in-depth research in the future. The suitability of the BlueSky air traffic simulator by Hoekstra and Ellerbroek (2016) is also briefly discussed, evaluating its usefulness for weather-related studies in aviation.

To allow for comparability with related research, the project adapts the agent architecture designed by Van Gelder (2023), who used an identical algorithm-simulator combination (DQN and BlueSky) for handling conflict resolution in air traffic control. While the overall structure remained unchanged, adjustments were made to relevant parts of the architecture to accommodate the different domain.

After introducing the related research in Section 1.1, the paper explains the methodology in Section 2, including the aforementioned DRL algorithm comparison. The results are discussed in Section 3, covering training stability and agent performance. This is followed by Section 4, which provides interpretation of the results, and explores implications for the industry, as well as areas for future improvement. Finally, the paper concludes with a summary of key findings and takeaways in Section 5.

1.1 State of the art

Since the introduction of the Traffic Alert and Collision Avoidance System (TCAS) in the early 1990s, the increasing volume of air traffic has pushed the industry to investigate tools aiding human decisions in handling this traffic, and improving its flow efficiency (Kuchar & Yang, 2000). Research aimed at refining conventional approaches to air traffic conflict resolution has been ongoing for several decades, with comprehensive reviews conducted by the aforementioned Kuchar and Yang (2000), and more recently by Ribeiro et al. (2020). Naturally, these approaches only become more complex with time, in order to account for all possible scenarios that could be encountered. As with any computer system, the growing complexity could result in a more complicated long-term maintenance, while also increasing the risk of failure when edge cases are encountered.

As stated by Wang et al. (2022), Machine Learning (ML)—specifically Deep Reinforcement Learning—offers a promising alternative for improving these approaches, being particularly suitable for solving sequential decision-making problems, such as conflict resolution in air traffic control. The authors further note that, unlike traditional methods, DRL can process and learn from extensive sets of parameters through trial and error, without requiring complete prior knowledge of its environment. This allows it to autonomously determine appropriate actions within specific contexts, maximizing its performance in executing a certain task. It should be noted that this learning process carries a conceptual resemblance to the training of air traffic controllers, where both involve learning from interactions with (simulated) environment, with the overall goal of achieving optimal performance.

In the last decade, there has been an increasing amount of research focused on DRL application for conflict resolution in aviation. Among the various DRL algorithms, Deep Q-Network (Mnih et al., 2015) and Proximal Policy Optimization (Schulman et al., 2017) appear to be frequently used for these applications, although other algorithms have also been explored (Wang et al., 2022). The works of Brittain and Wei (2019) and Van Gelder (2023) provide examples of the use of PPO and DQN, respectively, in the domain of conflict resolution.

While there has been a substantial amount of research on the possibilities of using DRL in aviation, none of the proposed systems have yet been put into service. This is explainable by the safety-critical nature of the industry with human lives at stake, historically relying on human-in-the-loop to assure safety (Degas et al., 2022). The trend is also reflected in the comparative reviews cited previously, where the works of Kuchar and Yang (2000) and Ribeiro et al. (2020) focus solely on the traditional approaches, while the reviews by Wang et al. (2022) and Degas et al. (2022) cover the use of DRL and other Artificial Intelligence (AI) methods, respectively, providing very little direct comparison between the two.

Furthermore, despite the extensive research on the use of DRL for conflict resolution, the possibility of its application for avoiding weather remains largely unexplored, as pointed out by Wang et al. (2022). The research covering weather avoidance is limited to non-ML approaches, with the work of Love et al. (2009) serving as a representative example.

Of these two gaps in the literature, addressing the absence of real-world deployment of the systems is beyond the scope of a single project, as it requires sustained research efforts, repeatedly validating the suitability and safety of the proposed approaches. However, the second gap in the literature—concerning the application of DRL for weather avoidance in aviation—represents a much more reasonable research opportunity, which this project seeks to address.

2 Methods

This section explains the specifics of the approaches employed to obtain the results. A preliminary survey of existing literature has been done to determine the most appropriate DRL algorithm to be used, as implementation and performance comparison of multiple algorithms would not be feasible within the scope of the project.

With the aforementioned discussed, the explanation moves on to the implementation of the simulation environment, followed by the implementation of the agent itself. The environment is then formally defined as a Markov Decision Process (MDP), adapting its components from the original design by Van Gelder (2023) to the new domain. Finally, the training process is described, concluding the explanation of the methodology used to produce the project's results.

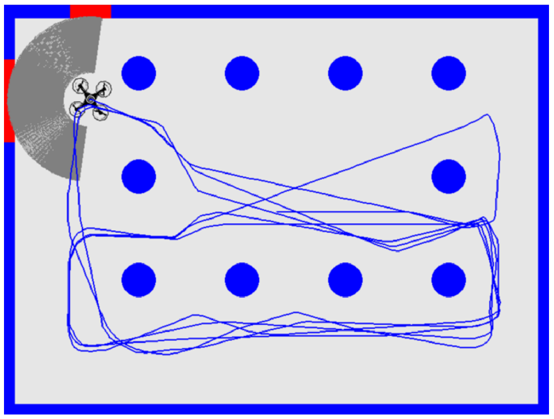
2.1 Tool selection

As mentioned in Section 1, to keep the study comparable with related research, the tools and agent architecture will generally be adapted from the work of Van Gelder (2023), who applied them to conflict resolution in air traffic control. The following subsections briefly investigate any alternatives that might promise significantly better results, otherwise the original choices will be kept.

2.1.1 Training algorithm

The selection of appropriate DRL algorithm is important for the study to provide a reasonably representative evaluation of DRL as a whole. In his paper, Van Gelder (2023) originally chose DQN for this purpose. Based on the comprehensive review conducted by Wang et al. (2022), DQN and PPO appear to be the most frequently used algorithms for conflict resolution in aviation, and were therefore selected for further comparison.

A study by Rybchak and Kopylets (2024) directly compares the ability of DQN and PPO to avoid obstacles in simulated 2D drone navigation, concluding that PPO generally outperformed DQN in maximizing the reward received in the simulations. However, it also found that the agents using DQN were *'more agile, easily weaving through tight spaces and obstacles in enclosed areas'* while the agents using PPO *'prefer safer, open areas, avoiding the riskier task of navigating through narrow spaces between obstacles'*. Given these findings, the authors recommend a scenario-specific algorithm selection, depending on the task at hand. Figure 1 shows the difference in navigational strategy of the drone-controlling agents utilizing DQN and PPO, respectively.


(a) Agent using DQN

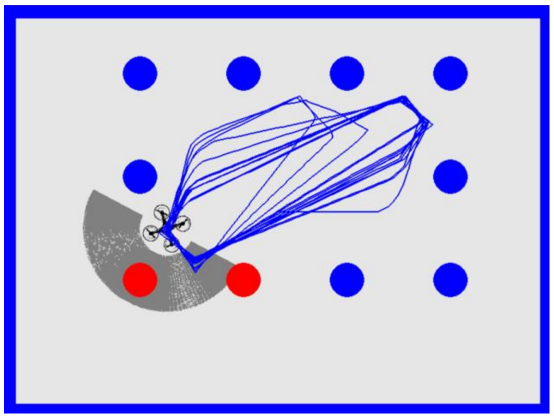

(b) Agent using PPO

Figure 1: Navigational strategies of drone-controlling agents using different DRL algorithms. Adapted from Rybchak and Kopylets (2024), licensed under CC BY 4.0.

This difference in navigational strategies is a major factor in the choice of the algorithm. While some of the described behavior could likely be tweaked by adjusting the reward function, the presented evidence generally points to the agile nature of DQN encouraging navigating through tighter passages, rather than avoiding the obstacle entirely like PPO. Naturally, this behavior of the DQN is desirable for an agent designed to navigate around adverse weather—the option of maneuvering between thunderstorms is often preferable to evading the stormy area completely, as doing so could result in the aircraft being unable to reach its destination altogether.

Further supporting the suitability of DQN, Kalidas et al. (2023) investigated the ability of DQN, PPO, and a third algorithm, Soft Actor-Critic, to navigate a drone through dynamically changing 3D environments. In such a scenario, *'PPO performed poorly among the algorithms, suggesting that on-policy algorithms are not suitable for effective collision avoidance in large 3D environments with dynamic actors'*. The better sample efficiency enabled by the replay buffer of the DQN is stated as the main reason for its advantage over PPO. The authors further note that *'in small paths and turns, DQN might not be as advantageous due to its restricted discrete action space'*. However, this downside could be considered insignificant for our purpose, as the amount of space to navigate through relative to the size of the aircraft is much greater compared to drone navigation.

At this stage, it should be mentioned that the implementation covered in this paper focuses solely on simulation in 2D, as will be described in Section 2.2. However, any follow-up research should ultimately aim to extend this to 3D, to be able to demonstrate the capabilities of the technology on real-world traffic. Therefore, the ability to perform well in this future three-dimensional extension of the simulator environment is considered a benefit of the DQN, even though it may only manifest with further research.

With the aforementioned justifications, the DQN algorithm will be used for the project. The goal will be to take the agent architecture used by Van Gelder (2023) and adapt it for use in the domain of weather avoidance, providing a comparison of two similar agent designs applied to different domains within air traffic control. All modifications made to the original design—be it different values of parameters, or adjusted formal definition of the environment—will be pointed out and justified. These adjustments primarily involve the significant differences in the agent's environment, which will be defined in Section 2.4.

2.1.2 Simulator

Let us now direct our attention to the choice of the air traffic simulator. A logical candidate would be the *BlueSky: Open Air Traffic Simulator* by Hoekstra and Ellerbroek (2016), which was also used by Van Gelder (2023), while being the single most chosen platform among the papers reviewed by Wang et al. (2022).

For this study, it was important to verify that the simulator provides sufficient weather functionality, given that previous research on the use of DRL in air traffic control only dealt with conflict resolution among aircraft, not with aircraft avoiding weather. Having reviewed the documentation and relevant parts of the source code, it appears that the only supported weather-related functionalities are the wind and a simple turbulence model. Since these alone would not be sufficient for the needs of the project, other simulator options were briefly investigated.

Among the potential alternatives encountered, the *AirTrafficSim: An Open-Source Air Traffic Simulation Environment* by Hui et al. (2023) offers a more modern user interface, built-in weather data integration, and straightforward control via an API. However, due to its smaller scale, it seems to be generally less supported compared to BlueSky, which could complicate its use in this project, potentially limiting long-term utility.

After a further review and initial implementation attempts, it became clear that the better maturity of the BlueSky simulator is a critical advantage. Given its frequent use in research, there are multiple projects that can be used as additional resources on top of the documentation. Unlike the AirTrafficSim, BlueSky also allows the possibility of being used headless as a Python package, making it easy to integrate into the project—without the need to interact with a separate external program.

To compensate for the lack of support for simulating adverse weather, a custom weather implementation was developed separately from the simulator, and is discussed in Section 2.2.2 as part of the greater simulation environment. While this additional work was necessary to make the BlueSky simulator viable for the project, it was deemed a justified tradeoff to be able to benefit from the aforementioned advantages it provides.

2.2 Environment implementation

Having selected the tools to be used, we will now focus on the implementation of the simulation environment itself. Given that adverse weather is not supported by the simulator directly, the environment module serves as an abstraction layer compensating for this, encompassing the interaction with the simulator, as well as the weather implementation.

With the smaller scope of the project in mind, the following choices have been taken with respect to the capabilities of the environment. These points shall be referred to as *design choices* throughout the document.

1. **The simulation includes a single aircraft at a time.** This keeps the project focused strictly on weather avoidance. Navigating multiple aircraft simultaneously would require the domain to be shifted back toward conflict resolution to avoid collisions, which has been covered by previous research discussed in Section 1.1.
2. **The aircraft flies along a single fixed route.** This generally simplifies the training process and its implementation, while being straightforward to expand on in the future. Since this choice might cause the agent to learn some route-specific behaviors, the state space features will be taken relative to the attitude of the aircraft, making the learned capabilities of the agent more universal.

3. **The altitude and true airspeed of the aircraft are fixed.** With the navigation restricted to lateral maneuvers, the problem becomes strictly two-dimensional, thus reducing the size of the state and action spaces of the agent, and simplifying visualization of the simulation runs. Additionally, avoiding thunderstorms by changing these quantities would require them to be varied significantly, making such approach rather impractical.

4. **The agent is only used to navigate around adverse weather.** The flight path of the aircraft is generally controlled by the autopilot, which follows a predefined sequence of waypoints. When the state space of the agent indicates the presence of weather that needs to be avoided, the control is handed over to the agent, and given back to the autopilot when no more adverse weather is detected.

5. **The simulation includes wind.** Initially, it seemed reasonable to leave the wind out to simplify the environment and its implementation further. However, with no weather movement due to wind, the DRL algorithm would be acting as a simple pathfinder, which could be implemented in much more efficient way without the use of ML. Therefore, the wind is included in the environment, being uniform across the entire simulation area, and constant in its direction and speed for each simulation run.

Before the simulation starts, the two environment components—the simulator and the weather implementation—are initialized to their respective initial states, as will be described in Section 2.2.1 and 2.2.2. From there, the environment is updated in discrete steps, each corresponding to $dt = 5\,\text{s}$ of simulation time, chosen as a balance between performance and simulation accuracy. The core of the update process is described in Algorithm 1 (hereafter also referred to as the *core algorithm*). The `position` of the plane is a placeholder for its longitude and latitude, while the `wind` variable represents the speed and bearing components of the wind. The `simulator_features` and `weather_features` together make up all the information needed to describe the state s of the simulation for the agent, as explained in Section 2.4.

Algorithm 1 Process of incrementing the environment by a single step (also referred to as the *core algorithm*)

```
Input: Action $a$ that the aircraft in the simulator should take, environment $E$ itself
Output: Set of features describing the new state of the simulation $s$
  position, heading, wind, simulator_features $\leftarrow$ SIMULATOR_UPDATE($a$)
  weather_features $\leftarrow$ WEATHER_UPDATE(position, heading, wind, $E$.step)
  $E$.step $\leftarrow$ $E$.step + 1
  $s$ $\leftarrow$ COMBINE_FEATURES(simulator_features, weather_features)
```

However, this algorithm is only the internal implementation of the update process, which always executes a single simulation step. When other parts of the program interact with the environment, the simulation can proceed in two distinct ways (per design choice 4), which are discussed in the following paragraphs. Both of these options return the features of state s_{t+1} reached after executing the update, and a boolean indication of whether the current simulation run should terminate with the latest step due to the step limit being reached, or the plane leaving the map. The step limit was set to 400, corresponding to $t_{max} = 2000\,\text{s}$ of simulation time. This value was found to provide a good balance with respect to the length of the flight path, allowing the agent enough time to take a reasonable detour to avoid the adverse weather, while ensuring that the simulation terminates early in case the agent goes completely off course.

The first update option is the **simple step**. In this case, the action a that the core algorithm receives is simply an instruction to automatically follow the predefined route on autopilot (so called lateral navigation, or `LNAV`). This means either continuing or resuming the autopilot mode, depending on previously taken actions. Since the action is not produced by the agent, its outcome does not need to be evaluated.

The more complex update option is called **steps with action evaluation**. Here, the core algorithm is given an action a chosen by the agent. As such, the action has to be evaluated to be able give feedback back to the agent. To do this, the core algorithm is executed once with this action, so that it can be registered in the simulator. Next, several steps of the core algorithm with the `NONE` action are executed, to wait for the agent's instruction to take a full effect (e.g. the plane completing its turn). Section 2.4.2 defines the actions available to the agent, with some taking up to 18 seconds of simulation time to complete. Since this duration falls between multiples of the 5-second simulation step dt, cooldown periods of 3 and 4 steps were tested, with the 3-step cooldown selected for its better results. After this period, the reward is calculated based on the outcome of the action according to the reward function discussed in Section 2.4.3, and is returned together with the aforementioned information provided by both update methods.

On top of this, the environment also has the ability to record the position and heading of the plane at every step of the simulation, for the purposes of visualizing the run. This is only used for evaluation, as there is no need to record such information during training. Since the simulator is only used in headless mode and not as an external program with graphical user interface, the visualization is implemented separately using the matplotlib Python package, and the recorded simulation data.

Following design choice 2, a single route passing over the Netherlands was selected, and is defined in detail in Section 2.2.1. While not relevant for the simulation itself, the Dutch border is included in all visualizations to serve as a reference point. The external border of the country uses the data from GADM (2022), with the outlines of the most prominent lakes and polders obtained by hand-tracing over map data by OpenStreetMap contributors (2025). The resulting border shape can be seen in both plots in Figure 2.

2.2.1 Simulator

As mentioned before, using the BlueSky simulator as a headless package allows interacting with it more programmatically in discrete steps. Since the environment consists of multiple components, updating them on a per-step basis makes the implementation much more straightforward.

When the environment is created, the simulator is also set to its initial state. After initializing the BlueSky backend itself, the length of the **simulation step *dt* is set**, and remains constant for throughout all uses of the simulator. As mentioned in Section 2.2, its value was chosen to be 5 seconds.

Next, the commands that restore the state of the simulation before every run are executed. This starts by **adding a single aircraft** to the simulation, as specified by design choice 1. The aircraft is set to fly at an altitude of 35 000 feet, at a true airspeed corresponding to 0.8 Mach at that altitude. These values could be considered typical for an airliner during cruise, and are kept constant throughout the simulation runs, following design choice 3. The aircraft type used is the Airbus A320neo.

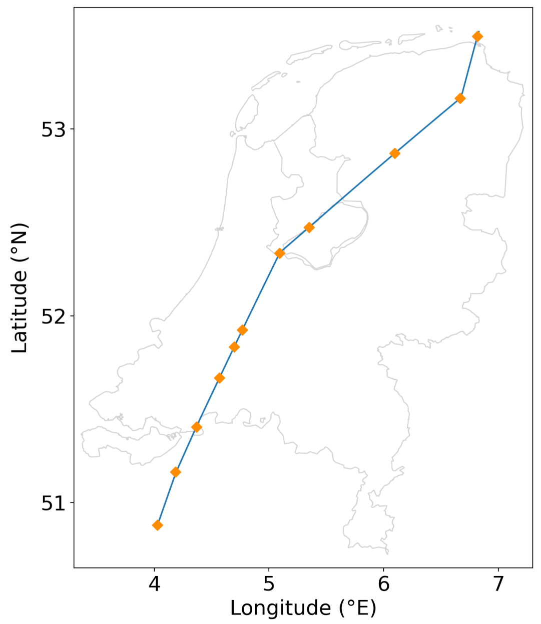
With the aircraft now existing in the simulation, the next step is to **assign route waypoints** to it. Per design choice 2, the aircraft is set to fly along a single pre-set flight path, which was chosen by handpicking navigation waypoints that could hypothetically be followed on a route from Southwestern Europe to Scandinavia. Based on the en-route charts by the respective authorities—Belgian skeyes (2025) and Dutch LVNL (2025)—the following sequence of waypoints was obtained, with the resulting flight path shown in Figure 2a.

`DENOX`, `NIK`, `WOODY`, `KUDAD`, `LARAS`, `LEKKO`, `PAM`, `LILSI`, `EH528`, `EEL`, `LABIL`, `ENGM`

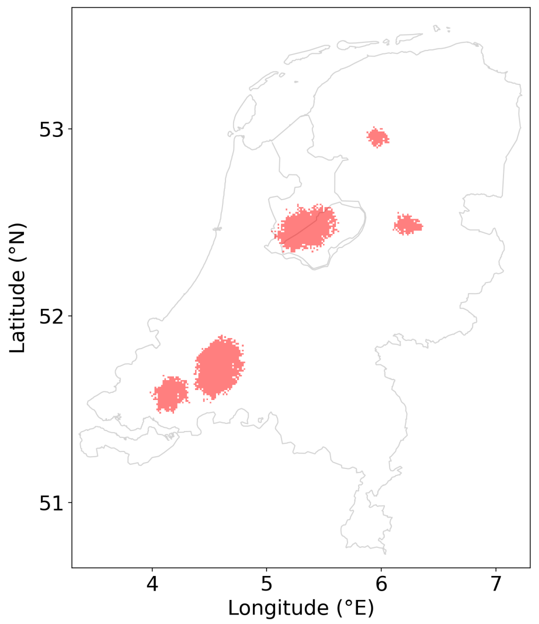
There are two exceptions in this list - the first is the `EH528`, which is a waypoint that does not appear in the charts. It was instead handpicked from the map of the simulator due to being located almost exactly on the desired path, helping to keep the distances between individual waypoints more uniform. The second one is `ENGM`, which is the ICAO code[i] of the Oslo airport. This was added so that the aircraft would have a waypoint to fly toward virtually indefinitely after leaving the map.

The last step of the initialization before every run is **adding wind** with randomized characteristics to the simulation. Unlike the rest of the weather, which is implemented separately, the wind is used by BlueSky to calculate the ground speed of the simulated aircraft, and must therefore be stored in the simulator. As stated in design choice 5, the wind is uniform across the entire map, and constant for each simulation run. It is described by the bearing it is coming from, and its speed. Since the wind parameters are initialized randomly, the bearing is simply a floating point number drawn from a uniform distribution over the half-open interval $[0, 360)$. The speed is a floating point number drawn from an exponential distribution with the

[i]Four-letter code identifying an airport, published by the International Civil Aviation Organization (ICAO).



(a) Pre-set flight path in the direction from bottom-left to top-right, with waypoints highlighted in orange



(b) Example of an initialized state of the weather grid

Figure 2: Map plots showing different components of the environment.

scale parameter $\beta = 25$. To be able to limit the wind speed to a maximum desired value, samples from the distribution are drawn repeatedly until a value below the threshold is obtained. The maximum for the speed of the wind is set to 50 knots. While winds at altitude can reach speeds over 200 knots (McCabe, 2022), the project aims to focus only on moderate wind conditions, leaving the more extreme scenarios for future studies.

With the simulator initialized and the simulation underway, all that is left to discuss are the updates of the simulator at every step. Algorithm 1 shows the SIMULATOR_UPDATE function, which is the centralized interface of the simulator used for this purpose. As shown, it accepts an action to be processed by the simulator component of the environment module. This action can either come from the agent's action space, which will be defined in Section 2.4.2, or be one of the `LNAV` and `NONE` actions used by the **simple step** and **steps with action evaluation** methods in Section 2.2, respectively.

Processing the action is as straightforward as interfacing with the BlueSky backend to translate the actions into appropriate commands. Once the action command is executed, the whole backend simulation is incremented by a single discrete step. Finally, the function retrieves information from the backend, needed elsewhere in the environment. This includes the plane's position, its true heading, the wind information, and the simulator part of the features describing the current state of the simulation for the agent.

2.2.2 Weather abstraction

As a balance between realism and ease of integration, the weather is implemented as a 2D grid of squares, covering the entire map. The discretized representation was chosen primarily to allow natural-looking updates to the weather shape using cellular automata (CA). The boolean value of each square of the grid represents to the following:

`True` - the square contains adverse weather

`False` - the square is safe to fly through

Note that in relation to these weather squares, the terms *adverse weather* and *thunderstorm* will be frequently used interchangeably, generally referring any meteorological event that should be avoided.

This two-dimensional representation of the weather is made possible by design choice 3, which avoids the need to store the weather information for different altitudes. The length of the side of each square corresponds to 1 kilometer on the map. This was not varied at any point throughout the training and evaluation, though the implementation supports changing this value through a constant. Given the small size of a single square compared to the size of the entire map, the grid serves as a good approximation of the continuous nature of real-world weather.

As said, the choice of representing weather as an array of boolean squares provides a convenient way of updating the weather as CA. However, the CA rules necessary to make thunderstorms maintain their overall shape as they move across the map, while also allowing for some organic variation, can be rather complex. Therefore, the two behaviors were implemented separately, with each executed individually during the update process.

To keep the overall shape of the weather, the **movement** is achieved by translating the entire grid in the four cardinal directions. Since the weather movement is based solely on the wind, the grid has to support the movement in an arbitrary direction on the continuous interval of $[0, 360)$ degrees. This is done by applying the following trigonometric functions to separate the bearing the wind is coming from, denoted as α, into its components corresponding to the cardinal directions. The lower bound of the values is set to zero, with the assumption that weather never moves against the wind.

$$\{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\} = \begin{cases} \max(0, -\cos\alpha) \\ \max(0, \cos\alpha) \\ \max(0, -\sin\alpha) \\ \max(0, \sin\alpha) \end{cases} \tag{1}$$

When moving the grid based on the wind direction α by a distance corresponding to the size of one square, these values represent the components of the movement in the four cardinal directions. From this, based on the timestep dt, the grid square size, and the constant wind speed, we can determine the number of squares by which the grid should shift in each of the four directions at every step.

Given that the grid can only be moved in increments of whole squares, the value for each direction is stored as an accumulator persistent across simulation steps. If the value of the accumulator becomes greater than or equal to one for any direction by adding the components from the current step, the grid is moved in that direction, and the corresponding accumulator is decremented by one. With the lower bound introduced in Equation 1, this can happen for at most two (adjacent) directions in a single step.

Since it is now assured that weather will maintain its general shape as it moves, light **altering of the shape** can be added on top of the movement. When triggered, this functionality utilizes the concept of CA, with the intention of giving the weather updates a more organic look. The implementation uses the Moore neighborhood, which is defined as the eight adjacent squares surrounding a central square. Using 2D convolution, the number of `True` neighboring cells is obtained for each cell. Based on this, the boolean value of all cells is updated according to the probabilities given in Table 1. These were obtained through trial and error, with the goal of keeping the general size of the thunderstorms roughly constant for the duration of the simulation run.

Neighbors	0	1	2	3	4	5	6	7	8
Probabilities	0	0.1	0.2	0.35	0.55	0.7	0.8	0.9	1

Table 1: Probability of a weather grid cell being `True` after an update, given the number of its `True` neighbors in the previous state.

Parameter	Description	Value interval
e	eccentricity	$[0.6, 0.9)$
b	length of semi-major axis	$[5, 20)$ km
θ	counterclockwise axis rotation	$[-\pi, \pi)$

Table 2: Intervals from which the parameters of initial weather ellipses are randomly drawn.

Having discussed the ways the weather can be updated, let us now describe how the grid is initialized before the start of the simulation. The initial state is created by first adding 5 parametrically defined ellipses. The parameters of these ellipses are all drawn from uniform distributions on the intervals given in Table 2.

The coordinates of the center point of each ellipse are also chosen uniformly, sampled from the middle 60 percent of each map dimension. The only exception to this is the first of the five ellipses, which is deliberately placed onto the location of a randomly chosen waypoint from the following list, obtained by trimming the full list of flight path waypoints at both ends. Placing one of the ellipses like this guarantees that the plane will always have to avoid at least one thunderstorm in every run.

`LARAS`, `LEKKO`, `PAM`, `LILSI`, `EH528`

Choosing to limit the area in which the ellipses can initially appear, as well as keeping only the more centrally-located flight path waypoints to place the first ellipse at, ensures the plane is able to safely enter and leave the area while flying on autopilot, with no weather nearby. Since the ellipses created by the initialization process are square-perfect, the CA update process is run 5 times after the ellipses are generated, to bring them into a more weather-like shape. An example of what this generated weather can look like is shown in Figure 2b.

From this initialized state, similarly to the simulator updates, the WEATHER_UPDATE function called in Algorithm 1 can be used to increment the state of the weather by one step. The overall weather update mechanism executes the previously described procedure to determine whether the grid needs to be moved in the current step, and performs the movement when required. If the grid is moved, the CA algorithm is always executed as well, to make the update look natural. Otherwise, the CA update is executed on a probability basis, depending on the length of a simulation step dt, and the desired number of random weather shape updates, which is set to 6 per hour of simulation time.

2.3 Agent architecture

As discussed in Section 2.1.1, the algorithm chosen for use by the agent is the DQN. Since the agent architecture was adapted from the work of Van Gelder (2023), his paper provides full elaborate description and explanation. For completeness, a brief summary is provided here. Unless otherwise specified, all parameter values are also taken directly from this paper.

Like other reinforcement learning algorithms, DQN utilizes the MDP, which defines a generic environment $\mathcal{M}$ as follows:

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\} \tag{2}$$

This means that an environment is characterized by the state space $\mathcal{S}$ and the action space $\mathcal{A}$. Furthermore, the transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ describes the transition from a state $s_t \in \mathcal{S}$ to a state $s_{t+1} \in \mathcal{S}$ by taking an action $a_t \in \mathcal{A}$. This action is given a reward $r_t \in \mathcal{R}$ according to the reward function $\mathcal{R}$. Finally, the discount factor $\gamma \in [0, 1]$ determines the relative importance of future rewards compared to immediate rewards. The entire environment $\mathcal{M}$ can be defined by describing its elements specific to our implementation, as will be formalized in Section 2.4.

The term reinforcement learning generally signifies that the agent gains experience through interactions with the environment (i.e. the specific MDP), without any prior knowledge of it. Its task is therefore to interact and experiment with the MDP, in order to optimize its behavior based the rewards received (Van Otterlo & Wiering, 2012).

Suitable for fully observable, low-dimensional state spaces, the original Q-learning algorithm by Watkins (1989) is able to achieve consistent convergence toward an optimal solution for discrete and finite action-value pairs (Watkins & Dayan, 1992). The Q-value (or *action-value*) $Q(s,a)$ represents the expected cumulative future reward obtained by taking action a in state s and following an optimal policy thereafter. These properties make Q-learning a viable option when tabular representation of all possible Q-values is feasible.

For larger-scale, partially observable, and continuous state spaces—such as the air traffic control environment in our paper—the DQN by Mnih et al. (2015) replaces the tabular representation of Q-values with a deep neural network, allowing it to handle more complex environments. However, due to the partial observability of the environment, the stable convergence toward the optimal solution is no longer guaranteed. Instead, the optimal Q-value $Q^*(s,a)$ has to be approximated using a function approximator θ:

$$Q(s,a;\theta) \approx Q^*(s,a) \tag{3}$$

In place of θ, various linear and non-linear approximators can be used. In the case of DQN, the aforementioned deep neural network takes on this role. However, the use of non-linear approximators, such as neural networks, introduces instability into the training process, potentially causing them to diverge quickly. To address this, Mnih et al. (2015) introduce two concepts—experience replay and iterative updates.

For **iterative updates** during training, two networks are used—the target model θ^- and the main policy model θ—with both sharing the same structure, and being identically initialized. The main model θ is updated after every training episode utilizing a process described below, while leaving the target model untouched. The target model θ^- is instead updated only every C episodes, by copying over the the weights of the main model θ, with C being set to 100 episodes.

During the training, the **experience replay** buffer D stores the encountered transitions (or *experiences*) of the format $\langle s_t, a_t, s_{t+1}, r_t \rangle$, up to its predefined capacity $|D| = 10\,000$, at which point the oldest transitions start being replaced by the latest ones. Instead of training the main model θ on a sequence of the most recent transitions, the replay buffer is uniformly sampled for B experiences, where $B = 64$ is the batch size. These sampled transitions are then used to update the model parameters through a process described in the following paragraphs. Sampling from previous experiences stabilizes the learning by allowing the agent to take older experiences into account as well (hence the name *experience replay*). This technique also avoids a problem of correlation between consecutive transitions, which could occur if only the latest transition was used for learning. Note that Van Gelder (2023) experimented with multiple different values for the buffer size $|D|$ and the batch size B[ii], but the aforementioned values yielded the best results for this project.

From the sampled transitions, the states s_t and actions a_t are used to obtain the Q-values $Q(s_t, a_t; \theta)$ from the main model θ, while the resulting states s_{t+1} are used to produce 'target' Q-values from the target model θ^-. These target Q-values represent the expected maximum Q-values for the next states s_{t+1} over all possible actions $a \in \mathcal{A}$. Combined with the sampled rewards r_t, and the discount factor $\gamma = 0.95$, these represent the temporal-difference target (TD) y_t:

$$y_t = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) \tag{4}$$

In combination with the Q-values produced by the main model, the TD is passed onto the loss function defined in Equation 5. Taken once again from the original architecture by Van Gelder (2023), the Huber Loss function is commonly used with DQN to reduce sensitivity to outliers (Obando-Ceron & Castro, 2021). The default value of $\delta = 1$ is used.

$$\mathcal{L}(\theta) = \begin{cases} \frac{1}{2}(y_t - Q(s_t, a_t; \theta))^2, & \text{if } |y_t - Q(s_t, a_t; \theta)| \le \delta \\ \delta(|y_t - Q(s_t, a_t; \theta)| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{5}$$

[ii]The notation B for the batch size was not used by Van Gelder—it was introduced in this paper to improve clarity.

The gradient of the loss is then used for updating the weights of the main model θ via backpropagation and gradient descent.

2.4 Formal environment definition

With the theory of the agent architecture summarized, we will now formally define our specific environment (MDP) by describing its individual components listed in Equation 2.

The simplest component to define, which does not warrant its own subsection, is the discount factor γ. As mentioned previously, the value $\gamma = 0.95$ was used.

Another component with a straightforward definition is the transition function $\mathcal{P}$. The BlueSky simulator itself acts as this function, dictating how different actions transition the simulation from one state to another. While the process with which the simulator updates the state is inherently intuitive, the specifics of the internal update process are hidden behind several layers of abstraction, and are not within the scope of this paper to cover. For our purposes, we can simply treat the transition function as a 'black box', which updates the simulation state in a consistent way. As said before, the essence of reinforcement learning is for the agent to explore an environment without any prior knowledge, which remains applicable in this case.

2.4.1 State space

The state space should include all the necessary features to provide sufficient information about the current state of the environment to the agent. At the same time, it should not be too extensive, in order to avoid providing the agent with noise that is not relevant to its task (Van Gelder, 2023).

Based on the subject they describe, the features could be divided into two different categories. These subjects are the attitude of the aircraft, and the weather information. This distinction can already be seen in Algorithm 1, where the two categories were identified as `simulator_features` and `weather_features`, respectively.

As explained in design choice 2, the features of the state space are taken relative to the aircraft's orientation whenever possible, to try to mitigate route-specific behaviors. In combination with design choice 3, this means that the agent only needs to be provided with very little information regarding the **attitude of the aircraft**, since most of it will be constant throughout all simulation runs. Nevertheless, making the agent aware of the influence the wind has on its trajectory would be beneficial. A simple way to achieve this is to provide the agent with the aircraft's ground speed gs. Given the constant true airspeed, any variation in the ground speed directly reflects the changes in relative wind speed and direction. This allows the agent to correlate changes in the gs feature with variations in flight path turn radius and potentially other subtle factors, which it can learn to identify.

Secondly, while design choice 4 specifies that the agent does not need to navigate the aircraft the whole way, it should still be aware of its current deviation from an optimal path. To achieve this, the agent is given a relative heading to the last relevant waypoint on the map, as a general indication of the preferred direction. From the waypoint list defining the flight path, the `LABIL` waypoint fits this criterion, being the top-rightmost point highlighted in Figure 2a. This difference between the heading of the aircraft and the optimal heading is the feature hdg_{opt}, reported in degrees on the interval $[-180, 180)$. Specifically, $hdg_{opt} = 0$ represents the aircraft heading directly for the waypoint, with positive values indicating that the waypoint is on the right, negative values on the left, and $hdg_{opt} = -180$ directly behind.

Providing the agent with **weather information**, the following approach was inspired by the implementation of obstacle detection by Rybchak and Kopylets (2024) shown in Figure 1. The weather detection area is limited to a 180-degree circular sector in front of the plane, with a 30 km radius. To provide additional information about the directional density of the weather, this half-circle is split into three sectors. The middle sector is 20 degrees wide and centered on the plane's heading, while the two remaining sectors each cover the remaining 80 degrees on their respective sides. A representation of this can be seen in Figure 3. The number

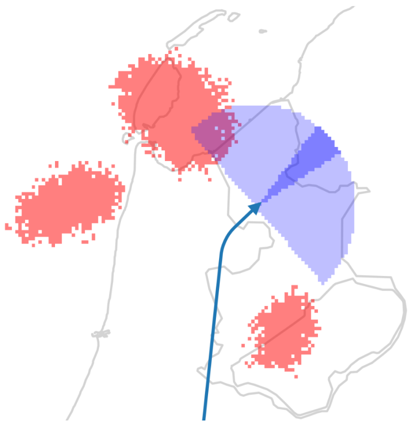


Figure 3: Representation of the aircraft's weather detection area in blue, with the middle sector shown darker than the side sectors.

of adverse weather squares in each of the three sectors gives us another 3 features for the state space—the n_L, n_C, and n_R for the left, center, and right sector, respectively. In the implementation, the 30-kilometer radius is divided by the size of a grid square, obtaining the distance limit as a number of squares. All cells that fall within this Euclidean distance and appropriate bearing angles relative to the square the plane is currently in are then counted for the respective sectors.

While the square counts give the agent a great overview of the weather amounts in the general directions, further information about the squares within the detection range helps the agent avoid taking unnecessary actions. This is achieved by providing the direction hdg_{near} and distance $dist_{near}$ to the nearest weather square in the aforementioned half-circle where weather can be detected. Similar to the features introduced in the previous paragraph, both the heading and the distance are calculated relative to the square the plane is currently in, instead of using its precise position. Since the resolution of the weather grid is quite high, this does not have significant effect on the training. The hdg_{near} feature is given to the agent in the same relative form as hdg_{opt}, following the justification from design choice 2.

By definition, the features hdg_{near} and $dist_{near}$ are only valid when weather is detected. However, for transitions that result in the detection area being completely free of weather, specific values had to be defined for these features to be able to represent the 'out of range' state. In such cases, the values $hdg_{near} = 180$ and $dist_{near} = 10\,000\,\text{km}$ are used, indicating that the closest weather square is located directly behind the plane, at a significant distance. While the range of values for hdg_{near} calculated by the simulation environment can only ever fall on the interval $[-90, 90]$, the value of 180 still logically represents a direction behind the plane, and did not cause any issues during training.

Combining all the aforementioned, the gs and hdg_{opt} make up the features extracted from the simulator (`simulator_features` in Algorithm 1), while the n_L, n_C, n_R, hdg_{near}, and $dist_{near}$ are calculated by the weather abstraction (`weather_features` in Algorithm 1). All together, these 7 features describe a state $s_t \in \mathbb{R}^7$ for arbitrary timestep t, and by extension also formally define the full state space $\mathcal{S}$:

$$s_t = \{gs, hdg_{opt}, n_L, n_C, n_R, hdg_{near}, dist_{near}\} \in \mathcal{S} \tag{6}$$

2.4.2 Action space

Given design choice 3, the actions that the agent can take will be limited to controlling the plane's heading, i.e. the horizontal direction, without the possibility to adjust its speed or altitude. Since DQN requires discrete action space to be able to calculate the Q-values of individual actions, the amount by which the heading will be varied has to be preassigned to the actions. With this in mind, the action space $\mathcal{A}$ is defined as follows:

$$\mathcal{A} = \{\texttt{TURN\_LEFT}, \texttt{KEEP\_HDG}, \texttt{TURN\_RIGHT}\} \tag{7}$$

There are two differences compared to the original action space design by Van Gelder (2023). Firstly, since we strive to minimize flight path disruptions caused by excessive turning, the turn actions correspond to a heading change of 20 degrees, instead of the original 45. Secondly, the option to continue on autopilot from the original action space has been replaced by the option to keep the current heading, without resuming the autopilot navigation. Following design choice 4, this action is meant to be taken when weather is in range, but does not obstruct the aircraft's current trajectory.

2.4.3 Reward function

To provide the agent with useful feedback, allowing it to learn efficiently, the reward function $\mathcal{R}$ defines the reward corresponding to different outcomes of the action a_t taken by the agent. Remember that the environment only calculates the reward for its *steps with action evaluation* method, since that is when an actual agent action is used for the update. For the purposes of reward calculation, the environment tracks the aircraft's closest approach to a weather square over the course of the action cooldown. Let us denote this shortest approach distance d_{min}. Note that this distance is only calculated from the squares within the 180 degree detection sector in front of the plane, by taking the minimum of the $dist_{near}$ values of all states encountered during the action cooldown. Any squares outside of this sector are not a concern, given that weather moves much slower than the plane itself.

Being the main focus point of the entire paper, failing to avoid approaching adverse weather will be penalized with a significant negative reward. Rather than simply punishing coming into contact with a thunderstorm, a safe distance limit $d_{lim} = 5\,\text{km}$ will be set. Entering this radius, in the sense that $0 < d_{min} < d_{lim}$, will cause the agent to receive a reward of -20. If the plane enters the thunderstorm itself, i.e. when $d_{min} = 0$, a larger negative reward of -50 is applied instead. Since these are the most important rules for the agent to learn, no other rewards will be applied in these cases.

If the agent avoids the weather safely, i.e. if $d_{min} \geq d_{lim}$, additional rewards are applied to further improve its behavior, with two main objectives. Firstly, we generally do not want the agent to turn unless it is necessary. Since this is of less importance compared to avoiding the weather, we want to encourage the agent positively when it keeps the same heading, rather than punishing it when it chooses to turn. Therefore, the action `KEEP_HDG` receives a positive reward of 10. This justification is similar to the one made by Van Gelder (2023) with regard to his LNAV Incentive Reward Function.

Furthermore, in Section 2.4.1, the hdg_{opt} feature was introduced to give a general indication of the preferred direction for the plane. Rather than forcing the agent to keep as close as possible to the pre-set flight path, this feature is used to punish the agent if it turns unreasonably far away from the optimal heading. Specifically, a negative reward of -10 is applied if the agent's action results in the plane being deviated more than 90 degrees from the optimal heading, unless the action improved the deviation. Setting the angle to 90 degrees ensures that the agent will have sufficient room to maneuver around the weather in the vast majority of scenarios. To make the definition of the reward function more concise, we introduce the following term for any transition $\langle s_t, a_t, s_{t+1}, r_t \rangle$:

$$\texttt{deviated} := |hdg_{opt\_end}| > 90 \land |hdg_{opt\_end}| > |hdg_{opt\_init}| \quad \text{where } hdg_{opt\_init} \in s_t,\ hdg_{opt\_end} \in s_{t+1} \tag{8}$$

It should be noted that rewarding the agent based on this condition goes against the general notion that reward functions should be kept as simple as possible to facilitate stable learning. The condition is also based on state space features that the agent has direct control over, rather than features that are a result of its interaction with the environment. However, this has been a conscious compromise with respect to design choice 4, allowing us to provide the agent with feedback about the direction it should prefer, without requiring it to navigate the plane along the pre-set route completely on its own.

With all of the individual components discussed, the full reward function definition is given in Equation 9. Notice that it is possible for the $a_t =$ `KEEP_HDG` and `deviated` conditions to be satisfied simultaneously, resulting in the combined reward of 0.

$$r_t = \begin{cases} -50 & \text{if } d_{min} = 0 \\ -20 & \text{if } 0 < d_{min} < d_{lim} \\ 10 & \text{if } a_t = \texttt{KEEP\_HDG} \wedge d_{min} \geq d_{lim} \\ -10 & \text{if } \texttt{deviated} \wedge d_{min} \geq d_{lim} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

2.5 Network design

While other parts of the architecture required considerable changes to accommodate the weather avoidance domain, the design of the agent's neural network remained very similar to the original network used by Van Gelder (2023). Here, the only concern was to ensure that the dimensions of the input and output layers were consistent with the size of the newly defined state and action spaces, respectively. The input layer was therefore shrunk to $|\mathcal{S}| = 7$ neurons instead of the original 11, while $|\mathcal{A}| = 3$ neurons were kept for the output layer, since the action space size is the same.

The two hidden layers remained unchanged, consisting of 64 and 32 neurons, respectively. The activation functions were also kept the same, utilizing the Rectified Linear Unit (ReLU) function for the hidden layers, and a regular linear activation function for the output layer. An overview of the entire network is provided in Figure 4.

For training the network, the Adam optimizer with a learning rate of $\lambda = 0.0001$ was adopted without any modifications. This architecture is used by both the main model θ and the target model θ^-, to allow for the iterative updates explained in Section 2.3.

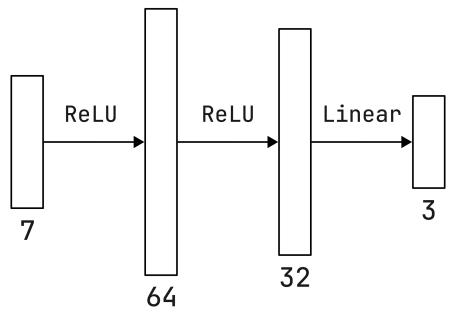


Figure 4: Network architecture used for both the main model θ and the target model θ^-. Adapted from Van Gelder (2023).

2.6 Training

Finally, with all the theory and implementation regarding the DQN agent covered, let us now describe the training process.

Through experimentation, it was found that the agent is asked to take an action approximately 30 times on average during a single simulation run, with the rest of the navigation being handled by the autopilot. Since this value is sufficiently large, it is reasonable to make a single training episode correspond to one simulation run, consisting of traveling the entire pre-set route shown in Figure 2a.

Being a direct implementation of design choice 4, Algorithm 2 shows the iterative process of updating the simulation throughout a single episode. It uses the two external interfaces of the environment introduced in

Section 2.2, asking the agent to select an action if needed. When the chosen action is executed and evaluated, the information making up the transition is pushed into the experience replay buffer of the agent, and the reward is added to the cumulative sum of rewards for the current episode. Each run is stopped when the `done` parameter returned by the environment indicates that the simulation should terminate, which can occur either when the step limit is reached, or when the plane is about to leave the map.

As the algorithm shows, the transition $\langle s_e, a_e, s_{e+1}, r_e \rangle$ and reward r_e are not recorded in the rare case that the simulation ends with the *steps with action evaluation* update. In such a scenario, it is possible that the action cooldown did not fully complete, making the returned reward and resulting state inaccurate.

Algorithm 2 Interaction between the agent and the environment during a single episode

```
Input: Initialized environment $E$ and agent $A$, episode number $e$
Output: Total reward $r_{\text{total}}$ for all actions taken in the run
  done ← False
  $r_{\text{total}} \leftarrow 0$
  $s_e \leftarrow E$.RESET()
  while ¬ done do
      if $E$.IS_AGENT_NEEDED() then
          $a_e \leftarrow A$.SELECT_ACTION($s_e$, $e$)
          $s_{e+1}$, $r_e$, done ← $E$.STEPS_WITH_ACTION_EVAL($a_e$)
          if ¬ done then
              $A$.PUSH_TO_BUFFER($s_e$, $a_e$, $s_{e+1}$, $r_e$)
              $r_{\text{total}} \leftarrow r_{\text{total}} + r_e$
          end if
      else
          $s_{e+1}$, done ← $E$.SIMPLE_STEP()
      end if
      $s_e \leftarrow s_{e+1}$
  end while
```

The described process is repeated for 5 000 episodes, providing the agent with sufficient time to optimize its behavior. After each episode, the main policy network of the agent is updated, while the target network is overwritten by the main policy network every 100 episodes, as explained in Section 2.3. To ensure sufficiently diverse exploration of the environment, the agent utilizes the decayed ϵ-greedy approach used by Van Gelder (2023) for selecting the action a_t:

$$a_t = \begin{cases} \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s_t, a) & \text{with probability } 1 - \epsilon \\ a_t \sim U(\mathcal{A}) & \text{with probability } \epsilon \end{cases} \tag{10}$$

With the parameter set to $\epsilon = 1$ at the start of the training, the agent initially selects a random action sampled from a uniform distribution over the action space $\mathcal{A}$, to allow for maximum exploration. From there, the parameter is decreased linearly by $\Delta\epsilon = 0.002$ per training episode, down to a fixed minimum of $\epsilon_{min} = 0.01$ reached by episode 495. As ϵ decreases, the agent is more likely to select its preferred action, which corresponds to the largest learned Q-value for the given state s_t.

Table 3 summarizes all important parameters used for the training. As stated previously, the values are directly based on the work of Van Gelder (2023), with the majority of them taken without any modifications. The only exceptions are the batch and buffer sizes, which Van Gelder (2023) varied across his experiments, but which were kept constant across all results presented in this paper. Different values of $\Delta\epsilon$ and $|D|$ were briefly experimented with before carrying out the aforementioned training process, with their influence on the training behavior covered in Section 4.2.3.

Parameter	Description	Value
γ	reward discount factor	0.95
λ	learning rate	0.0001
C	target model update rate	100
ϵ	initial exploration parameter	1
ϵ_{min}	minimum exploration parameter	0.01
$\Delta\epsilon$	linear decay factor for ϵ	0.002
B	batch size	64
$\lvert D\rvert$	buffer size	10 000

Table 3: Values of training parameters.

Lastly, Table 4 lists other simulation parameters specific to the project, which were mentioned throughout the Methods section. These range from the characteristics of the weather implementation, to the values for the $dist_{near}$ and hdg_{near} features of the state space when no weather is detected. The angle intervals of the weather detection area are given relative to the plane's heading.

Parameter description	Value
Simulation step dt	5 s
Simulation length limit	400 steps
Weather grid square size	1 km
Weather detection radius	30 km
Weather detection area (full)	$[-90, 90]$ deg
Weather detection area (sectors)	$[-90, -10), [-10, 10], (10, 90]$ deg
Weather safe distance limit d_{lim}	5 km
Random weather shape updates	6 per hour of simulation time
Maximum wind speed	50 kts
'Out of range' value for hdg_{near}	180
'Out of range' value for $dist_{near}$	10 000 km

Table 4: Values of other simulation parameters.

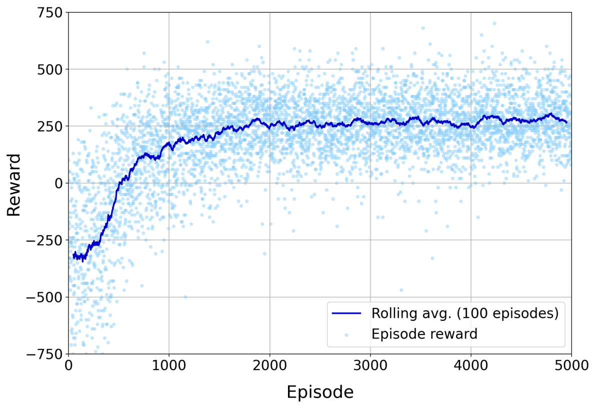
3 Results

In this section, we will discuss the experimental results obtained using the described methodology. Before covering the agent's performance, the training behavior shall also be examined, given that it is an important part of any reinforcement learning implementation.

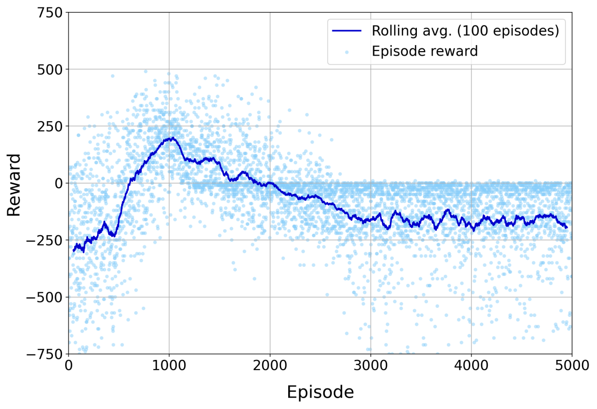
3.1 Training stability

As said previously, each agent was trained for 5 000 episodes, using the parameter values given in Table 3. To be able to observe various trends in training behavior, 50 agents were trained simultaneously. Figure 5 shows examples of per-episode reward progression for two different training runs.

One can clearly see the initial exploration phase of 495 episodes, where the actions of the agent were progressively less likely to be picked at random, according to the exploration strategy given by Equation 10. This is followed by a gradual increase in the rewards, as the agent starts selecting majority of the actions based on its knowledge. The two plots also demonstrate an important trend—not all training runs result in the episode reward converging on a high value. For successful runs, such as the one in Figure 5a, the average



(a) Reward progression of a successful training run



(b) Reward progression of an unsuccessful training run

Figure 5: Plots showing the progression of episode rewards during different training runs.

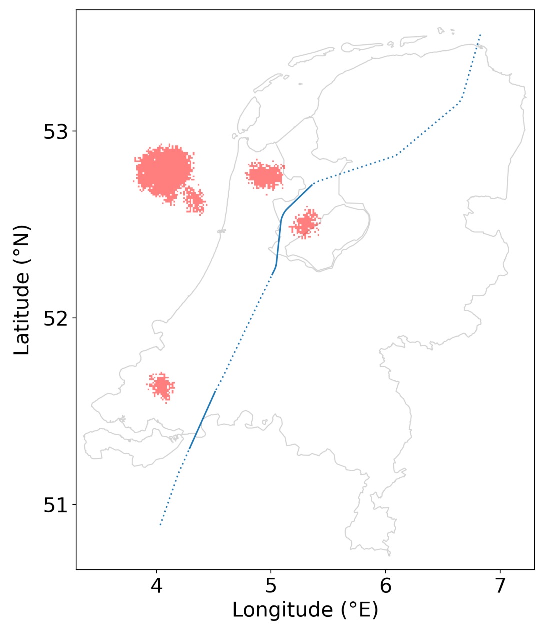
value of the reward stabilizes around 250, at which point majority of the variability comes from the random nature of the weather. In contrast, for unsuccessful runs, the average reward falls well below zero, as shown in Figure 5b. Of the 50 agents that were trained, 19 failed to converge to the high positive reward.

From Figure 5b, it is apparent that on top of simply not stabilizing on a high value, the episode rewards also gradually start being limited to a maximum of 0. Since any positive reward can only be obtained by taking the `KEEP_HDG` action, as specified by the reward function in Equation 9, this means that the output neuron corresponding to that action stops receiving any meaningful activation signals as the network is trained. This was thought to be the result of the neurons in the previous layers suffering from the so-called 'dying ReLU' problem. The term refers to the scenario described by Maas et al. (2013), where a ReLU neuron remains in a perpetually inactive state due to all of its inputs being negative, preventing gradient backpropagation. Addressing the issue of zero output for negative inputs, the Leaky ReLU (Maas et al., 2013) and the Exponential linear unit (ELU) (Clevert et al., 2016) functions allow a small gradient to pass through, instead of blocking it entirely. However, changing the activation function of the hidden layers from ReLU to either of these alternatives did not improve the described training behavior. Since the approximate 60% training success rate is more than sufficient to be able to evaluate the performance of the agents, the design of the neural network was kept as defined in Section 2.5.

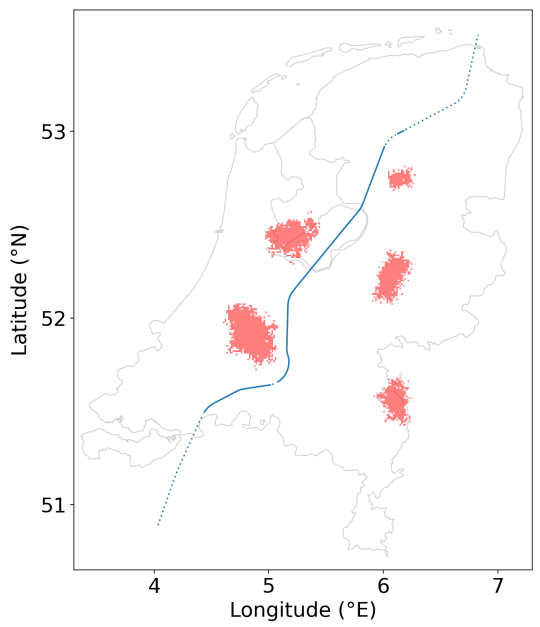
Once the agent approaches the aforementioned threshold reward of about 250 per episode, it generally exhibits the same behavior as any well-trained agent, which is described in the next subsection. This is the case even for agents that later end up 'collapsing' below the episode reward of 0. As an example, if an agent reaches the threshold reward after 2 000 episodes, it can already be used in its current partially trained state, even if its future version collapses as the training process continues.

3.2 Agent performance

The reward plot in Figure 5a already demonstrates the agent's ability to obtain high rewards, even with the 1 percent random exploration chance ($\epsilon_{min} = 0.01$) present during training. As discussed previously, the main takeaway from the reward plot of the unsuccessful training run in Figure 5b is that the agents with such training pattern learn to never select the `KEEP_HDG` action. However, no other significant conclusions can be made from these plots alone. Let us therefore discuss the agents' performance further in a qualitative way, by exploring the individual simulation runs they produced. For completeness, we will cover the runs by agents that trained successfully, as well as those that failed to converge to the positive reward threshold.



(a) Simple avoidance maneuver (reward 180)



(b) Complex avoidance maneuver (reward 350)

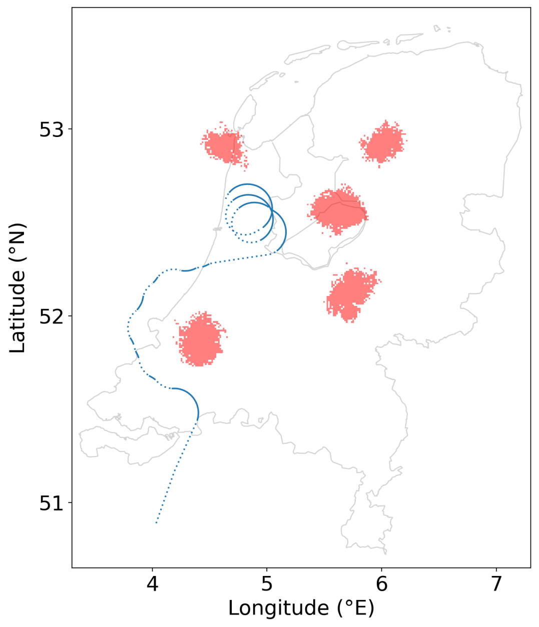
Figure 6: Map plots showing examples of optimal performances of a well-trained agent.

Since the positions of the weather and the aircraft change throughout the simulation, it is impossible to represent both in a static plot with full accuracy. As an approximation, the flight path of the aircraft is shown, and the state of the weather is taken at step 200, which is half of the maximum simulation length. Because of this, some visualizations had to be discarded, as the weather movement due to wind made it appear as though the aircraft flew through a thunderstorm, when in reality the flight path crossed the position of the weather at different point in time. In the plots, the segments flown by the agent are represented by solid lines, while the segments where the autopilot was in control are shown as dotted lines. This visualization approach was employed for all plots showing the performance of the agent.

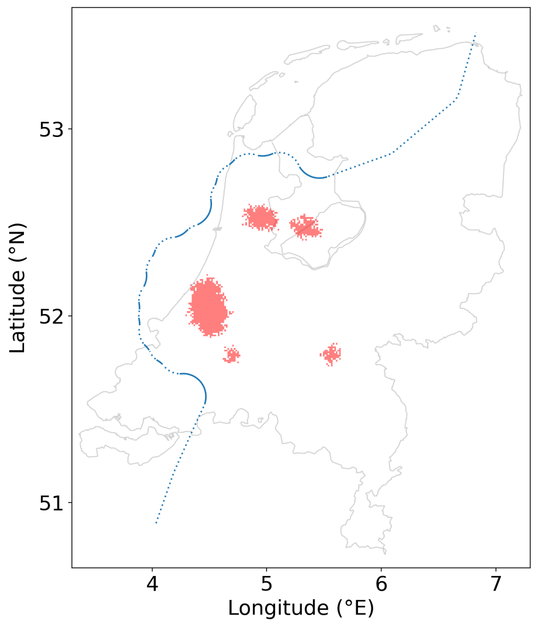
Figure 6 shows some of the best simulation runs of the **well-trained agent**. In both situations, it successfully evades all weather, while keeping the flight path looking realistic, without unnecessary turns. In cases where the weather does not need to be avoided, the agent maintains the current heading and continues without any disruption to the flight path. When a turn is necessary, it is executed quite early after the weather is detected, which could be caused by the relatively short detection range of 30 kilometers, making the reaction window somehow narrow.

The plots also indicate that the size of positive rewards is proportional to the amount of weather the agent encounters, i.e. the amount of actions it takes. With less weather along the way, such as in Figure 6a, there are fewer opportunities to use the positively-rewarded `KEEP_HDG` action, resulting in a lower reward of 180. When the agent flies parallel to the larger volume of weather in Figure 6b, the greater number of `KEEP_HDG` actions results in a higher reward of 350.

Naturally, not all simulation runs yield an ideal performance. Section 4.1 relates the project's design choices to the possible reasons for less optimal behavior, examples of which can be seen in Figure 8. In these cases, the agent still often manages to avoid turning where it is not necessary. However, it is evident that the shape of the flight paths is not very natural, in the sense that an aircraft would be unlikely to ever avoid weather in this way in real-world traffic. The important conclusion, even from these imperfect runs, is that that the well-trained agent manages to avoid the weather in vast majority of scenarios, fulfilling the main objective of the project.



(a) Undesirable 'spinning' behavior (reward -180)



(b) Undesirable 'contour' behavior (reward -50)

Figure 7: Map plots showing examples of undesirable performances of poorly-trained agents.

Lastly, let us briefly take a look at the performance of the **poorly-trained agent**, for which the training did not converge to a high reward. Here, two distinct behaviors can be observed. For agents that did not reach the episode reward threshold of about 250 before collapsing, as in the case of the reward progression in Figure 5b, the plane often ends up 'spinning' in circles upon encountering weather, which can be seen in Figure 7a. As previously concluded from the reward progression plot, the poorly-trained agent indeed ignores the `KEEP_HDG` action, using the turning actions instead to keep itself at roughly constant distance from the weather. This results in the flight path looking like an offset contour of the weather, clearly visible in both plots in Figure 7.

In the other training progression scenario, where the agent initially stabilized on the threshold reward for at least a few hundred episodes, but ended up collapsing later, the 'spinning' behavior tends to occur less often, with the aircraft following the 'contour' flight path instead, as shown in Figure 7b. However, both the 'spinning' and 'contour' behaviors occur in all poorly-trained agents, to varying degrees. Interestingly, even with the unsuccessful training, the thunderstorms are often completely avoided, though weather encounters are somehow more frequent compared to well-trained agents.

Notice that the runs in Figure 7a and Figure 8b were both cut short by the step limit introduced in Section 2.2, showing that it works as expected. In both of these runs, the agent took a much longer path than necessary to avoid the weather.

4 Discussion

This section further justifies and explains the agent behavior presented in Section 3.2. Alternative versions of the project architecture are also discussed to clarify how we arrived to the final setup described in Section 2. Next, the implications of the project's results for the industry are covered, representing the main outcome of the project. Finally, proposed directions of further work are presented, be it smaller-scale steps to improve the project's architecture, or the larger steps necessary to take the design from its current state into a real-world deployment.

4.1 Agent behavior

While the agent showed good performance in most of the simulation runs, there were still some suboptimal flight paths taken. These need to be linked to the project's design choices in order to justify them, and to help improve the resulting behavior in future iterations of the project.

The main source of 'unnatural-looking' flight paths, characterized by excessive or unnecessary turns, are the transitions from agent to autopilot control, implemented based on design choice 4. When the agent completes the avoidance maneuver, to the point where there are no more weather squares detected in the 180-degree sector in front of the plane, the autopilot takes over, turning toward the appropriate heading to resume the planned route. However, in doing so, it occasionally makes the plane face the weather again, resulting in the agent stepping in, correcting the turn, and continuing on the new heading. This behavior can be seen in both Figure 6b and Figure 8b, at approximately 51.75° N, 5.25° E.

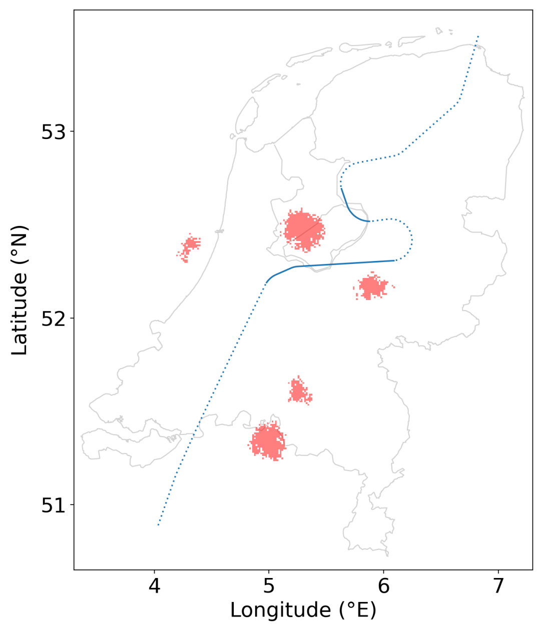
Based on the flight paths in these plots, one can notice that when the agent resumes control after the short autopilot segment, it briefly continues the turn toward the weather, before finally turning away. A closer examination revealed that the agent's `KEEP_HDG` action, corresponding to switching off lateral navigation in the BlueSky simulator, does not stop the turn previously initiated by the autopilot—which is not the behavior expected from the simulator. Consequently, the agent keeps choosing the positively-rewarded `KEEP_HDG` action until the continued turn causes the plane to face the weather again. At that point, the agent resumes taking appropriate avoidance maneuvers, as it would otherwise.

When attempting to correct for the simulator behavior manually—by making the `KEEP_HDG` action disable the lateral navigation and simultaneously set the heading to remain fixed—the training became very unstable, with barely any agents converging to a positive reward. Given this rather unexpected outcome, a decision was made to keep the implementation in its current form, tolerating the slight imperfections in the flight path.

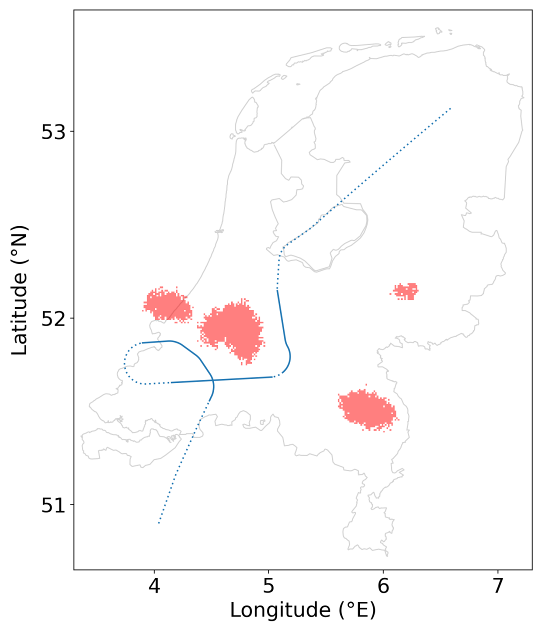
In cases where the control is given back to the autopilot when the plane is flying significantly off-course, the transition often results in a sharp turn, sometimes well over 90 degrees. When multiple back and forth transitions between the agent and the autopilot control happen in quick succession, the flight path can end up looking completely unrealistic, as can be seen in Figure 8a. In the ideal runs in Figure 6, the agent leaves the plane very close to the optimal heading, resulting in little to no course change when the autopilot takes over. Since the agent is not aware of its position relative to the pre-set flight path—other than the hdg_{optim} indicating the general desired direction—it is largely a product of chance whether it ends up being on course after avoiding the weather. In the most common scenario where the agent avoids a single thunderstorm at a time, the transitions are often smooth, with the occasional autopilot-initiated turns being stopped under the agent's control, as discussed at the beginning of this subsection.

Additionally, the limited ability to navigate the aircraft along its intended route also sometimes results in the agent flying parallel to the thunderstorms for extensive amount of time. Due to the constant proximity of weather in such situations, the autopilot is unable to take over, resulting in the agent taking the plane far off course. An example of this is the long straight segment in eastern direction in the center of Figure 8a, though more extreme cases were also encountered.

On rare occasions, the control transition problems are made worse by the agent's general difficulty in handling thunderstorms that form a longer 'barrier' across the pre-set flight path. When the aircraft encounters weather shaped in this way, the limited information that the state space provides to the agent makes it virtually impossible for the situation to be solved well. In such cases, poorly-trained agents usually immediately resort to spinning in circles without any any further progress, while the well-trained agents tend to be able to resolve the situation in an acceptable way, given the limitations. However, when the plane navigates along the barrier to avoid it, it often ends up significantly off-course, which occasionally produces a full loop in the flight path, due to the subsequent transition to autopilot control. This scenario can be seen in Figure 8b—it is virtually the only case where a full loop occurs in a flight path produced by a well-trained agent.



(a) Multiple autopilot-agent control transitions resulting in an unrealistic flight path (reward 160)



(b) Longer weather 'barrier' causing a full loop in the flight path

Figure 8: Map plots showing examples of mediocre performances of a well-trained agent.

Since the described behavior is a combined result of design choice 4 and the limited state space, the best solution would be a significant redesign of the project architecture, accounting for these shortcomings. However, given the limited scope of the project, the `deviated` term introduced in Equation 8 for the reward function in Equation 9 is considered an acceptable middle ground. By penalizing the agent when it turns further than 90 degrees away from the optimal heading, the sharp turns and full loops after switching to autopilot control can be kept to a minimum.

As mentioned, the main constraint when avoiding larger thunderstorms is the limited detection area that the agent is aware of, since the features of the state space only provide information about weather located within the 30 km radius. Seeing the entire map in the plots, it is straightforward to find an optimal path the agent should take. From such perspective, the thunderstorm 'barrier' would be easily avoided by taking an action earlier, making the detour much more gradual. However, with the limited detection range, the agent has to decide on optimal action in each specific situation, based on the information at hand. Similar justification also applies to minimizing the overall length of the path taken, since the agent focuses on solving the problem locally for each step.

With this explanation, it is apparent that, apart from the agent lacking information about more distant thunderstorms, the actions being handled individually is also a potential downside. Two actions might appear identical in their respective state spaces and rewards, while being completely different if they were to be evaluated as part of the full flight path taken. As said before, a fundamentally different architecture would be needed to eliminate these problems. To avoid going beyond the scope of the project, this overhaul was left for the Future work section, where the possible solutions are covered in detail.

4.2 Alternative architectures

Having explained the reasons behind specific agent behaviors, we will now explore other architectures which were tested in an effort to improve the behavior, leading to the final design presented in Section 2. While these were not tested as extensively as the final version, their influence on the outcomes of the simulations will still be discussed, in the hope of highlighting or discouraging certain directions of future research.

Throughout the section, one might notice that the design choices were not yet present during the initial implementations. Instead, they were formalized later to keep track of the compromises made with respect to the scope of the project, and their justifications.

4.2.1 State spaces

Unlike most other parts of the project, the state space had to be significantly modified compared to the original implementation by Van Gelder (2023). Given that the same agent architecture is applied to a different domain, a distinct set of features was needed to provide relevant information about the environment.

In the very initial version of the project, the number of adverse weather squares in the detection area in front of the plane was given as just two 90-degree sectors, i.e. n_L and n_R. With the addition of n_C, taking the detection sectors to their current form, the agent saw significantly improved performance and faster training convergence. These improvements could be interpreted as the agent gaining information about the weather in the exact direction it is going, as well as the directions it could turn toward. In this configuration, each of the sectors directly corresponds to a specific action from the action space, making agent's decisions more straightforward.

The speed and heading of the wind were also part of the initial version of the state space. However, they were later removed, with the assumption that the speed of the aircraft is always significantly higher than the speed of the wind (i.e. the speed at which the weather moves), making these features less relevant. As a result, the agent has to react to the weather movements in real time, rather than learning to predict them from the wind information. While some level of weather movement prediction will certainly be helpful in future follow-up projects, this approach to weather avoidance was deemed sufficient for our purposes. Crucially, no loss in agent performance was observed, despite shrinking the state space by two features. As a result, more important information could be added to the state space, such as the aforementioned n_C.

With the removal of the wind information, a previously included hdg_{true} feature, which represented the true heading of the plane, was also removed. Consequently, the hdg_{near} was adjusted from its initial absolute form to the current relative form described in Section 2.4.1. After this change, the agent no longer receives information about the direction the plane is flying, and its behavior is therefore completely orientation-independent, limiting the risk of learning route-specific behavior due to design choice 2. All directional information about the weather is represented by the predefined relative directions of the borders of the detection sectors, and by the—now relative—hdg_{near}. As mentioned, these modifications did not result in any observable loss of agent performance, while simplifying the state space and making the individual features more useful.

The last significant change made to the state space concerned the hdg_{opt}, which was initially defined as a heading to the next waypoint in the flight path. As discussed in Section 2.4.1, in the final version, this feature represents the relative heading to the last waypoint on the map, providing the agent with a general direction to fly toward. Following this adjustment, it became apparent that the initial version caused the agent to receive lower rewards overall. Since the aircraft is always closer to the individual waypoints compared to the last one, the value of hdg_{opt} varied much more rapidly. Sudden changes could also occur when the target waypoint was updated as the plane progressed through the flight path, potentially contributing to training instability, since the agent had no way to prevent these rapid shifts in the value of the hdg_{opt}.

4.2.2 Action spaces

While the original state space by Van Gelder (2023) had to be adjusted to account for the new domain, the action space was initially taken without any modifications, as defined in Equation 11. Although the project domain is different, the means for the plane to avoid obstacles, be it other planes or weather, remains largely the same.

$$\mathcal{A} = \{\texttt{TURN\_LEFT}, \texttt{LNAV}, \texttt{TURN\_RIGHT}\} \tag{11}$$

Here, the `LNAV` action represents the command to resume (or continue) the autopilot navigation, just like in the original paper. The same action is also used by the environment when no weather is detected, as explained in Section 2.2. Remember that the turning actions correspond to a turn of 20 degrees, as compared to the 45 degrees used by Van Gelder (2023).

Following initial testing runs, it became clear that the `LNAV` action was chosen very rarely, resulting in a 'contour' flight path shape, similar to what was shown in Figure 7b. The agent likely failed to fully comprehend the consequence of this action, given that it lead to turns in different directions, depending on the plane's location relative to the intended flight path. In the rare cases when the action was used, it often resulted in what could be called a 'bouncing' behavior. The agent would turn away from weather, but subsequently take the `LNAV` action again, turning back toward the weather, requiring another turn action to avoid it—sometimes repeating the cycle several times.

In an attempt to improve the behavior, the `KEEP_HDG` action was added to the action space, allowing the agent to continue flying in its current direction:

$$\mathcal{A} = \{\texttt{TURN\_LEFT}, \texttt{LNAV}, \texttt{KEEP\_HDG}, \texttt{TURN\_RIGHT}\} \tag{12}$$

While this resulted in a slightly smoother flight paths, which were later tweaked further by adjusting the reward function, the `LNAV` action was still used rarely, even when given higher positive reward than the `KEEP_HDG` action. Therefore, a decision was made to remove the `LNAV` action completely, bringing the action space to its final form, given in Equation 7.

Removing the `LNAV` action meant that the agent effectively lost the ability to navigate the pre-set flight path on its own. However, the state space and reward functions were designed to provide a viable alternative, as justified in Section 4.1. With fewer actions to choose from, the agent became more likely to take the well-rewarded `KEEP_HDG` action whenever possible, resulting in the smoother flight paths shown in Figure 6.

4.2.3 Training stability improvements

During development, a considerable amount of time was spent on tweaking and testing various changes to ensure a satisfactory percentage of agents trained successfully. The two most significant factors that could be changed to improve the training stability were the reward function given by Equation 9, and the training parameters listed in Table 3.

Initially, the magnitudes of the rewards in the **reward function** were heavily inspired by the three different functions used by Van Gelder (2023), limiting the values to the range between 0 and 10. However, it quickly became apparent that the magnitude of the values influenced the training stability significantly, with greater magnitudes resulting in more stable learning, even for the same relative size of individual rewards. Therefore, the original reward magnitudes of 1, 2 and 5 were multiplied by a factor of 10, yielding the final reward function in Equation 9.

Another subject of experimentation was the angle used in the `deviated` term in Equation 8. From seeing the whole picture in the flight path plots in Figures 6-8, it might appear that a far shallower angle would be sufficient to avoid the weather in most situations. However, one has to keep in mind that the agent is only aware of thunderstorms located in the 30-kilometer detection range, requiring somehow sharper turns to evade the weather within this distance. During testing runs, where the angle was gradually reduced from 90 to 60 degrees, the training stability progressively worsened—an indication that the agent was struggling to avoid the weather within the limited detection range, without getting penalized for deviating from the optimal heading too far.

The constraint this angle places onto the agent is even narrower than might be expected. As explained in Section 2.4.1, the hdg_{opt} feature is only an approximation of the optimal direction, indicating the relative heading to the final waypoint on the map. However, since the pre-set route is not a straight line, the absolute value of hdg_{opt} can vary by as much as 30 degrees, even when the plane flies the whole route on autopilot with no deviations. This fact already limits the maneuvering space for the agent, and is made worse if the angle in the `deviated` term is made narrower. The value of 90 degrees was therefore selected as most suitable, since it forces the agent to keep moving toward the general direction of the final waypoint, without restricting its maneuverability too much.

The second set of factors that significantly influenced the training stability were the **training parameters**. In his paper, Van Gelder (2023) experimented with three different values for both the buffer size $|D|$ and the batch size B. For the earliest iteration of this project, the buffer size was set to 10^6 experiences, which is the largest of the three values that Van Gelder (2023) tested. Given that each training episode generates roughly 30 transitions, we would expect the buffer to contain approximately 150 000 transitions after the 5 000-episode training run. Therefore, with $|D| = 10^6$, the buffer never filled up and no experiences were ever discarded, preventing gradual experience updates that stabilize DQN learning. Based on this observation, the smaller buffer sizes of 10^5 and 10^4 were tested. Among these, the smallest value showed a significant improvement in the training stability, and was therefore used for the final version. The batch size B—which Van Gelder (2023) varied between 32, 64 and 128—was fixed at 64 in this paper, without any further experimentation.

Although Van Gelder (2023) did not vary the exploration decay factor $\Delta\epsilon$ during his experiments, our project briefly explored its influence on the training behavior, hoping to improve the stability further. The value of the parameter was initially set to 0.001 to ensure thorough exploration over 990 episodes. However, the prolonged exploration phase made the training very unstable, with almost no runs converging to a positive reward. Halving the length of the exploration to 495 episodes, corresponding to $\Delta\epsilon = 0.002$, resulted in much greater success rates. Reducing the exploration length further did not bring any additional improvements, so the value $\Delta\epsilon = 0.002$ from Van Gelder (2023) was left unchanged.

4.3 Industry implications

With the possible alternative architectures covered, we will now use the aforementioned results and descriptions of agent's behavior to draw conclusions about the broader implications of the project for the domain of air traffic control, and aviation as a whole.

The main objective was to select a representative DRL algorithm, and demonstrate its potential for weather avoidance in aviation. For comparability with related research, the project adopted the DQN agent architecture from Van Gelder (2023), originally used for conflict resolution in air traffic control. Adapting the agent for use in weather avoidance involved making a few changes to the original architecture, which have been covered in Section 2.

The following conclusions will be drawn solely from the behavior of the well-trained trained agents, since agents that failed to achieve high rewards should never be used outside of testing. As presented in Section 3.2, the well-trained agents successfully automate the process of avoiding weather, keeping conflicts to a minimum. In most runs, they follow realistic-looking flight paths without unnecessary turns, within the limitations discussed in Section 4.1. However, even the well-trained agents occasionally enter the 5-kilometer safe distance limit d_{lim}, although actual weather encounters are very rare. Due to these conflicts and the aforementioned limitations, the agents are understandably not ready to handle real-world traffic. That said, a completely flawless performance was never the goal of the project, given its intended role as a feasibility study.

From an initial evaluation perspective, the conclusions clearly advocate for further research in this domain, aimed at addressing the drawbacks discussed throughout Section 4. While Van Gelder (2023) originally used the agent architecture for conflict resolution, our project demonstrated its suitability for weather avoidance as well. These results highlight the potential for its use in a future air traffic control system, which would unify both domains. Section 4.4 outlines specific features and changes for future research to focus on, to ultimately make a system that could be certified by the relevant authorities for use in real-world traffic.

It should be emphasized that the presented conclusions concern only the selected DQN algorithm and might not apply to the field of DRL as a whole. While the rationale behind the choice of DQN as a representative example of a DRL algorithm was explained and justified, future projects should aim to verify the suitability of DQN by performing a comparative analysis between various DRL algorithms.

4.4 Future work

The limitations presented above provide extensive opportunities for future research. Possibilities range from improving specific weak points in the current architecture, all the way to making the system suitable for handling real-world traffic.

The downsides of the current approach are directly liked to the design choices presented in Section 2.2. Among these, the simplest to eliminate would be design choice 2, which could be done by applying the same implementation to several different flight paths. This could involve using the same routes both for training and evaluation, as well as training the agent on one set of routes and evaluating it on another. The latter could be used to assess the extent to which our agent learned route-specific behaviors, indicating whether they were successfully kept under control.

A second design choice that could be addressed without adjusting the project's architecture significantly is design choice 5. As discussed in Section 4.2.1, the performance did not noticeably worsen when wind information was removed from the state space. To confirm this observation, one could vary the wind speed and direction organically over the course of each run to see whether these changes impact the performance. In the current version of the agent, other features are more relevant for describing the environment than the wind information. However, larger-scale future projects will likely benefit from re-introducing these features to the state space, as it will allow for finer improvements to the agent's behavior.

Compensating for the remaining three design choices would generally require a more substantial overhaul of the architecture, and addressing them individually would not be advised, due to significant overlap in the necessary changes. The first step would be making the agent navigate the entirety of the pre-set flight path without the help of an autopilot, eliminating design choice 4. Implementing this modification would necessitate expanding the state and action spaces, giving the agent a more detailed overview of the situation and finer control through additional actions.

To allow the agent to properly understand the consequences of its behavior, the system of evaluating its actions individually would have to be adjusted, or removed altogether. Instead, the agent's performance would be assessed based on the overall outcome of each run, such as the total number of weather conflicts or the time taken to reach the final waypoint. For even more precise feedback, the current system could be combined with the proposed per-run system—rewards could be given for individual actions, and then retrospectively combined with a reward for the whole run, before adding all transitions from the run to the agent's buffer. To make the per-run feedback useful, the aforementioned expansion of the state space would have to be sufficiently detailed, so that the agent could learn the factors leading to specific rewards.

Eliminating the last two design choices would be an important step for moving the project toward being ready for industry adaptation. By adding actions allowing the agent to adjust its true airspeed and altitude, design choice 3 would no longer apply. The environment would become three-dimensional, making the whole system much more realistic. Naturally, further changes to the state space would be necessary to provide the agent with information about the quantities it would now be able to control.

With the removal of design choice 1, the project's scope would expand beyond its current domain. On top of being responsible for weather avoidance, the agent would also need to prevent collisions between the individual aircraft. In the area of conflict resolution, significant amount of DRL research is already underway, as discussed in Section 1.1. By combining the two domains, the agent would be able to solve a variety of situations that could arise in air traffic control.

From here, the possibilities for further expansion are virtually limitless. One option might be to implement different types of adverse weather, categorized by how undesirable they are for the agent to fly through, representing storms of varying strengths. Alternatively, with more weather information added to the state space in future iterations, a full predictive weather model could become part of the decision system of the agent. This would combine the project's findings with yet another aviation-related application of DRL, helping to exchange knowledge between different domains.

Throughout the process of implementing the aforementioned changes, an effort should also be made to further improve training stability. For a small-scale feasibility study, only a few well-trained agents are needed to evaluate their behavior, making the lower stability more acceptable. However, for a system used for real-world traffic, the required training success rate would likely be much higher. Given the scale of the aviation industry, even a small improvement in the percentage of successfully trained agents would result in significant global reductions in training time and electricity consumption. The stability could improve naturally with some of the changes discussed in this subsection, or as a result of utilizing a different DRL algorithm, as suggested in Section 4.3.

Ultimately, the goal of any future efforts building on this project should be to integrate it with findings from the aforementioned areas, creating a universal method for automating all aspects of air traffic control. The resulting system should aim to maximize operational efficiency, while adhering to the rigorous safety standards that the public has come to expect from the aviation industry.

5 Conclusion

The study aimed to demonstrate the feasibility of using Deep Reinforcement Learning to automate navigating aircraft around adverse weather. Selecting the Deep Q-Network algorithm as a representative example of DRL, the project adapted the agent architecture which Van Gelder (2023) originally used for conflict resolution in air traffic control. In the process, the architecture was modified in order to accommodate the new domain.

To achieve the goal of the project without making the implementation overly complex, five design choices were introduced to simplify the agent's environment. These included simulating only a single aircraft at a time, flying a pre-set route which remained unchanged across all runs. Additionally, the aircraft maintained a constant altitude and true airspeed, limiting the action space to lateral maneuvers. The agent took control of the aircraft only when weather was detected in its state space, with the rest of the route flown by the autopilot, keeping the aircraft on the predefined flight path. Lastly, to make the environment more dynamic through weather movement, the implementation included wind, which was kept constant throughout the duration of each simulation run, and across the entire map.

Following these design choices, a custom weather abstraction was created on top of the BlueSky air traffic simulator to compensate for its limited weather functionality. The environment was discretized into a grid of boolean squares, each 1 km in size. This allowed for straightforward implementation of the weather movement through shifting the whole grid, and changing the thunderstorm shape organically using cellular automata.

Throughout the implementation testing, the system displayed considerable sensitivity to changes in the training parameters, making it challenging to achieve convergence to high positive rewards. With the final parameter values presented in the paper, 31 out of 50 agents trained successfully, which was more than sufficient to be able to evaluate their performance.

The successfully trained agents displayed the intended behavior, characterized by evading all weather in vast majority of evaluation runs, as well as avoiding unnecessary turns throughout, keeping the flight path looking realistic. For the agents that did not converge to the high rewards during training, the evaluation runs still demonstrated their ability to avoid the weather in most situations. However, these agents struggled to produce a realistic flight path though minimizing turns. Instead, they flew in what could be described as an offset contour of the weather, failing to obtain the positive reward for keeping the heading constant.

While suboptimal flight paths were occasionally produced even by the well-trained agents, the limitations were justified as direct consequences of the design choices, made with respect to the intended scope of the project. These also act as a list of topics that future studies could build on—some being addressable individually, and others requiring a more substantial overhaul of the architecture. Additionally, future work should also focus on improving the training stability of the system, with one of the suggested options being a comparative analysis between DQN and other DRL algorithms.

In conclusion, the study demonstrated that DRL—specifically the DQN algorithm—offers a promising alternative to conventional approaches, successfully automating weather avoidance in aviation, within the context presented by the project. With appropriate modifications, the agent architecture by Van Gelder (2023) proved suitable for the new domain, providing a basis for developing a system that would unify both weather avoidance and conflict resolution. While the implementation was not intended to handle real-world traffic given the design choices, it presents a good foundation for future research, with extensive options for refining the current approach, or expanding it further. The ultimate goal of these efforts should be to combine the domains of conflict resolution, weather avoidance and others, in order to create a universal system for automating air traffic control.

References

AirTrafficSim: An Open-Source Air Traffic Simulation Environment. (2023). Retrieved 2025-03-12, from `https://github.com/HKUST-OCTAD-LAB/AirTrafficSim` (GitHub repository)

BlueSky: Open Air Traffic Simulator. (2016). Retrieved 2025-03-12, from `https://github.com/TUDelft-CNS-ATM/bluesky` (GitHub repository)

Brittain, M., & Wei, P. (2019). Autonomous Separation Assurance in An High-Density En Route Sector: A Deep Multi-Agent Reinforcement Learning Approach. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)* (p. 3256-3262). doi: 10.1109/ITSC.2019.8917217

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv preprint arXiv:1511.07289*. doi: 10.48550/arXiv.1511.07289

Degas, A., Islam, M. R., Hurter, C., Barua, S., Rahman, H., Poudel, M., ... Aricó, P. (2022). A Survey on Artificial Intelligence (AI) and eXplainable AI in Air Traffic Management: Current Trends and Development with Future Research Trajectory. *Applied Sciences*, *12*(3). doi: 10.3390/app12031295

GADM. (2022). *Netherlands.* Retrieved 2025-03-26, from `https://gadm.org/`

Hoekstra, J. M., & Ellerbroek, J. (2016). Bluesky ATC Simulator Project: An Open Data and Open Source Approach. In *7th International Conference on Research in Air Transportation.*

Hui, K. Y., Nguyen, C. H., Lui, G. N., & Liem, R. P. (2023). AirTrafficSim: An open-source web-based air traffic simulation platform. *Journal of Open Source Software*, *8*(86), 4916. doi: 10.21105/joss.04916

ICAO. (2016). *Doc 4444, Procedures for Air Navigation Services — Air Traffic Management.* International Civil Aviation Organization.

Kalidas, A. P., Joshua, C. J., Md, A. Q., Basheer, S., Mohan, S., & Sakri, S. (2023). Deep Reinforcement Learning for Vision-Based Navigation of UAVs in Avoiding Stationary and Mobile Obstacles. *Drones*, *7*(4). doi: 10.3390/drones7040245

Kuchar, J., & Yang, L. (2000). A Review of Conflict Detection and Resolution Modeling Methods. *IEEE Transactions on Intelligent Transportation Systems*, *1*(4), 179-189. doi: 10.1109/6979.898217

Love, J., Chan, W., & Lee, C. (2009). Analysis of Automated Aircraft Conflict Resolution and Weather Avoidance. In *9th AIAA Aviation Technology, Integration, and Operations Conference (ATIO)* (p. 6995). doi: 10.2514/6.2009-6995

LVNL. (2025). *Aeronautical Information Services Publications.* Retrieved 2025-03-21, from `https://lvnl.nl/diensten/aip`

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL).* Retrieved 2025-05-01, from `https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf`

McCabe, K. (2022). *Jet stream and stormy weather.* Royal Meteorological Society. Retrieved 2025-05-20, from `https://www.rmets.org/metmatters/jet-stream-and-stormy-weather`

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, *518*, 529–533. doi: 10.1038/nature14236

Obando-Ceron, J. S., & Castro, P. S. (2021). Revisiting Rainbow: Promoting more Insightful and Inclusive Deep Reinforcement Learning Research. In *Proceedings of the 38th International Conference on Machine Learning* (Vol. 139, pp. 1373–1383). PMLR.

OpenStreetMap contributors. (2025). *OpenStreetMap.* Retrieved 2025-03-26, from `https://openstreetmap.org/`

Ribeiro, M., Ellerbroek, J., & Hoekstra, J. (2020). Review of Conflict Resolution Methods for Manned and Unmanned Aviation. *Aerospace*, *7*(6). doi: 10.3390/aerospace7060079

Rybchak, Z., & Kopylets, M. (2024). Comparative Analysis of DQN and PPO Algorithms in UAV Obstacle Avoidance 2D Simulation. In *International Conference on Computational Linguistics and Intelligent Systems.* doi: 10.31110/colins/2024-3/025

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *CoRR*, *abs/1707.06347*. doi: 10.48550/arXiv.1707.06347

skeyes. (2025). *Aeronautical Information Publication.* Retrieved 2025-03-21, from `https://ops.skeyes.be/`

Van Gelder, J. (2023). *A Centralized, Deep Q-Learning Based Approach for Conflict Resolution in Air Traffic Control for Arrival Traffic* (Master's thesis, University of Groningen). Retrieved 2024-08-28, from `https://fse.studenttheses.ub.rug.nl/id/eprint/31166`

Van Otterlo, M., & Wiering, M. (2012). Reinforcement Learning and Markov Decision Processes. In *Reinforcement Learning: State-of-the-Art* (pp. 3–42). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-27645-3_1

Wang, Z., Pan, W., Li, H., Wang, X., & Zuo, Q. (2022). Review of Deep Reinforcement Learning Approaches for Conflict Resolution in Air Traffic Control. *Aerospace*, *9*(6). doi: 10.3390/aerospace9060294

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards* (Doctoral dissertation, University of Cambridge). Retrieved 2025-04-02, from `https://www.cs.rhul.ac.uk/~chrisw/thesis.html`

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292. doi: 10.1007/BF00992698