Autonomous Conflict Resolution for High-Density Urban Operations Using Deep Reinforcement Learning

Ricardo Santana

TU Delft

Autonomous Conflict Resolution for High-Density Urban Operations Using Deep Reinforcement Learning

by

Ricardo Santana

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday, April 3 at 13:00.

Student number:	5369525	
Project duration:	May 2022 – April 2023	
Thesis committee:	Dr. B.F. Lopes Dos Santos	TU Delft, Chair
	Dr. O.A. Sharpans'kykh	TU Delft, Supervisor
	Dr.ir. E. van Kampen	TU Delft, Examiner

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Acknowledgements

They say that running a marathon can teach you everything you need to know about yourself. I would say something similar for writing a dissertation. There are many ups and downs. Sometimes you feel like you are on top of the world because you had a great idea, and others, you feel devastated to find out it does not end up working the way you conceived it. The only way to successfully cross the line of such a demanding and long work is to surround yourself with bright, supportive people, who are there every time you need them. I want to take the chance to appreciate those people. To my supervisor, Professor Alexei, for giving me the opportunity to work with him and for all the guidance during the course of this project. To Dr. Peng and Ph.D. candidates Lasse and Álvaro for helping me navigate the vast technical field I set out to work in. To my parents, my sister, and my grandmother, who were always there to support me and believe in my success. To the friends I met in Delft, with whom I had unforgettable moments, and to the friends back in Portugal, who made the experience of studying remotely much more enjoyable. Did I learn everything I want to know about myself during the past year? Certainly not, but I did learn many valuable lessons, that will shape the person and the professional I am about to become. I feel grateful to be closing such an important chapter in my life and I can not wait to see what the future holds.

Ricardo Santana
Coimbra, March 2023

[1]

[1] Front cover image credits: NASA Graphics/Alex Gulino

Contents

List of Figures

List of Tables

List of Abbreviations

AAC	Advanced Airspace Concept
AC	Actor Critic
AI	Artificial Intelligence
ANN	Artificial Neural Network
ATC	Air Traffic Control
ATCos	Air Traffic Control Operations
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
CTCE	Centralized Training Centralized Execution
CTDE	Centralized Training Decentralized Execution
CV	Coefficient of Variation
DAA	Detect and Avoid
DD-MARL	Deep Distributed Multi-Agent Reinforcement Learning
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q Network
DDRQN	Deep Distributed Recurrent Q-Network
DL	Deep Learning
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
DRPIQN	Deep Recurrent Policy Inference Q-Network
DTDE	Decentralized Training Decentralized Execution
FAA	Federal Aviation Administration
FL	Flight Level
GAE	Generalized Advantage Estimation
ICAO	International Civil Aviation Organization
KL	KullbackLeibler
LOS	Line of Sight
LSTM	Long Short-Term Memory
LTE	Long-Term Evolution
MADDPG	Multi-Agent Deep Deterministic Policy Gradient

MAPPO	Multi-Agent Proximal Policy Optimization
MC	Monte Carlo
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
NM	Nautical Miles
NN	Neural Network
O-D	Origin-Destination
PER	Prioritized Experience Replay
PPO	Proximal Policy Optimization
R-MADDPG	Recurrent Multi-Agent Deep Deterministic Policy Gradient
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SAA	Sense and Avoid
SARSA	State-Action-Reward-State-Action
SB3	Stable-Baselines3
SGD	Stochastic Gradient Descent
SSD	Solution Space Diagram
TD	Temporal Difference
TD3	Twin Delayed Deep Deterministic Policy Gradient
TRPO	Trust Region Policy Optimization
TSAFE	Tactical Separation Assured Flight Environment
UAM	Urban Air Mobility
UAS	Unmanned Aerial Systems
UTM	Urban Air Mobility Traffic Management
VDN	Value Decomposition Networks
VLL	Very Low-Level
VTOL	Vertical Take-off and Landing

Introduction

During my master's degree at the TU Delft, I had the opportunity to develop my interest and knowledge in the field of Artificial Intelligence. I knew this technology would make great contributions to the future and I wanted to understand it. I took as many classes in the subject as I could, and started having ideas to build my own AI projects. I tried many different things, but some of the most worth mentioning was a rip current detector and a sign language alphabet translator, both based on machine learning algorithms. After taking my internship at an AI startup building autonomous stores, I was seeking to pursue my academic thesis on the topic. This was when I contacted Professor Alexei, who I had first met during the agent-based modeling course I enjoyed. Unexpectedly, the Professor had been following my work online and decided to give me the chance to work with him on a project to apply Reinforcement Learning to Urban Air Mobility Operations.

Urban Air Mobility has been around for a while, with concepts such as air taxis or drone deliveries making newspaper headlines. However, there are multiple challenges to taking this idea to life. After discussions with Professor Alexei, we concluded that interesting work could be developed along the line of automating a centralized Air Traffic Controller, to be able to scale it to what is expected to be a crowded Urban Airspace. The challenge was launched, and after some intense months of work, I could not be prouder of the results. Since I was a little boy, I have been passionate about science, and I am immensely grateful for the opportunity to make my first contribution to a future I believe in.

This thesis report is organized as follows: In Part I, the scientific paper is presented, containing the main literature supporting this research, the problem statement, the solution approach used, and the results. Part II contains the Literature Study work done to initiate this research before the development of this project. Finally, in Part III, some background context is provided to the reader and relevant information to support the paper is added.

I

Scientific Paper

Autonomous Conflict Resolution for High-Density Urban Operations Using Deep Reinforcement Learning

Ricardo Santana,*

Delft University of Technology, Delft, The Netherlands

Abstract

Low-altitude, high-density air traffic is expected to grow in the coming decades with several companies being certified to initiate urban operations for both freight and passenger transport. However, traditional human-centered Air Traffic Control operations (ATCos) are not scalable to handle the increased demand to maintain safe separation between aircraft. Thus, new autonomous solutions to resolve potential conflicts between aircraft must be developed, allowing humans to supervise and understand machine actions. We propose a centralized Deep Reinforcement Learning (DRL)-based framework to provide speed advisories to vehicles, ensuring safe and efficient operations. We used Proximal Policy Optimization (PPO) to train the intelligent controller and show that our framework is capable of handling challenging merging point settings. We evaluated our model extensively using a custom OpenAI Gym environment and proved that it can achieve a 99% success rate in conflict resolution across multiple merging point configurations.

1 Introduction

Urban Air Mobility (UAM) is a futuristic airspace concept, where goods and passengers are transported by a network of aerial vehicles with a high level of autonomy. One key research field for the success of urban operations is the development of appropriate infrastructure, both in terms of ground infrastructure and UAM traffic control. Regarding ground infrastructure, vertiports are considered the requirement for vertical take-off and landing of UAM aircraft. Considering UAM traffic control, the drone market is expected to grow, with companies such as Matternet, focusing on drone deliveries of medical supplies, and Joby Aviation, developing electrical aircraft for aerial commutes, being approved by the Federal Aviation Administration (FAA) to initiate UAM operations [Matternet, 2022, Joby, 2022, Straubinger et al., 2020]. Traditionally, human Air Traffic controllers adopt one or a combination of three strategies to resolve en-route conflicts, occurring when two aircraft lose a minimum vertical or horizontal separation distance: altitude adjustments, heading adjustments, and speed adjustments [Wang et al., 2022]. However, existing Air Traffic Control (ATC) cannot scale to handle high-density operations. Therefore, it is essential to develop a system to resolve conflicts between aircraft under these high-demand circumstances, by either adopting a centralized UAM Traffic Management (UTM) provider or a decentralized de-confliction at the vehicle level [Straubinger et al., 2020].

The ability to accommodate higher air traffic density and automatically resolve conflicts has been quested for many years. Air Traffic Controllers already use automated conflict detection tools. However, when it comes to the resolution procedures, system maturity levels needed for operational deployment have not yet been attained [Erzberger, 2005]. Heinz Erzberger and his colleagues at NASA Ames Research Center proposed the Advanced Airspace Concept (AAC), an autonomous Air Traffic Control architecture that relies on two safety layers - the autoresolver and the TSAFE (Tactical Separation Assured Flight Environment). The autoresolver generates resolution trajectories that are transferred to aircraft autonomously, while the TSAFE acts as a backup for imminent conflict [Erzberger, 2004, Erzberger, 2005]. In more recent studies considering Unmanned Aerial Systems (UAS) for low-altitude operations, the requirement for an automated traffic flow control solution in structured airspace becomes evident due to the high-throughput, unpredictable, and flexible nature of autonomous operations [Jang et al., 2017, Hunter and Wei, 2019].

Artificial Intelligence (AI) techniques have been rising in popularity over the past years. Specifically, Reinforcement Learning (RL) has successfully achieved super-human performance in several sequential games. In 1997, supercomputer IBM Blue beat the world champion in the game of Chess [Campbell et al., 2002]. In 2013, Deepmind's scientists proposed a novel method capable of outperforming humans in classical Atari games [Mnih et al., 2013]. More impressively, in 2016, the same company developed AlphaGo, an AI algorithm capable of beating the strongest Go players in the world [Silver et al., 2016]. The premise of RL is to find optimal policies or sequences of actions without prior knowledge of an environment by interacting and collecting rewards, or

*MSc Student, Air Transport Operations, Faculty of Aerospace Engineering, Delft University of Technology

trial-and-error. RL methods can be divided into value-based and policy-gradient. Value-based methods typically estimate an action-value function, i.e., the long-term expected cumulative reward given the current state of the environment and each possible action the agent can take. On the other hand, policy-gradient methods directly learn a parameterized policy, i.e., the probability of selecting a given action given the current state of the environment. The combination of RL and Neural Networks (NNs) is called Deep Reinforcement Learning (DRL) and is able to handle environments with continuous observations by using NNs as function approximators to select the best action at each step.

DRL algorithms present a very suitable solution for the autonomous ATC conflict resolution problem due to the sequential decision-making power and the capability of learning without prior environment information of RL, and the generalization capabilities of NNs, resulting in the agent's capability to adapt to previously unseen situations [Wang et al., 2022]. DRL has been used in recent works on en-route conflict resolution. Marc Brittain and his colleagues developed a line of work focusing on en-route deconfliction using speed adjustments only. In [Brittain and Wei, 2018] a centralized agent is proposed to select both the optimal route and speed for two aircraft to cross NASA's Sector 33 environment. In [Brittain and Wei, 2019, Brittain et al., 2020, Brittain and Wei, 2021] the concept is expanded to a multi-agent distributed setting to operate a larger number of aircraft over a BlueSky intersection sector. Although the latest methods achieve very high performance, it can become difficult for humans to understand and supervise a distributed system. Moreover, as suggested in [Jang et al., 2017] NASA's urban airspace operational concept, urban traffic flow operations should be regulated by a system responsible for supervising and controlling traffic flow, broadcasting traffic information, and detecting and communicating the presence of unauthorized flights. Therefore, the goal of this research is to bridge the gap between the ease of adoption of a centralized controller and the scalability achieved by vehicle-level deconfliction methods, by designing a centralized and scalable conflict resolution approach based on DRL for UAM.

The main contributions of this paper are proposing and evaluating a centralized framework for low-altitude en-route conflict resolution in high-density stochastic merging point traffic settings. A merging point refers to an aerial sector configuration in which multiple entry routes merge into a single exit route. The system is trained in stochastic settings, i.e., the relative position between entry routes is randomized, to make the agent deployable in multiple locations. The framework proposed resembles existing ATC operations. At every step, each aircraft communicates its position and speed to the centralized controller. Based on the observation of the current state of the environment, the autonomous agent selects a speed adjustment for one aircraft at most. The process is repeated until every aircraft has exited the sector successfully or a collision is imminent, in which case a redundant safety system (out of the scope of this research) would have to be activated to avoid a collision.

This paper is structured as follows. Section 2 introduces the main DRL theory, including value-based, policy-gradient, and actor-critic methods. Furthermore, it describes the current state of ATC, some important considerations when considering UAS, and the main applications of DRL methods to the en-route conflict resolution problem with speed advisories. Section 3 thoroughly describes the model developed, both in terms of the environment and learning framework. Namely, it covers the learning environment, observation and action spaces, reward function, and learning algorithm. Section 4 defines the experiments to be conducted on the trained model to assess the performance of the agent under different environment conditions, such as variable aircraft arrival settings, sector configurations, and communication blockage situations. Section 5 shows training metrics, such as the evolution of the average reward, episode length, and success rate. Moreover, it presents the results of the experiments defined in the previous section. The performance metrics used, such as the likelihood of a collision, the average time to cross the sector, and the average distribution of actions over a route, are evaluated on these different experiment scenarios. Section 6 summarizes the work done and relevant results obtained. Finally, Section 7 discusses the main limitations of this work, as well as relevant lines for future research to be conducted.

2 Literature Review

This research is developed at the intersection between two fields: DRL and ATC. In this section, state-of-the-art model-free DRL theory is addressed, as well as previous applications to en-route conflict resolution.

2.1 Reinforcement Learning

Reinforcement Learning is a sub-field of AI concerned with how an agent can learn new behavior through trial-and-error interaction with a previously unseen environment [Kaelbling et al., 1996]. In typical supervised learning settings, an AI model learns using input-output pairs. In Reinforcement Learning, the agent explores the environment and the only external guidance it receives comes in the form of rewards. The single learning agent RL problem can be described as a Markov Decision Process (MDP), characterized by the tuple $< X, U, f, \rho >$, where X is the state space, describing the current state of the environment, U is the action space, the set of all possible actions an agent can take, f is the transition probability function, the probability that the agent

transitions from one state to another given an action choice, and ρ is the reward function, the incentive given according to the agent's goals. For every time step k in an episode, the agent observes the current state $x_k \in X$ and takes an action $u_k \in U$. Given the current state of the environment, the action selected, the transition probability function, and the reward function, the environment returns the next state x_{k+1} and a scalar reward r_{k+1} indicating how well the agent performed [Buşoniu et al., 2010]. During the learning process, the agent updates its policy (π), the method used to choose which possible action to take, to maximize the expected cumulative reward [Lonza, 2019].

2.1.1 Value-Based Methods

Value-based RL methods rely on learning an action-value function that expresses the long-term quality of an action given the current observed state. The discounted sum of rewards over a trajectory (τ), or sequence of state-action pairs, is called discounted return (R) [Lonza, 2019]:

$$R = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \tag{1}$$

Where γ^k, the discount factor at time step k sets how much importance an agent should attribute to future rewards, in relation to immediate rewards. The value function (V_π) is simply the expected discounted return or the long-term quality of a state following a policy. Similar to the value function, the action-value function (Q_π), or Q-function, expresses the expected return given not only the current state but also the action selected by the agent [Lonza, 2019]:

$$Q_\pi(x, u) = \mathbb{E}_\pi[R|x_0 = x, u_0 = u] \tag{2}$$

While one can run full trajectories and average the return results to estimate V_π and Q_π, this is computationally expensive. Methods that use this technique to estimate the return are called Monte Carlo. A less expensive way is to use the Bellman equation as it enables the estimation of these functions recursively, from subsequent states. Equation 2 can be adapted using the Bellman equation [Lonza, 2019]:

$$Q_\pi(x, u) = \mathbb{E}_\pi[r_k + \gamma Q_\pi(x_{t+1}, u_{t+1})|x_k = x, u_k = u] \tag{3}$$

Using the immediate reward and the next state value to compute the value for the current state is called bootstrapping. RL methods that learn online (the agent is improving by collecting new data from the environment) using bootstrapping are called TD (temporal difference). While TD methods are generally faster and have lower variance than Monte Carlo, they suffer from a high bias problem [Lonza, 2019]. Off-policy (the policy being updated - target policy - and the policy acting on the environment - behavior policy - are different) Q-learning was proven convergence in [Dayan and Watkins, 1992]. This method estimates the Q-function by iteratively updating the Q-values:

$$Q(x_k, a_k) \leftarrow Q(x_k, a_k) + \alpha_k \cdot [r_{k+1} + \gamma max_{u'} Q(x_{k+1}, u') - Q(x_k, u_k)] \tag{4}$$

Using a learning rate α_k and a TD-error. The apostrophe refers to the next step. The training process of Q-learning uses a tabular approach to store the Q values and explores the environment choosing between random and greedy (maximum Q value) actions with an exploration probability ϵ. The problem with this tabular approach is that it can only consider discrete state-action pairs, meaning the observation values must be quantized before learning. [Mnih et al., 2013, Mnih et al., 2015] proposed Deep Q Network (DQN), a combination between Q-learning and NNs that can handle continuous state spaces. The architecture uses a NN that takes the observations as inputs and directly predicts the Q-values for all possible actions. They introduced two concepts to stabilize the learning process: an experience replay and a target network. The experience replay stores experience samples $(x_k, u_k, r_{k+1}, x_{k+1})$ and used to train an online network. A target network is used to predict the target values for the loss function in the online network update. Every C steps, the weights are transferred from the online to the target network. The online network is updated using the following loss function:

$$L(\theta) = \mathbb{E}_{(x,u,r,x') \sim U(D)}[(r + \gamma max_{u'} Q(x', u'; \theta^-) - Q(x, u; \theta))^2] \tag{5}$$

Where θ are the parameters of the online network and θ^- are the parameters of the target network. The network is updated using minibatches $(x, u, r, x') \sim U(D)$ drawn uniformly at random from the experience replay memory [Mnih et al., 2015].

2.1.2 Policy-Gradient and Actor-Critic Methods

Instead of learning an action-value function to find a good policy, policy-gradient methods directly parameterize the policy (π_θ), in order to maximize the expected return ($\mathbb{E}[R|\theta]$) with respect to the function approximator parameters θ. This policy outputs a probability value for each action in discrete action settings. As seen before, the goal of an agent is to maximize the cumulative reward over a trajectory $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$. The policy gradient theorem gives the derivative of the objective function J that can be used to update the parameters [Lonza, 2019]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(u|x)Q_{\pi_\theta}(x,u)] \tag{6}$$

On-policy (the policy being updated - target policy - and the policy acting on the environment - behavior policy - are the same) REINFORCE was introduced in [Williams, 1992] and uses Monte Carlo (MC) returns to estimate the return R (to substitute the term $Q_{\pi_\theta}(x,u)$ in Equation 6) by summing the discounted future rewards. The expectation is removed from Equation 6 by sampling from the stochastic policy and averaging the results. As this architecture uses MC returns on complete trajectories, it is unbiased. However, this comes at the cost of high variance (because the policy is stochastic, and re-running it may lead to different rewards) and slower learning (because MC estimates rely on full trajectories). An estimation of the value function is often subtracted as a baseline in the estimation of the return to reduce variance [Lonza, 2019]. However, these disadvantages are typically overcome by combining the benefits of value-based and policy-gradient methods - the so-called actor-critic methods. Actor-critic (AC) was proposed in [Konda and Tsitsiklis, 1999] and uses two NNs as function approximators: the actor that parameterizes a policy π_θ, and the critic that provides feedback to the actor on how good were actions based on the value function V_w [Haydari and Yilmaz, 2020]. The one-step actor update is obtained by substituting the Q value in Equation 6 with a baseline by the one-step bootstrapping $Q(x,u) = r + \gamma V(x')$, where x' is the next state [Lonza, 2019]:

$$\theta = \theta + \alpha(r_k + \gamma V_w(x_{k+1}) - V_w(x_k))\nabla_\theta log\pi_\theta(a_k|x_k)) \tag{7}$$

To find a trade-off between the benefits of TD and MC methods, AC usually employs an n-step return, rather than a one-step. This means that the advantage estimation in Equation 7 becomes $R_{k:k+n} + \gamma^n V_w(x_{k+n}) - V_w(x_k)$. The critic parameters w are updated using the target values $y_i = R_{k:k+n} + \gamma^n V_w(x_{k+n})$ and a squared error loss function [Lonza, 2019].

Multiple advancements have been suggested to AC methods. A3C (asynchronous advantage actor-critic) proposed in [Mnih et al., 2016] and its synchronous version A2C use multiple actor-learners in parallel to explore the environment, accelerating and stabilizing the learning process. AC methods have a disadvantage: they may suffer from instability in the policy that causes a decline in agent performance. This issue is tackled using a trust region in TRPO (Trust Region Policy Optimization [Schulman et al., 2015a]), or a clipped objective function in PPO (Proximal Policy Optimization [Schulman et al., 2017]). TRPO uses the Kullback-Leibler (KL) divergence function (measurement of how different two policy distributions are) between the new and old policies in an update to create a trust region [Lonza, 2019]:

$$\text{maximize}_\theta \ \ \mathbb{E}_{\pi_{old}}\left[\frac{\pi_\theta(u_k|x_k)}{\pi_{\theta_{old}}(u_k|x_k)}A_k\right] \tag{8}$$

$$\text{subject to } \ \mathbb{E}_{\pi_{old}}[KL[\pi_{\theta_{old}}(\cdot|x_k), \pi_\theta(\cdot|x_k)]] \leq \delta \tag{9}$$

Where A_k is an estimation of the advantage function of the old policy $A_{\pi_{old}}(x,u) = Q_{\pi_{old}}(x,u) - V_{\pi_{old}}(x)$. The surrogate objective function considers data sampled from the old policy π_{old}. $\pi(\cdot|x)$ is the distribution of actions conditioned to the state x [Lonza, 2019, Schulman et al., 2015a]. PPO introduced a novel clipped surrogate objective function to simplify TRPO, making it more general and sample efficient [Schulman et al., 2017]:

$$L^{CLIP}(\theta) = \mathbb{E}_{\pi_{old}}[min(r_k(\theta)A_k, clip(r_k(\theta), 1-\epsilon, 1+\epsilon)A_k] \tag{10}$$

Where r_k is a probability ratio similar to Equation 8 $r_k(\theta) = \frac{\pi_\theta(u_k|x_k)}{\pi_{\theta_{old}}(u_k|x_k)}$, and ϵ is a hyperparameter, typically $\epsilon = 0.2$. The goal of this objective is to penalize policy changes that move r_k away from 1 (i.e. the update being too large). In case the advantage function is positive (meaning the action taken is better than the old policy), there is a positive upper bound to the objective. In case the advantage function is negative, there is a minimum but no maximum negative value to the objective [Schulman et al., 2017]. PPO uses a truncated version of Generalized Advantage Estimation (GAE) proposed in [Schulman et al., 2015b] to estimate the advantage value. Denoting $\delta_k^V = r_k + \gamma V(x_{k+1}) - V(x_k)$ the one-step TD advantage estimate, the $(T-k)$-step estimator for timestep k in a length-T trajectory is:

$$A_k = -V(x_k) + r_k + \gamma r_{k+1} + ... + \gamma^{T-k+1} r_{T-1} + \gamma^{T-k} V(x_T) \tag{11}$$

The truncated version of GAE is obtained as the exponentially-weighted average of estimators of all possible step lengths, where λ is a parameter compromising between bias and variance ($0 < \lambda < 1$) [Schulman et al., 2015b, Schulman et al., 2017]:

$$A_k = \delta_k + (\gamma\lambda)\delta_{k+1} + ... + ... + (\gamma\lambda)^{T-k+1}\delta_{T-1} \tag{12}$$

PPO uses fixed length-T trajectories and N parallel actors to collect data. Therefore, the loss function in Equation 10 is optimized for these NT data samples using Adam (in [Kingma and Ba, 2014]). Finally, a squared error loss term with respect to the critic is added to the loss function to be maximized in Equation 10 if the actor and critic share parameters. An entropy bonus term is also added to ensure exploration [Schulman et al., 2017]. The following pseudo-code describes the algorithm:

Algorithm 1 PPO actor-critic algorithm (adapted from [Schulman et al., 2017])

for iteration $= 1, 2, ...$ **do**
  for actor $= 1, 2, ..., N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates A_1, ..., A_T
  end for
  Optimize surrogate L with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

[Schulman et al., 2017] tested PPO against other state-of-the-art policy-gradient methods on multiple tasks, including continuous control and Atari game playing, proving the superiority of this algorithm in terms of performance and sample complexity.

2.2 Deep Reinforcement Learning for En-Route Conflict Resolution

According to the International Civil Aviation Organization (ICAO), a conflict between two aircraft is defined when the distance between them is lower than a certain threshold. For civil aviation, the values are 5NM horizontally, and 1000ft vertically. In traditional settings human ATC operators use one or a combination of three methods to deconflict aircraft: altitude adjustment uses flight levels (FL) expressed in hundreds of feet to maintain vertical separation; heading adjustment changes the heading direction of an aircraft; lastly, speed adjustment varies aircraft speed in multiples of 10kt. The conflict resolution method to adopt is chosen according to three high-level principles: safety is the number one priority; resolving a conflict should not raise other conflicts; and air traffic disruption and ATCos workload should be minimized [Wang et al., 2022]. A recent study by Hunter and Wei suggests that the solution for urban operations, with particular characteristics such as high autonomy levels, low-reliability communications, dense operations, low-altitude flight, and unpredictability, is to design structured urban airspace with autonomous ATC [Hunter and Wei, 2019].

Planning the trajectories of all vehicles in space and time (4D trajectories) has been proposed as a possible solution (e.g. [Joulia et al., 2016]). However, it does not cope with the unpredictable and on-demand nature of urban operations. A recent line of work by Brittain et al. uses DRL with speed advisories to maintain safe separation between aircraft in challenging en-route aerial sectors.

In [Brittain and Wei, 2018] the authors used Double DQN, a variation of DQN introduced in [Van Hasselt et al., 2016] that decouples the target value in Equation 5 to tackle the overestimation problem. It uses the online network to choose the action according to the greedy policy and the target network to compute its value. [Brittain and Wei, 2018] use NASA's Sector 33 environment and a hierarchical RL approach to select the optimal route and speed for N aircraft to get to their destinations. An episode is terminated if every aircraft reaches its goal position, at least one aircraft got out of time, or two aircraft collided, meaning that the euclidean distance between them is lower than a pre-defined threshold. The hierarchical RL approach involves two agents: a parent and a child. The parent agent processes the raw image of the environment using a Convolutional Neural Network (CNN) and chooses a route combination for the aircraft to follow at the beginning of every episode. On the other hand, the child agent observes the aircraft coordinates, speed, and the route chosen by the parent to decide on one of 6^N possible speed combinations for the N aircraft at every step (time between steps is 4 seconds of simulation). The reward function at each time step is two-fold: the parent agent is rewarded based on the inverse of the distance between each aircraft and their goal position, while the child agent is rewarded based on the speed of each aircraft for efficiency. Thus, longer routes and lower aircraft speeds are penalized. Additional one-time rewards are added when an episode finishes: -10 for collision, -3 for out-of-time, and +10 for optimal solution [Brittain and Wei, 2018].

$$r_p = \frac{1}{\sum_{i=1}^{N} |g_{x_i} - x_i|} \tag{13}$$

$$r_c = 0.001 \sum_{i=1}^{N} v_i - 0.6 \tag{14}$$

Where x_i represents the current position of aircraft i, g_{x_i} is the goal position, and v_i is the current speed [Brittain and Wei, 2018]. In [Brittain and Wei, 2019] the authors adopted a decentralized centralized training decentralized execution (CTDE) multi-agent approach to the separation assurance problem using the BlueSky simulator. Instead of a centralized controller, they consider each aircraft as a decision-making agent, containing its own NN, whose parameters are shared between every aircraft, trained using A2C with the loss function from PPO and parameter sharing between actor and critic. The environment simulates a sector containing an intersection or merging point configuration, where N aircraft enter according to a uniform distribution. The terminal state is achieved once every aircraft has exited the sector. To fix the observation space length, it contains information about the ownship, as well as the N-closest aircraft that may cause a conflict: agents on the same route or agents on a conflicting route who have not yet reached the intersection. The information about the ownship contains distance to the goal, speed, acceleration, distance to the intersection, route identifier, and the loss of separation distance. The information for each intruder contains distance to the goal, speed, acceleration, distance to the intersection, route identifier, and distance from the ownship to the intruder. Based on this observation, each agent takes action independently by selecting one out of three possible speed values every time step (the time between steps is 12 seconds). The reward function for the multi-agent system penalizes only the agents involved in a conflict:

$$r_k = \begin{cases} -1, & \text{if } d_o^c < 3NM \\ -\alpha + \beta \cdot d_o^c, & \text{if } 3NM \leq d_o^c < 10NM \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Where d_o^c is the distance between the ownship and the closest aircraft, and α and β are constants to penalize agents for approaching the loss of separation distance ($d^{LOS} = 3NM$). In [Brittain and Wei, 2021] the authors improved the previous multi-agent method by using a long short-term memory (LSTM) network (proposed in [Hochreiter and Schmidhuber, 1997]) to handle the fixed-size observation space problem (the N-closest agents may not always be the most relevant to consider in the observation space) by encoding a variable number of intruder aircraft sorted based on the distance to the ownship (in descending order for relevance) into a fixed-length vector and PPO to train the multi-agent architecture. The information observed and rules to determine conflicting aircraft are similar to previous work. A new action space is proposed so that an agent can either decelerate, hold the current speed or accelerate. The reward function used is the same as in Equation 15 [Brittain and Wei, 2021]. Finally, in [Brittain et al., 2020] the authors provide two more contributions. First, substituting the LSTM with an attention mechanism (introduced in [Bahdanau et al., 2014]) to avoid losing important information from the past as the attention mechanism has access to all hidden vectors, as opposed to the LSTM which only accesses the information that has already been propagated through the network. Secondly, adding a constant reward term to minimize the number of speed adjustments (ψ) [Brittain et al., 2020].

$$r_k^u = \begin{cases} 0, & \text{if } u = \text{Hold} \\ -\psi, & \text{otherwise} \end{cases} \tag{16}$$

3 Methodology

This research aims at creating an autonomous ATC system to resolve conflicts in high-density traffic merging point sectors. This type of urban airspace configuration was chosen because it is particularly challenging from a coordination point of view. First, the decision-making agent must ensure safe separation before and after the merging point. Secondly, the agent must coordinate high-density traffic coming from multiple routes in a single exit route (after the merging point) [Brittain and Wei, 2019]. Static airspace configurations have been considered in prior work. Although easier from a learning perspective, they pose additional problems for real-world implementations: an agent trained and tested in a given airspace configuration cannot be safely deployed in a different airspace sector. For that reason, this research aims at training and testing a DRL policy in variable airspace configurations, generating an agent that is agnostic to the angular distance between entry routes (before the merging point).

There is yet another critical aspect to define the scope of this work. As discussed in Section 2, Brittain et al. first used a centralized approach. However, they were constrained in the number of aircraft that could

be controlled because the action space size would grow exponentially with the number of agents. Later, they used a decentralized multi-agent approach, no longer bounded by the size of action space, with the advantage of handling a more significant number of aircraft. To fix the observation space size, they first considered the N-closest agents only, and then used different Deep Learning (DL) techniques to encode this information in a fixed-length vector [Brittain and Wei, 2018, Brittain and Wei, 2019, Brittain and Wei, 2021, Brittain et al., 2020]. Despite the scalability gains, a decentralized approach presents disadvantages to adopting autonomous ATC. First, today's operations are mostly centralized. For instance, Area Control Centers (ACC) manage air traffic flying in areas under their jurisdictions. Secondly, decentralized systems are likely harder to supervise, as there are many moving parts. Therefore, this research aims to develop a centralized but scalable approach to autonomous ATC, designing an observation and action space that scales linearly with the number of aircraft in the sector.

The proposed centralized training centralized execution (CTCE) architecture relies on two main components: the airspace sector environment, containing the simulation settings, the state transition rules, and the reward function reflecting the learning objectives; and the agent, which learns by observing the environment, deciding which action to take based on its current policy, and updating its NN based on the feedback received from the environment. The interaction loop between the agent and the environment during learning is illustrated in Figure 1.

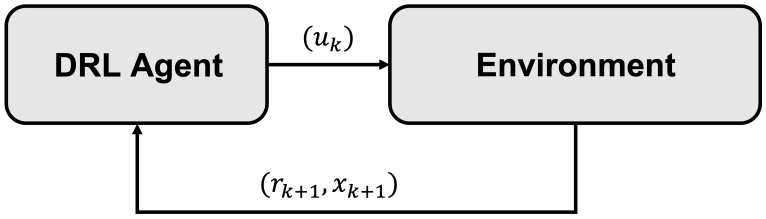


Figure 1: Single-agent DRL architecture for autonomous urban ATC. u_k is the action selected based on the current policy. x_{k+1} and r_{k+1} are the next state and reward provided by the environment.

3.1 Problem Formulation

As mentioned before, the goal of this work is to develop an automated ATC system that can handle high-density urban traffic in challenging merging point configurations. A custom deterministic environment (the next state can be determined based on the current state and action only) was developed in Python using the OpenAI Gym library for RL research and rendered using OpenCV [Brockman et al., 2016, Culjak et al., 2012]. Our controller should be able to prioritize safety, i.e. no collisions in the sector while maximizing efficiency, i.e. minimizing the average time it takes an aircraft to cross the aerial sector. Although aircraft typically travel at economical speeds in commercial aviation, the model proposed considers delay minimization as a higher priority in the context of short-distance urban operations [Wang et al., 2022]. Furthermore, in the scope of this study, a fixed number of aircraft ($|\mathcal{N}|$) crossing the sector per episode is considered.

3.1.1 Objective

The objective of the model proposed is to provide real-time speed advisories to $|N|$ aircraft, coming from multiple entry routes, safely and efficiently merging them into a single exit route. In order to do that, it must ensure that three conditions are met: aircraft do not collide with other aircraft on the same route before or after the merging point; aircraft do not collide with other aircraft in a conflicting entry route near the merging point; and aircraft cross the sector as rapidly as possible given the prior constraints. For simplification purposes, aircraft are considered to fly at the same constant altitude and follow their respective routes. Therefore, a collision is defined when the euclidean distance between two aircraft is lower than a certain threshold (δ_i):

$$\sqrt{(x_n - x_m)^2 + (y_n - y_m)^2} \leq \delta_i, \forall n \neq m; n, m \in \mathcal{N} \tag{17}$$

To define the conflict resolution process, the concepts of inner and outer boundaries are considered. The inner boundary represents a near-collision condition or the minimum distance between two aircraft before the collision. The outer boundary represents the last chance to maneuver the threshold before the inner boundary is violated [Hunter and Wei, 2019]. Thus, in Equation 17, δ_i represents the inner boundary threshold. The model is considered to be successful in an episode if all $|\mathcal{N}|$ aircraft exit the route without collisions, considering g_n the goal position of aircraft n, or the point on the exit route from which the agent no longer has control over the aircraft.

$$|g_{x_n} - x_n| = 0, \forall n \in \mathcal{N} \tag{18}$$

3.1.2 Environment Settings

The custom-designed aerial sector depicted in Figure 2 is 4 km wide by 4 km high. The length of the section to be controlled on each route is 1.8km. The merging point is located in the center of the sector, $(x_i = 2km, y_i = 2km)$. The value for the outer boundary threshold (δ_o) is considered to be directly proportional to the wingspan. Thus, if we consider the conflict threshold for a Boeing 747-8 to be 5NM, with 68m of wingspan, the outer boundary for a Joby S2 electric VTOL (vertical take-off and landing) designed for urban operations, with 9m of wingspan is approximately $\delta_o = 1km$ [Karen Dix-Colony, , Stoll et al., 2014]. The inner boundary is set to be $\delta_i = 75m$, or 7.5% of the outer threshold.

A minimum angular separation of 15° between entry routes is considered for realism. Moreover, the minimum angle between an entry route and the exit route is 90°. Given these values, we can compute the danger radius around the merging point, i.e. the maximum distance from the merging point at which collisions between aircraft in conflicting entry routes may happen. This value is $\frac{75m/2}{sin(15°/2)} \approx 287m$, which means that if two aircraft are both 287m from the merging point at two routes with the minimum angular separation, they collide. The sector configuration is randomized on each episode according to the rules: first, the number of entry routes is drawn at random from a uniform distribution over 2 and 3; secondly, routes are picked at random from the set of all possibilities $(180°/15° + 1)$, bringing the total number of possible routes in the sector to 14 (including exit route).

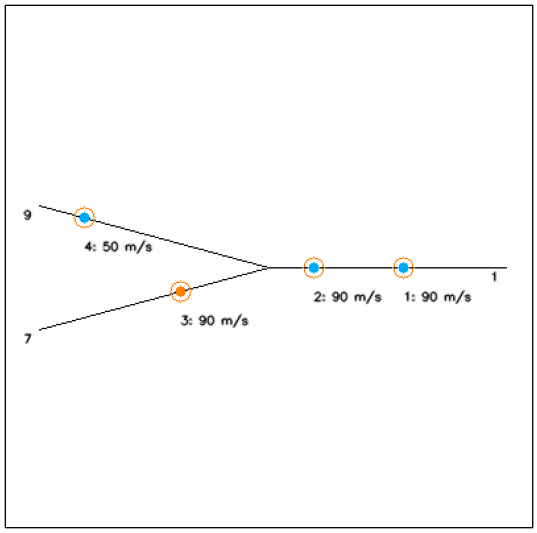


Figure 2: Merging point sector with two entry routes. Aircraft enter the sector through routes 7 or 9 and are merged into route 1 before exiting.

Aircraft arrive at the first point on each entry route (1.8km away from the merging point) according to a Poisson process that is independent per route, meaning two aircraft can arrive at the same time on separate routes. If we consider I^j to be the random variable that represents the inter-arrival time at route j, $I^j \sim Poisson(\lambda)$, where λ is the expectation. Furthermore, it is assumed that aircraft have a constant entry speed of $v_e = 50m/s$. λ is computed such that the expected initial separation between aircraft on the same route equals the outer boundary. To ensure that collisions would not happen before the autonomous controller could take action, a lower boundary of $3s$ was set for I^j.

$$\lambda = \frac{\delta_o}{v_e} = 20s \tag{19}$$

Considering the Joby S2 as a reference once again, the maximum value for the speed envelope is set to be the cruise speed of this vehicle at approximately $90m/s$ [Stoll et al., 2014]. The minimum speed value is set to $30m/s$. Finally, speed variations can only happen in intervals of $\Delta v = 10m/s$. Episode steps in this environment represent a $\Delta t = 1s$ of flight time. Such a high action rate is considered due to the expected autonomy in the context of urban operations [Hunter and Wei, 2019].

3.1.3 Reinforcement Learning Formulation

The interactions between the intelligent controller and the environment must be specified in terms of observation space, or what features of the environment the agent can see; action space, or what types of action can the agent take to change the current state of the environment; terminal state, or the conditions under which an episode is finished; and reward function, or the feedback the agent gets for achieving or getting closer to the

goals set. These different parts make up the RL formulation for our centralized solution. Our agent receives information about the current state of all aircraft in the sector and takes actions to minimize collisions and maximize efficiency while collecting feedback from the system's overall performance.

State Space The state space (also called observation space) should contain all the information the agent requires about the current state of the environment to make an action choice. As the environment represented in Figure 2 only allows speed adjustments, its current state can be fully described with the following information for every aircraft - route identifier, current speed, and position within the current route:

$$X = [r_1, v_1, d_{1,i}, r_2, v_2, d_{2,i}, ..., r_{\mathcal{N}}, v_{\mathcal{N}}, d_{\mathcal{N},i}] \tag{20}$$

Where r_n is the current route identifier or route angle for aircraft n, v_n is the current speed and $d_{n,i}$ is the distance to the intersection point (i) (always positive, regardless of whether the aircraft has already crossed the intersection). The route identifier of the exit route is always 0. The identifier for the entry routes is the angle with the exit route measured counter-clockwise. The distance to the intersection represents the route length between the current aircraft position and the merging point. The navigation data in the observation space could be measured using onboard sensors and transmitted using a communication link (e.g. 5G networks) [Hunter and Wei, 2019]. The size of X scales linearly with the total number of aircraft, as $|X| = 3|\mathcal{N}|$. There are three additional details regarding the implementation of the observation space that are important for learning. First, all the features in Equation 20 are normalized between -1 and 1 before entering the NN. This is a standard technique in DL to improve model performance [Kim, 1999]. This is achieved using the following equation: $f_{\text{normalized}} = \frac{f - f_{min}}{f_{max} - f_{min}} \cdot 2 - 1$, where f is the variable to be normalized. Secondly, as seen in Equation 20, the state space contains information about all aircraft. However, the relative position of each aircraft in this vector is not set at random. Instead, aircraft are ordered according to the arrival time at the sector, or the time they are first observed by the autonomous controller. This sorting technique has proven to stabilize learning, possibly because aircraft that arrive at similar times in conflicting entry routes are more likely to get involved in a conflict. Thus, aircraft are sorted from the earliest to the latest arrival time. Lastly, although we defined a fixed-length state space, the centralized agent is considered to only have access to information about active aircraft, i.e. aircraft that are currently flying over the sector, and those the agent can control. This means that the states respecting aircraft that have not yet entered the sector or that have already left (inactive aircraft) are padded with zeros (a similar zero-padding technique was used in [Vinitsky et al., 2018]).

Action Space The action space should contain all possible action choices to interact with the environment and achieve the required objectives. Centrally adjusting multiple aircraft speeds at a time could involve an exponentially growing action space. Besides, it makes it harder for a human supervisor to understand and follow the system's decisions. Therefore, the discrete action space defined in the scope of this research allows for single aircraft adjustments at a time:

$$U = [None, a_1^-, a_1^+, a_2^-, a_2^+, ..., a_{\mathcal{N}}^-, a_{\mathcal{N}}^+] \tag{21}$$

Where $None$ represents no speed adjustments, a_n^- decreasing aircraft n speed, and a_n^+ increasing its speed. Speed variations are always done by adding or subtracting Δv to the current aircraft speed, limited by lower and higher boundaries as previously defined. The size of the action vector is $|U| = 2|\mathcal{N}| + 1$ and only one of the entries can be 1 at each time step, with all the others equaling 0. As mentioned, not all aircraft are active at some point in an episode. Thus, an invalid action happens when the agent chooses to increase or decrease the speed of an inactive aircraft. If this is the case, the action is filtered, with no speed adjustments taking place.

Terminal State The terminal state is achieved and an episode ends in one of two circumstances: when Equation 17 is met by any pair of active aircraft, i.e. when a collision happens because the inner boundary threshold is violated, or when every aircraft has exited the sector without incidents.

Reward Function The reward function should express the goals of the model: in this case, to minimize the number of collisions and time to cross the intersection. This is done using a combination of four different reward components. To begin with, an episode reward is set for when the terminal state is achieved.

$$r_e(k) = \begin{cases} L, & \text{if all aircraft exit without collisions} \\ -L, & \text{according to (17)} \end{cases} \tag{22}$$

Where L is a constant used to penalize collisions and positively reward successful episode runs. The episode reward is given only once per episode. To provide additional guidance for the agent, dense rewards are also given at every time step, namely a separation reward, a speed reward, and an action reward. The separation reward penalizes the agent when two aircraft approach the inner threshold.

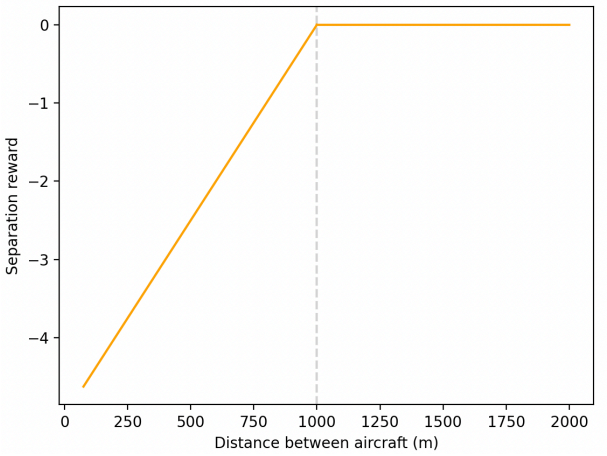


Figure 3: Separation reward per aircraft pair defined in branches ($\delta_o = 1000m$).

Figure 3 depicts the scaled reward attributed to every pairwise combination of active aircraft flying over the sector. It starts penalizing the agent when the distance for any pair of aircraft is lower than the outer boundary and decreases linearly with this distance, penalizing smaller distances the highest. This reward is then averaged over all possible pairs and given to the agent as:

$$r_s(k) = \begin{cases} \frac{1}{|\mathcal{P}|} \sum_{p_{m,n} \in \mathcal{P}} \beta_s \cdot (\delta_o - d_{m,n}), & \text{if } \delta_i \leq d_{m,n} < \delta_o \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

Where $\mathcal{P}$ is the set of all active pairs of aircraft flying over the sector, β_s a small negative constant weight to scale the reward, and $d_{m,n}$ the euclidean distance between aircraft in a given pair $p_{m,n}$. The speed reward is given to encourage higher aircraft speeds, reducing flight time over the sector.

The speed reward given to the agent is directly proportional to the velocity of each aircraft, varying between $30m/s$ and $90m/s$, and averaged over all active aircraft in the sector:

$$r_v(k) = \frac{1}{|\mathcal{A}|} \sum_{n \in \mathcal{A}} \beta_v \cdot v_n(t) \tag{24}$$

Where $\mathcal{A}$ is the set of all active aircraft and β_v is a small positive constant to scale the reward. The reason why the separations and speed rewards are averaged is to provide the agent with feedback on how its actions are affecting the overall performance of the system. Although these rewards already reflect the main goals of this work, there is one additional reward term added to minimize communication needs and error rates, an action reward.

$$r_u(k) = \begin{cases} 0, & \text{if } u = None \\ \beta_u & \text{otherwise} \end{cases} \tag{25}$$

Where β_u is a small negative constant to penalize speed adjustments. Thus, the agent will try to optimize its decision-making process to take as few adjustments (or actions) as possible, reducing communication effort. The overall reward function is obtained by combining the previous components.

$$r(k) = \begin{cases} r_e(k), & \text{if } x = x_{terminal} \\ r_s(k) + r_v(k) + r_u(k), & \text{otherwise} \end{cases} \tag{26}$$

The final reward function in Equation 26 balances the different requirements for the system. The task of the DRL agent is to optimize its policy to maximize this reward function over an episode. In Figure 2 aircraft ID 3 is shown in orange, meaning that at this step the seventh entry of the action space (a_3^+) equals 1 according to the policy learned and this vehicle's speed is being adjusted (increased to $90m/s$).

3.2 Solution Approach

The objective of our single-agent DRL solution is to find a policy that maximizes the expected return given the reward function defined in Equation 26. To train the agent in our custom environment explained before, the PPO algorithm was chosen and implemented using the Stable-Baselines3 (SB3) RL library based on PyTorch DL library [Raffin et al., 2021, Paszke et al., 2019]. The reasons for this decision are two-fold. First, policy

gradient methods are appropriate for this research as the environment is not expensive to sample from, and, as mentioned in Section 2, PPO outperforms other policy gradient methods. Secondly, previous work on en-route conflict resolution used the loss function (in [Brittain and Wei, 2019]) or the full PPO architecture (in [Brittain and Wei, 2021, Brittain et al., 2020]) to train their solutions and obtained good results, demonstrating the suitability of PPO for this type of complex task.

Hyper-parameter	Value
Optimizer	Adam [Kingma and Ba, 2014]
Learning rate (α)	3×10^{-4}
Horizon (T)	2048
Minibatch size (M)	64
Number of epochs (K)	10
Discount factor (γ)	0.99
GAE parameter (λ)	0.95
Clipping parameter (ϵ)	0.2
Value function coefficient (c_1)	0.5
Entropy coefficient (c_2)	0.0

Table 1: Hyper-parameters in PPO implementation (inspired from [Raffin et al., 2022]).

The implementation hyperparameters are summarized in Table 1. These hyperparameters were proposed in the original PPO paper ([Schulman et al., 2015b]) for robotic tasks and represent the default SB3 library values. The value function and entropy coefficients are respectively the constants c_1 and c_2 in the complete PPO loss function, considering the value and entropy terms:

$$L_k^{CLIP+VF+S}(\theta) = \mathbb{E}[L_k^{CLIP}(\theta) - c_1 L_k^{VF}(\theta) + c_2 S[\pi_\theta](x_k)] \tag{27}$$

Where L_k^{CLIP} is defined in Equation 10, and $L_k^{VF} = (V_\theta(x_k) - V_k^{targ})^2$ [Schulman et al., 2017]. The architectures for the policy and value networks are shown in Figure 4. The structure of the NN is also inspired by the original PPO work [Schulman et al., 2015b] and the default library values. The actor and critic have separate networks, each with two fully connected layers of 64 nodes and a tanh activation function. The output layer of the policy network has a softmax activation function to output the probability for each discrete action, while the value function has a linear output layer [Raffin et al., 2021].

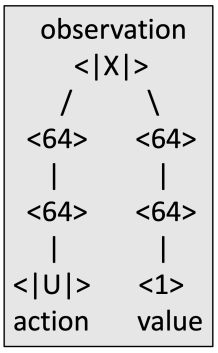


Figure 4: Policy and value network architectures (SB3-PPO implementation [Raffin et al., 2021]). The actor and critic networks each contain two layers of 64 units.

The policy and value network parameters are initialized using orthogonal initialization proposed in [Saxe et al., 2013]. The reward function was tuned using the parameters in Table 2. Finally, the agent has trained for a total of 3 million steps with $|\mathcal{N}| = 10$ aircraft flying over the sector per environment episode. During the rollout phase, a set of experiences of dimension NT is collected sequentially by 4 actors and used to update the actor and critic for K epochs. The environment is reset every time the terminal state is achieved. During training, the actions are sampled using the distribution output from the policy. During testing, the most likely action is chosen.

Reward parameter	Value
L	10
β_s	-0.005
β_v	0.01
β_u	-0.1

Table 2: Reward parameters in Equation 26.

4 Experiments

Real-world scenarios are likely to have unexpected interferences that were not considered during training, such as the presence of unauthorized vehicles, strong winds, tall buildings, and malicious activities [Jang et al., 2017]. Therefore, it is fundamental to understand the behavior of the autonomous centralized controller in as many simulated unpredictable circumstances as possible before real-world deployment. During these experiments, we varied aircraft arrival rates and speed, sector configurations, and communication availability.

In the scope of this work, we consider three distinct evaluation metrics to measure the performance of the trained agent: the likelihood of success computed as the number of successful episodes divided by the total number of episodes (this equals one minus the likelihood of collision), the average time to cross the intersection per aircraft, and the average distribution of actions along the route, or the number of times the agent takes a speeding up or slowing down action with respect to the distance to the merging point.

4.1 Experiment I: Varying Aircraft Arrival Rate and Speed

Arrival configurations are the conditions under which aircraft enter the sector, namely the inter-arrival time, determined by the Poisson process parameter for each route, initially set to $\lambda = 20s$, and the entry speed, initially set to $v_e = 50m/s$. In real-world scenarios the arrival settings may change due to multiple factors - strong wind gusts altering vehicle dynamics, different aircraft types, onboard system failures, and more. This experiment focuses on performing sensitivity analysis by varying these parameters and analyzing model outputs. For this analysis the sector is set as shown in Figure 2 and remains fixed, i.e. routes are not randomly generated as during training.

4.2 Experiment II: Varying Sector Structure

Route configurations refer to the structure of the airspace sector. This can vary depending on the location the agent is deployed. For this reason, we considered randomized environments during training. Two factors can vary in an airspace configuration: the number of entry routes in the merging point and the angular distance between routes. After extensive testing with different preset angular distances between entry routes, it was concluded that this parameter does not yield a significant difference in the performance indicators. This is counter-intuitive, as making routes closer to each other increases the likelihood of a collision happening. Moreover, in structured airspace, it is unrealistic to consider too small inter-route separation. Therefore, only the variation in the number of routes (between 2 and 6 entry routes) is considered. For this experiment, n routes were generated at random among the set of possible entries and λ is constant and equals the training reference value ($\lambda = 20$).

4.3 Experiment III: Blocking Aircraft Communications

There is a vast number of reasons why communication between the controller and aircraft may be blocked: unauthorized UAS that do not respond to speed commands, tall buildings or hills, network failures, and more. For this reason, it is important to assess whether using the trained agent would result in improved performance under communication blockage conditions. This is if the agent can readjust the system and provide speed advisories to responding vehicles in order to maintain safety in the sector, i.e, to avoid collisions, even though it was not trained in these settings. Or whether having no agent at all would represent a higher likelihood of success. This represents a hypothesis-testing case. The goal is to understand if deploying our agent in a situation of communication blockage improves the overall safety of the sector when compared with the case with no speed advisories given.

5 Results

This section is focused on displaying the results of the model developed to answer our main research question: How to design a centralized scalable conflict resolution approach based on DRL for UAM? To begin with, the

training metrics of the PPO training process are discussed. Then, the experiments described in the previous section are realized to assess the performance of our trained agent under variable environmental conditions. For the first and second experiments, the performance metrics discussed: average success rate, time to cross the sector, and action distribution are calculated for variable environment settings. For the third experiment, since the outcome of the experiment (the success of an episode) is a binary random variable, the Z-test for proportions is applied at a significance level of 5%.

5.1 Training Metrics

The model proposed trained for approximately 3 million timesteps, or until the cumulative reward curve in Figure 6 stabilized. Each timestep is a step of the environment during a rollout, meaning that approximately 1468 rollouts were completed in 4 environments (or 367 rollouts per environment). Since we used *DummyVecEnv* SB3 function to emulate paralleled environments, they are called sequentially during the training process. Thus, to collect one rollout of experience across the 4 environments for training takes 8192 timesteps. To obtain the training metrics, we ran the current policy every $10k$ timesteps in an evaluation environment for 100 episodes and averaged the results. Thus, training results correspond to the evolution of the agent during the process. Since the frequency of evaluation is lower than the frequency of update of the model, no consecutive evaluation runs use the same PPO agent.

In this subsection, besides displaying the training evolution, we want to compute accurate performance metrics of our trained agent using the default training environment. To ensure the simulation is run a sufficient number of episodes for the output results to stabilize, the coefficient of variation (CV) is used. We recompute this value every time new 100 data points, or outputs of the environment for a given performance metric, are simulated using the learned policy. One data point is a complete episode.

$$CV = \frac{\sigma}{\mu} \tag{28}$$

After running this formula for $10k$ episodes, we concluded that the success rate metric stabilizes after approximately $6k$ episodes, regardless of whether the agent takes action or not. Considering the time to cross the sector, the CV stabilized after $4k$ episodes. It should also be noticed that the version of the model picked for testing the agent once we were finished with training was chosen based on the maximum average cumulative reward achieved during the training evaluation callback described.

The most relevant training metric is the average success rate (over 100 episodes). Figure 5 depicts the evolution of this metric over time. As mentioned, success is achieved when Equation 18 is met, i.e., every aircraft exited the sector without collisions. To establish a baseline, we measured the success in case there is no controller or the controller always chooses $u = None$, and the average success rate is about 38% in this case.

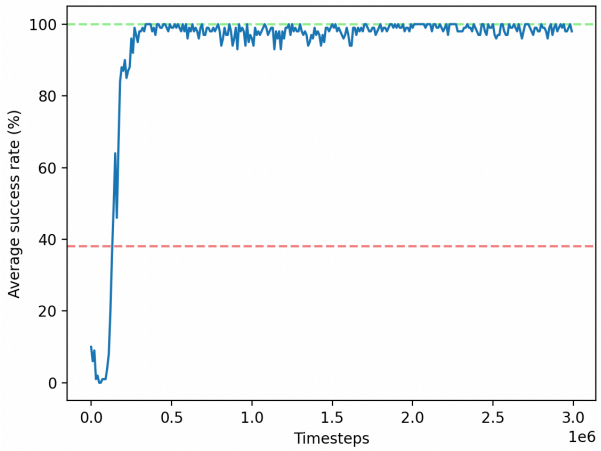


Figure 5: Evolution of the average success rate with training. In red is the average rate if no actions are taken. In green is the optimal rate.

We observe that the autonomous controller quickly surpasses this threshold and converges to the optimal success rate (100%). Once training was finished, the agent obtained an overall performance of about 99% in successfully providing aircraft with speed advisories in the custom environment designed. This means that the performance of the environment with the trained agent deployed is 61% superior to the performance without speed advisories. The average time to cross the sector per aircraft is $46s$. This represents 36% less time compared to the case with no actions taken ($\frac{1800m \times 2}{50m/s} = 72s$), proving that deploying the agent also improves the efficiency of the system. We also measured the average percentage of invalid actions, i.e. actions that are not directed to

an active aircraft, taken by the agent over the course of an episode. These represent approximately 11% of the total number of actions. Although this value has no effect on the overall safety of the system (because these actions are filtered), it may deteriorate the system as these actions are not being effectively used to increase the reward.

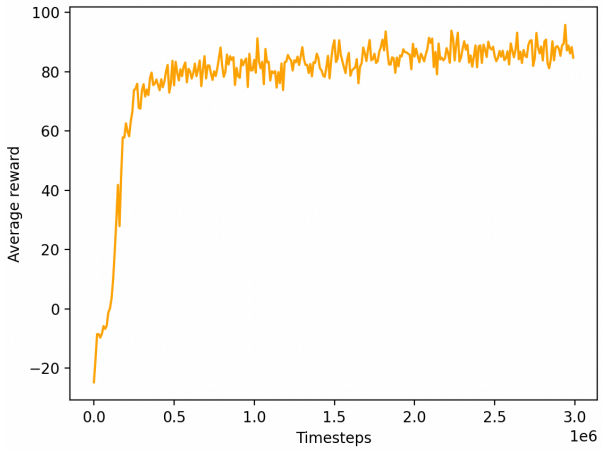


Figure 6: Evolution of the average reward with training.

Figure 6 depicts the evolution of rewards with training. These are average cumulative rewards obtained over the span of an episode by the agent, according to the reward function defined in Equation 26. We observe a similar pattern to the evolution of the success rate. Rewards start by rapidly increasing from a value of approximately -20, followed by a slower convergence to approximately 90. The resemblance between these two curves demonstrates the relationship between maximizing rewards and achieving higher success rates. Moreover, the smooth and rapid convergence of the cumulative reward shows that the agent is efficiently learning to maximize this function.

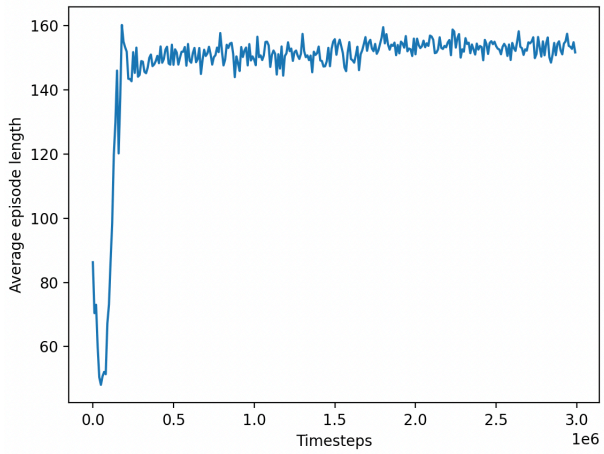


Figure 7: Evolution of the average episode length with training.

Figure 7 shows the evolution of the average episode length, i.e., the number of timesteps an episode lasts on average. To maximize the reward, the agent must guarantee no collisions for an episode length that is long enough for all aircraft to leave the sector. However, since there are also efficiency goals in place, this time should be limited. We observe that after a sharp increase in episode length during the beginning of training, this value stabilizes at around 150 steps per episode.

5.2 Experiment I: Analysis of Variable Aircraft Arrival Rate and Speed

Figure 8 depicts the variation of the success rate (average over 100 episodes) with the variation of the Poisson parameter λ. For each possible integer value of λ between 5 and 20, 10 data samples were computed. We observe an S-shaped curve where the evolution is approximately linear between $\lambda = 6$ and $\lambda = 11$, converging to 1 for $\lambda \geq 14$. This proves that although the model was not trained for $\lambda \neq 20$, it can generalize and handle more challenging cases, where the expected separation between aircraft is lower than the outer boundary, with a high success rate.

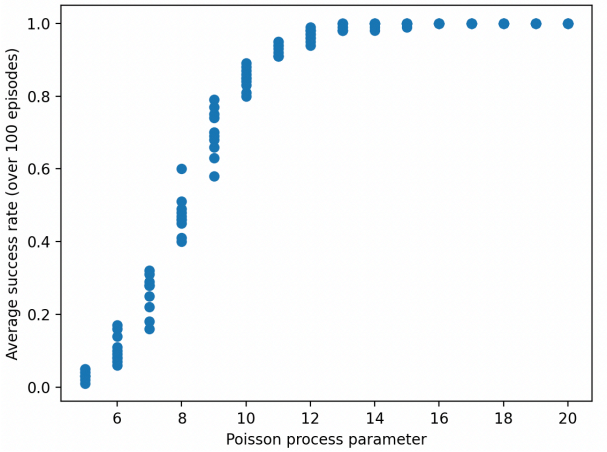


Figure 8: Variation of the success rate (averaged over 100 episodes) with the Poisson process rate λ in a two-entry routes sector configuration.

To analyze the variation of the average time to cross the sector with the Poisson parameter, only successful aircraft were considered, meaning that values from vehicles that do not finish crossing the sector due to a collision are filtered. As a baseline, one should remember that the time to cross the sector if the agent takes no action is $72s$.

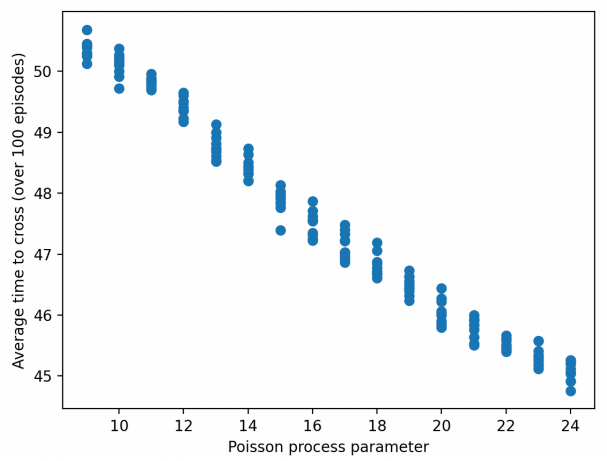
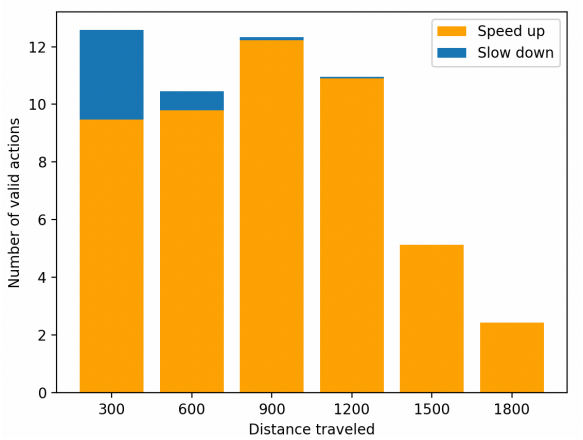


Figure 9: Variation of the time to cross the sector per aircraft (measured in seconds and averaged over 100 episodes) with the Poisson process rate λ in a two-entry routes sector configuration.

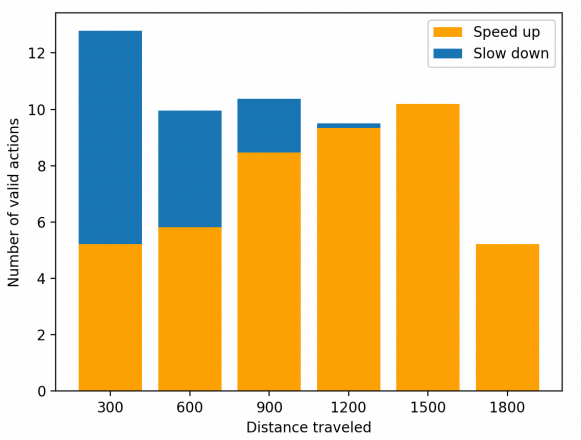
We observe that the average time to cross the sector decreases linearly with the increase in the inter-arrival time between aircraft. An explanation for this phenomenon is that as the Poisson parameter decreases the agent must handle a higher number of conflicting aircraft at the same time, mainly because the expected separation is reduced and merging aircraft are condensed in a shorter period of time. Therefore, it must slow down a higher number of vehicles in order to create sufficient separation to merge them safely, increasing the average time to cross the sector.

The explanation given for the trend found in Figure 9 can be viewed in the evolution of actions with the distance covered by aircraft. This is obtained by averaging the number of speeding up and slowing down actions applied on aircraft over the course of the first $1800m$ traveled. This average was done by collecting data from $6k$ episodes. Only actions taken during the entry route are considered as it was observed that most actions are concentrated during this segment. Moreover, invalid actions are filtered so that only commands that contribute to the outcome of the simulation are analyzed.

Figure 10 shows the results obtained. As stated, the number of slowing-down actions increases significantly when we halve the Poisson parameter value. This justifies the longer time to cross the sector. This trend is mainly observed during the first $900m$, when the agent is creating sufficient separation to be able to merge aircraft safely. The total number of actions when $\lambda = 20$ is approximately 54 (with only about 8% slowing down actions), while this number increases to 58 (with slowing down actions representing 24% of the total) when $\lambda = 10$. Thus, there is a three times higher share of actions to reduce speed when $\lambda = 10$, compared to the reference value used during training. Another trend that can be visualized is that the total number



(a) Average action distribution for $\lambda = 20$.



(b) Average action distribution for $\lambda = 10$.

Figure 10: Average action distribution with respect to distance traveled varying the Poisson parameter λ. Distance values are the upper boundaries of the intervals where actions were measured. Previous distance values are the lower boundaries.

of actions tends to reduce as aircraft get closer to the merging point, which is desirable as conflicts must be resolved earlier. In the more complex case ($\lambda = 10$), the number of speeding-up actions grows when aircraft get closer to the intersection. A plausible explanation is that the agent is first prioritizing safety, creating additional separation between potentially conflicting aircraft, and only later increasing their speed to match the efficiency requirements.

To test the impact of the case when an aircraft arrives at a different speed than the reference $v_e = 50m/s$, this value was varied for aircraft ID 5. Everything else remains as before, $\lambda = 20$, and the same airspace configuration.

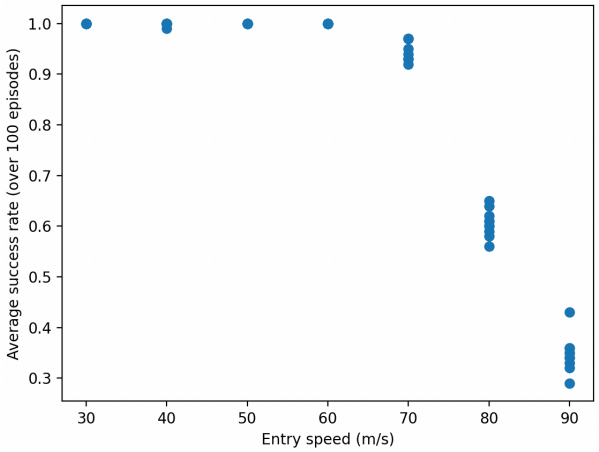


Figure 11: Variation of the average success rate with the entry speed of aircraft ID 5.

Figure 11 shows that if $v_e \leq 60m/s$, the agent has no problems generalizing, despite the fact that this parameter was constant during training. For higher entry speed values, the performance decreases linearly. A possible explanation is that the agent has not learned to decelerate aircraft that enter with a large speed difference from other aircraft in the same route, causing a later collision. To overcome this obstacle, future research could consider modifying this parameter during the training phase. Only the impact of 1 agent was studied as this is an extraordinary case and such high deviations from the preset entry value are not expected during controlled operations. Furthermore, this shows that the model can handle smaller and more likely deviations with almost no loss in accuracy.

5.3 Experiment II: Analysis of Variable Route Configurations

Figure 12 shows the results. It is observable that the performance highly decreases for the cases the agent was not trained on, i.e. 4, 5, and 6 entry routes. Furthermore, a decrease in performance also happens between 2

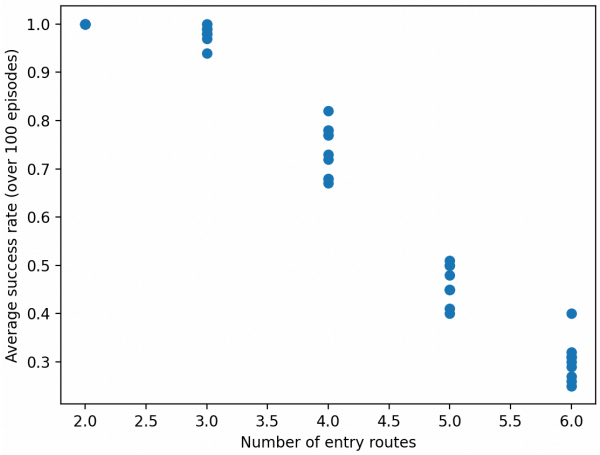


Figure 12: Variation of the average success rate (averaged over 100 episodes) with the number of entry routes.

and 3 routes, despite that the agent was trained approximately as many times for each case. An explanation for this decrease is that at increasing the number of routes, the likelihood of conflicting agents at the intersection point increases due to the fact that the Poisson process is independent per route. Moreover, it also increases the possible number of aircraft the agent must deconflict during a short time span. Therefore, adding more routes increase the complexity of the agent's task. For this reason, as with decreasing the Poisson parameter, increasing the number of routes also has a negative impact on the average time to cross the sector per aircraft.

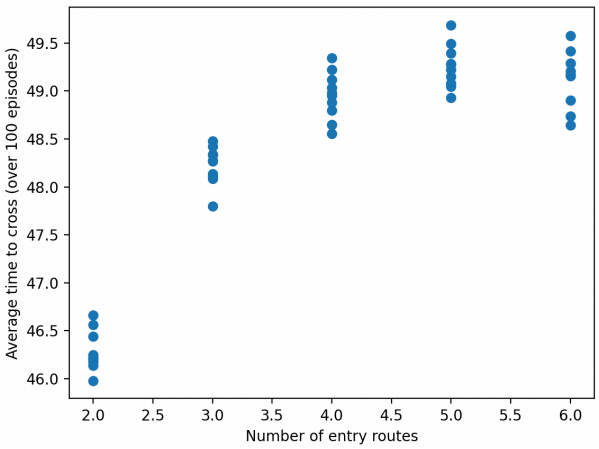
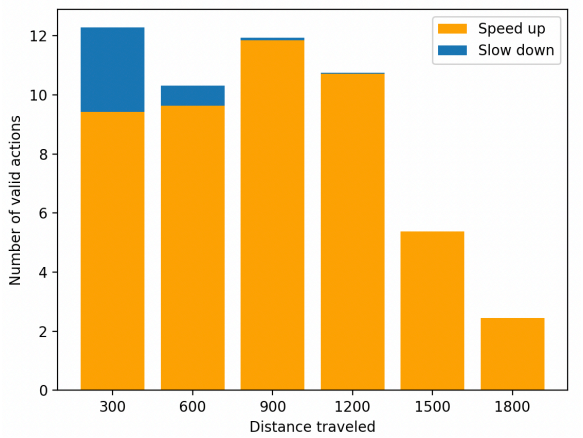


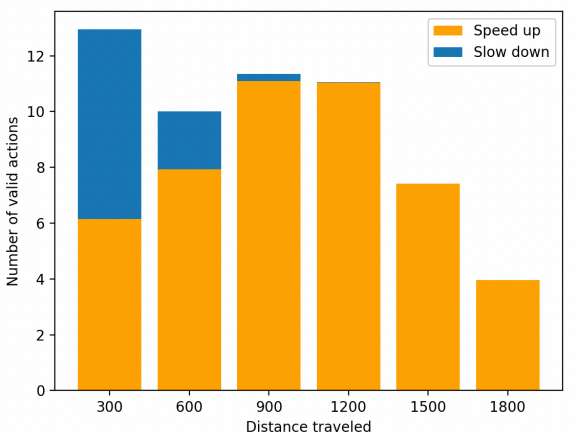
Figure 13: Variation of the time to cross the sector per aircraft (measured in seconds and averaged over 100 episodes) with the number of entry routes.

Figure 13 displays the variation of this variable with the increase in the number of routes. We see that the largest difference happens between 2 and 3 route configurations, and decreases thereafter. A justification for this may be that since the success rate decreases, and we are only considering successful aircraft in the time to cross calculation, the remaining aircraft may be the ones who arrived first and thus, have lower crossing times because do not need to take part in a deconfliction process. As before, this can be explained by the evolution of actions with distance traveled for 2 and 3 entry route cases in Figure 14.

It is observable that adding an extra entry route makes the agent take more deceleration actions in the first half of an aircraft's route. In the case with 2 routes deceleration actions represent only 7.5%, while with 3 entry routes it increases to about twice as much at 16% of total actions taken. We also see that by increasing the number of routes, the total number of actions increases by 4 (from 53 to 57). By comparing Figure 10 and Figure 14, one may conclude that the effect of adding more entry routes is very similar in terms of the distribution of actions to that of reducing the Poisson process rate. One more interesting consideration is that even in the more challenging cases, the average number of valid actions per aircraft does not surpass 6, not considering the *None* action, meaning the agent is also being efficient in terms of the number of communications it must establish with aircraft. Future research could consider incorporating a larger number of entry routes during training to improve these results, provided that this number would match the requirements for deployment in real urban settings.



(a) Average action distribution for $n = 2$.



(b) Average action distribution for $n = 3$.

Figure 14: Average action distribution with respect to distance traveled varying the number of entry routes n.

5.4 Experiment III: Analysis of the Effects of Blocking Aircraft Communications

To begin with, the mean success rate, if the agent does not communicate with aircraft at all in the case of the randomly generated routes (considering $n = 2$ and $n = 3$ entry routes), is 0.383. Given this value, we state the null and alternative hypotheses. The null hypothesis is that deploying the agent has a positive effect on the success rate of the system in case communication with a random aircraft is blocked over the course of an episode, i.e. the proportion of successful episodes is greater than or equal to the target value 38.3%. The alternative hypothesis is that using the agent on this specific disruption case has a negative effect on safety, or that the proportion of successful runs is lower than the target value. After sampling for 100 episodes, the proportion of successful episodes using the trained agent is 0.3. The Z-score is calculated as [Zou et al., 2003]:

$$Z = \frac{p - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}} = \frac{0.3 - 0.383}{\sqrt{\frac{0.383 \cdot (1-0.383)}{100}}} \approx -1.7 \tag{29}$$

Where p_0 is the hypothesized success rate proportion, or target value, p is the observed proportion, and n is the number of samples. Considering the significance level of 0.05, and the Z-score of -1.7, the p-value is 0.0446, which is lower than the significance level of 0.05, and thus, there is enough evidence to reject the null hypothesis at a 5% significance level. We conclude that in the communication blockage case, using the trained agent has a negative safety impact on the sector. Future research could consider random communication blockage during training to improve the performance of the autonomous controller in these cases that are likely to exist if the system is deployed in real-world conditions.

6 Conclusions

This research aimed to design and test a centralized and scalable approach to autonomous ATC for UAM based on DRL. Two main performance targets were considered to engineer this model: the maximization of the number of successful episodes without collisions and the minimization of the time it takes for an aircraft to cross the sector. Our approach in terms of observation and action spaces scales linearly with the number of agents, making it possible to experiment with a crowded sector containing 10 vehicles. By defining the inter-arrival times on each route according to a random Poisson process, the agent cannot memorize actions and has to learn the system's general dynamics to control it. Three main principles showed positive learning results when applied to the observation space: first, the normalization of the state values, second, sorting aircraft information in the observation vector according to their global arriving time, and lastly, using zero-padding in the positions of non-active aircraft. The action space is simple, yet effective according to our results. Considering acceleration and deceleration actions not only contributes towards the possibility to human-supervise the system but also to the facilitation of communication protocols between aircraft and the controller. Finally, the reward function was designed to reflect our safety and efficiency goals, densely penalizing aircraft approaching the inner threshold and rewarding those traveling at higher speeds. The minimization of actions was also considered in an effort to reduce errors and maximize communication efficiency. The custom environment developed was used to train a CTCE DRL framework using PPO. Training results depicted a stable learning agent, rapidly converging to high success rate values. After training, the best model for testing was chosen based on the maximum average episode reward achieved. When tested on the training environment, i.e., a sector of randomly generated routes,

with a number of entry routes varying between 2 and 3, this model achieves approximately 99% of successful simulations, around 61% more than the case where no agent is deployed on the environment, proving the effect of training and the versatility of the agent to be deployed in different airspace configurations, due to the random configuration nature of this train and test environment.

During experiments, different parameters of the environment were studied. First, the expected inter-arrival time parameter was varied, demonstrating that shorter times decrease performance, both in terms of success rate, but also average time to cross the sector per aircraft, which is proved by the distribution of actions, as in the more complex case where the expected arrival distance between aircraft is smaller, the number of decelerating actions taken at the beginning of aircraft journeys increases significantly. Varying the entry speed of a single aircraft also showed a decrease in the overall system's safety if this speed is greater or equal to $70m/s$, and thus, even though we are considering an autonomous operation, ground rules will most likely have to be defined in terms of the allowed entry speed envelope. Most surprisingly, for both tests, the agent demonstrated some generalization capabilities for environment parameter values that were not seen during training. For instance, despite being trained for an expected arrival separation of $1km$, the agent achieved a 100% success rate given an arrival separation 25% smaller, at $750m$, in the fixed two entry-route case scenarios. Varying the number of routes has proven to have a similar impact as shortening the expected arrival separation, with the variation from two to three entry routes, yielding approximately the same results both in terms of loss of safety and added delays as reducing the reference Poisson parameter value by 10%. Lastly, a hypothesis test conducted on the impact of blocking communications with a random aircraft throughout an episode concluded that our trained agent deteriorates safety in this case.

7 Discussion and Future Work

We identify four main limitations in this research: vehicle homogeneity, unrealistic flight dynamics, limited scalability, and invalid actions. First, vehicle homogeneity refers to the fact that all aircraft were considered to be the same during this study, i.e. respecting the same communication protocols, entry speed, speed envelope, acceleration requirements, collision distance, et cetera. However, in real-world conditions, vehicles are expected to be heterogeneous, ranging from small package delivery drones to larger passenger transportation VTOLs. Secondly, this research considered instantaneous speed changes that are not realistic in actual flight conditions. Thirdly, our solution presents limited scalability. Despite the fact that both action and observation spaces scale linearly with the number of aircraft, increasing the number of agents from a certain threshold will bring performance losses. Moreover, using zero-padding provides the NN with irrelevant information on inactive aircraft. Lastly, invalid actions, or speed advisories to inactive aircraft, revealed a shortcoming of this research approach, solved by filtering these values.

Regarding future research to address these shortcomings, our results indicate that including a range of possible Poisson parameters, route numbers, and communication blockage cases during training may improve performance under real-world failure scenarios. Moreover, future work could try a mixed airspace environment with multiple types of vehicles interacting and perhaps a dynamic collision threshold defined by additional variables to the vehicle's wingspan. This would make the system more realistic in terms of what is expected from future urban operations. In terms of improving flight dynamics realism, researchers could use the Bluesky simulator instead of a custom Gym environment. Another advantage of using Bluesky is to establish a comparable baseline with other research on en-route conflict resolution [Hoekstra and Ellerbroek, 2016, Wang et al., 2022]. A CNN or LSTM approach could be taken to compress state information into a fixed-size vector, potentially improving learning and scaling to more agents. However, a solution to the problem of a variable action space size is also required to cope with an indeterminate number of aircraft. Another solution approach that could be tested is to use the approach proposed in this research with dynamic allocation of new agents entering the sector. Since the expected separation between aircraft can be controlled using the Poisson parameter and is expected to be regulated in future operations, it is possible to determine the maximum number of aircraft in the sector at the same time. If the observation and action spaces can cope with this number of vehicles, new incoming ones can be allocated to the positions that were zero-padded because others left the sector. This dynamic allocation could bring additional training challenges but present a solution to the continuous day-to-day operations problem. Finally, future research could also focus on finding a reward function to minimize the number of invalid actions that deteriorate the overall performance of the system.

References

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Brittain and Wei, 2018] Brittain, M. and Wei, P. (2018). Autonomous aircraft sequencing and separation with hierarchical deep reinforcement learning. *International Conference on Research in Air Transportation*.

[Brittain and Wei, 2019] Brittain, M. and Wei, P. (2019). Autonomous air traffic controller: A deep multi-agent reinforcement learning approach. *arXiv preprint arXiv:1905.01303*.

[Brittain et al., 2020] Brittain, M., Yang, X., and Wei, P. (2020). A deep multi-agent reinforcement learning approach to autonomous separation assurance. *arXiv preprint arXiv:2003.08353*.

[Brittain and Wei, 2021] Brittain, M. W. and Wei, P. (2021). One to any: Distributed conflict resolution with deep multi-agent reinforcement learning and long short-term memory. In *AIAA Scitech 2021 Forum*, page 1952.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[Buşoniu et al., 2010] Buşoniu, L., Babuška, R., and De Schutter, B. (2010). Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221.

[Campbell et al., 2002] Campbell, M., Hoane Jr, A. J., and Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2):57–83.

[Culjak et al., 2012] Culjak, I., Abram, D., Pribanic, T., Dzapo, H., and Cifrek, M. (2012). A brief introduction to opencv. In *2012 proceedings of the 35th international convention MIPRO*, pages 1725–1730. IEEE.

[Dayan and Watkins, 1992] Dayan, P. and Watkins, C. (1992). Q-learning. *Machine learning*, 8(3):279–292.

[Erzberger, 2004] Erzberger, H. (2004). Transforming the nas: The next generation air traffic control system. Technical report.

[Erzberger, 2005] Erzberger, H. (2005). Automated conflict resolution for air traffic control.

[Haydari and Yilmaz, 2020] Haydari, A. and Yilmaz, Y. (2020). Deep reinforcement learning for intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems*.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hoekstra and Ellerbroek, 2016] Hoekstra, J. M. and Ellerbroek, J. (2016). Bluesky atc simulator project: an open data and open source approach. In *Proceedings of the 7th international conference on research in air transportation*, volume 131, page 132. FAA/Eurocontrol USA/Europe.

[Hunter and Wei, 2019] Hunter, G. and Wei, P. (2019). Service-oriented separation assurance for small uas traffic management. In *2019 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, pages 1–11. IEEE.

[Jang et al., 2017] Jang, D.-S., Ippolito, C. A., Sankararaman, S., and Stepanyan, V. (2017). Concepts of airspace structures and system analysis for uas traffic flows for urban areas. In *AIAA Information Systems-AIAA Infotech@ Aerospace*, page 0449.

[Joby, 2022] Joby (2022). Joby receives part 135 certificate from the faa.

[Joulia et al., 2016] Joulia, A., Dubot, T., and Bedouet, J. (2016). Towards a 4d traffic management of small uas operating at very low level. In *ICAS, 30th Congress of the International Council of the Aeronautical Sciences*.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[Karen Dix-Colony,] Karen Dix-Colony, B. B. Operating the 747-8 at existing airports.

[Kim, 1999] Kim, D. (1999). Normalization methods for input and output vectors in backpropagation neural networks. *International journal of computer mathematics*, 71(2):161–171.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Konda and Tsitsiklis, 1999] Konda, V. and Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, 12.

[Lonza, 2019] Lonza, A. (2019). *Reinforcement Learning Algorithms with Python*. Packt Publishing.

[Matternet, 2022] Matternet (2022). Matternet receives faa production certificate for its m2 drone delivery system.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[Raffin et al., 2021] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355.

[Raffin et al., 2022] Raffin, A., Kober, J., and Stulp, F. (2022). Smooth exploration for robotic reinforcement learning. In *Conference on Robot Learning*, pages 1634–1644. PMLR.

[Saxe et al., 2013] Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

[Schulman et al., 2015a] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015a). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR.

[Schulman et al., 2015b] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

[Stoll et al., 2014] Stoll, A. M., Stilson, E. V., Bevirt, J., and Pei, P. P. (2014). Conceptual design of the joby s2 electric vtol pav. In *14th AIAA Aviation Technology, Integration, and Operations Conference*, page 2407.

[Straubinger et al., 2020] Straubinger, A., Rothfeld, R., Shamiyeh, M., Büchter, K.-D., Kaiser, J., and Plötner, K. O. (2020). An overview of current research and developments in urban air mobility–setting the scene for uam introduction. *Journal of Air Transport Management*, 87:101852.

[Van Hasselt et al., 2016] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

[Vinitsky et al., 2018] Vinitsky, E., Kreidieh, A., Le Flem, L., Kheterpal, N., Jang, K., Wu, C., Wu, F., Liaw, R., Liang, E., and Bayen, A. M. (2018). Benchmarks for reinforcement learning in mixed-autonomy traffic. In *Conference on robot learning*, pages 399–409. PMLR.

[Wang et al., 2022] Wang, Z., Pan, W., Li, H., Wang, X., and Zuo, Q. (2022). Review of deep reinforcement learning approaches for conflict resolution in air traffic control. *Aerospace*, 9(6):294.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

[Zou et al., 2003] Zou, K. H., Fielding, J. R., Silverman, S. G., and Tempany, C. M. (2003). Hypothesis testing i: proportions. *Radiology*, 226(3):609–613.

II

Literature Study
previously graded under AE4020

1 Introduction

Urban Air Mobility (UAM) is a concept of the future airspace, where people and goods are transported by partially or fully autonomous air vehicles. Air taxis, VTOLs (vertical take-off and landing), and personal aircraft are some of the innovations we are seeing in this field. However, without a proper definition of operations and safety measures, these concepts will not massively take off anytime soon. Especially in highly complex areas such as the Urban Airspace, where there are many static and dynamic obstacles such as buildings, humans, cars, and both piloted and autonomous air vehicles. Moreover, the presence of unauthorized drones, strong wind, and malicious actors make UAM operations particularly challenging [30].

One of the first problems to be tackled is the Air Traffic Control (ATC) of high-traffic urban aerial spaces. Traditional ATC relies on human operators and three main conflict resolution methods: altitude adjustment, heading adjustment and speed adjustment [64]. However, with an increased demand for flight operations both in high altitude routes with the FAA estimating a growth en-route aircraft at a rate of 1.5% [1], as well as low altitude flights with passenger VTOL and delivery drone concepts getting approval to begin operations [5] [40], there is a growing need for an autonomous ATC system [29].

Artificial Intelligence (AI) algorithms have been outperforming humans in games over the past years. IBM Deep Blue beat the world chess champion Garry Kasparov in 1997 [14]. In 2013 Deepmind's scientists proposed a method called Deep Q-Networks that outperformed human experts in multiple arcade games using raw pixel inputs [41]. Later, in 2016 Deepmind developed AlphaGo and beat Lee Sedol, one of the strongest Go players, a game exponentially more complex than chess [57]. All these results rely on the same framework: Reinforcement Learning (RL). RL algorithms learn without prior environment knowledge by interacting via trial-and-error and a reward function. Divided into value-based and policy-gradient, and particularly when combined with Deep Learning (DRL) to handle continuous environments, these methods are well-suited for sequential decision-making games, such as autonomous ATCos (Air Traffic Control Operations) [64].

This work proposes using DRL to build an autonomous centralized urban (low-altitude) ATC agent capable of being deployed in locations with different airspace configurations. There are five main components to be defined: the environment for training, the information the agent requires (observation space), the type of actions the agent can take (action space), the reward function that reflects the goals and measures how well the agent is performing, and the training algorithm. This report aims to help define this structure. Chapter 2 introduces the three conflict resolution methods used in ATCos: altitude adjustments, heading adjustments, and speed adjustments. Moreover, it introduces tools, techniques, and challenges for Unmanned Aircraft Systems (UAS) conflict resolution, useful for defining the rules for low-altitude Urban Air Mobility (UAM). Chapter 3 introduces the principal RL and DRL techniques for single-agent environments. Chapter 4 answers the main problems that arise by extending these algorithms to multi-agent settings: non-stationarity, partial observability, the credit assignment problem, and continuous environments, as well as the main multi-agent frameworks: Centralized Training Centralized Execution (CTCE), Centralized Training Decentralized Execution (CTDE), and Decentralized Training Decentralized Execution (DTDE). Chapter 5 explores recent DRL solutions in literature for the en-route conflict resolution problem, including the principal five components mentioned earlier. Finally, Chapter 6 describes the research proposal that results from this review, including the research questions, the framework used, and suggested experiments.

the FAA to operate commercial flights with its fleet of eVTOL (electric vertical take-off and landing) that can carry one pilot and up to four passengers [5]. A report published in 2018 expects twenty thousand unmanned missions to be flown in the Paris urban airspace by 2035 [2]. Therefore, new ATM solutions must be found to cope with this increased demand, maintaining the safety and efficiency of airspace operations.

2

Conflict Resolution

To start this research on autonomous separation assurance, it is important to understand how current Air Traffic Management systems work. In this chapter, the definition of conflict is presented, along with the different resolution methods used by Air Traffic Controllers (ATCos) in traditional manned aviation. Then, the challenges and possible solutions for Unmanned Aircraft Systems are presented.

2.1. Conflict Resolution in Air Traffic Control

According to the International Civil Aviation Organization (ICAO), a conflict is defined as the loss of separation between two aircraft. The recommended horizontal separation is 5 Nautical Miles (NM) and the vertical separation is 1000ft [4]. This means that the safety area for an aircraft is defined as a three-dimensional cylinder with a 5NM radius and 2000 ft height centered on the airplane as shown in Figure 2.1. Thus, if a second air vehicle enters this safety area, separation is lost and a conflict occurs [64].

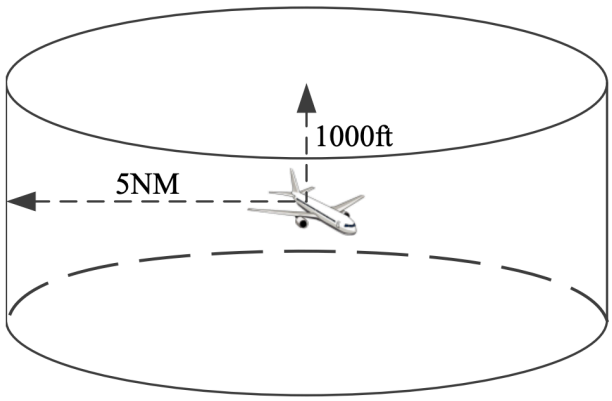
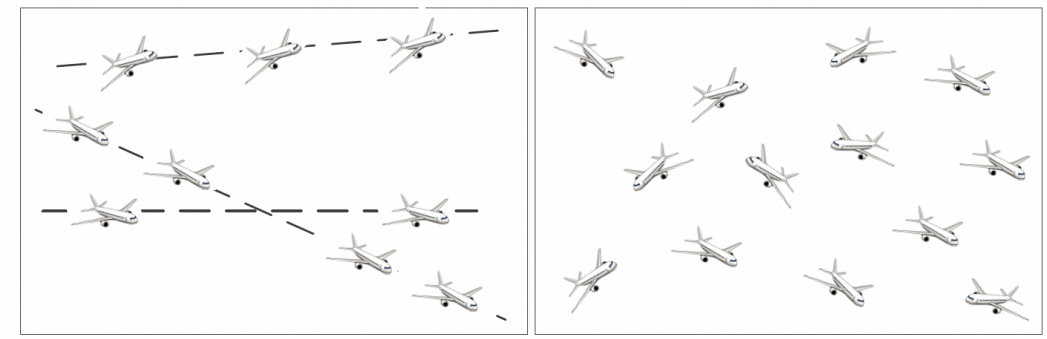


Figure 2.1: Three-dimensional safety area around aircraft [64].

There are two main scenarios in flight operations: free flight and en-route flight. Free flight means an aircraft can fly freely in the airspace without following a pre-specified route between an entry and exit point. On the other hand, en-route flight means an aircraft must follow fixed airways [64]. En-route flight has the advantage that routes can be deconflicted before take-off. However, this comes at the cost of traffic congestion, a problem that can be solved by free flight [64].



(**a**) En-route scenario (**b**) Free flight scenario

Figure 2.2: Different flight scenarios [64].

Three principal conflict resolution methods can be used on their own or in combination by ATCos to resolve conflicts:

1. **Altitude adjustment**. Flight levels are used to maintain vertical separation between aircraft. A Flight level (FL) is expressed in hundreds of feet and represents an aircraft's altitude at a constant atmospheric pressure. For altitude adjustments, the controller must choose a flight level. Below FL41, flight levels are separated by 1000 ft. Above this level, the separation between flight levels increases to 2000 ft. Adjusting altitude is the preferred conflict resolution method [64].

2. **Heading adjustment**. This method changes the heading direction of an aircraft, and consequently, its route. The two approaches for doing this are: first, changing the heading angle by making the aircraft turn left or right by an angle with its current route direction, or secondly, using the offset method, which makes an aircraft fly a pre-determined distance to the left or right to ensure lateral separation [64].

3. **Speed adjustment**. Usually, aircraft fly at a constant cruising speed for energy savings. Speed adjustments are expressed in multiples of 0.01 Mach or 10 kt (knots). Changing speed is the least intuitive approach for ATCos to resolve conflict between aircraft. Also, the speed envelope at cruising altitude is small, making this method the least preferred [64].

There are also two high-level principles to decide which resolution method to choose when facing a conflict in the airspace.

1. Safety is the number one priority, and the resolution of a conflict should not raise new conflicts [64].

2. The conflict resolution method that causes the least traffic disruption and requires the least monitoring effort should be chosen to reduce ATCos workload [64].

The air traffic demand is increasing, both in the traditional airspace, as well as low-level airspace with new services for Urban Air Mobility (UAM). The Federal Aviation Administration (FAA) estimates a rise of en-route aircraft at a rate of 1.5% per year until 2040 [1]. Furthermore, low-altitude carriers are likely to start operations shortly. Matternet, a USA-based drone delivery company recently received FAA approval to manufacture unmanned aircraft [40]. Joby Aviation, another USA-based company was also recently certified by the FAA to operate commercial flights with its fleet of eVTOL (electric vertical take-off and landing) that can carry one pilot and up to four passengers [5]. A report published in 2018 expects twenty thousand unmanned missions to be flown in the Paris urban airspace by 2035 [2]. Therefore, new ATM solutions must be found to cope with this increased demand, maintaining the safety and efficiency of airspace operations.

2.2. Conflict Resolution for UAS Traffic Control

There are seven main separation assurance techniques proposed recently for Unmanned Aircraft Systems (UAS) operations:

1. "**Radar**" [29]. Radar systems use radio waves to sense distances, movements, and velocity. There are two approaches to using radar for UAS safety assurance. First, the traditional method used in aviation today is ground-based air traffic control. This can be achieved through cooperative or non-cooperative radar. Cooperative radar relies on a transponder response from the UAS. If the vehicle is not equipped with a transponder, this method does not work. Non-cooperative radar systems are a solution for this shortcoming. However, due to the very low-level (VLL) flight requirements, the number of installations needed grows, and so does the cost in dense urban areas. As an alternative to ground stations, airborne radar presents an on-board sense-and-avoid (SAA) solution, with fewer communication requirements [29].

2. "**Electro-Optical or Infrared (EO/IR)**" [29]. EO/IR instruments are a passive alternative to radar. While these devices can provide accurate angular measurements, they are constrained by UAS size, weight, and power supply. Moreover, EO sensors lose accuracy in low-light conditions. This shortcoming can be overcome using IR. However, both EO and IR sensors degrade in rainy, foggy, and cloudy weather conditions [29].

3. "**Dependent Surveillance**" [29]. Dependent surveillance considers air vehicles to determine their own navigation data through an onboard satellite-based (GNSS - Global Navigation Surveillance System) system, augmented by other sensors such as accelerometers. This data is broadcasted to air traffic control and other aircraft to allow self-separation. As of 2020, ADS-B (Automatic Dependent Surveillance-Broadcast) is mandatory for all manned aircraft. The advantages of this solution are a long possible range and robustness to all weather conditions. The disadvantages of this solution are increased transmission challenges both to ground stations due to low-level flight and between vehicles due to interference sources such as buildings and terrain, widespread adoption requirements and possible signal integrity and security issues due to data anomalies or dropouts and cyber-attacks, respectively [29].

4. "**LTE and 5G Networks**" [29]. LTE (Long-Term Evolution) and 5G terrestrial communication networks present potential solutions for UAS surveillance. The low latency of 5G could be combined with a time of arrival (ToA) approach for location determination. LTE could also be combined with ADS-B to expand the capabilities of this system to low-altitude flight. Furthermore, ADS-B would improve the reliability of LTE in low-coverage rural regions [29].

5. "**Alerting Boundaries**" [29]. An alert boundary is used to maintain a safe separation from other aerial vehicles. Typically, an inner boundary is used to represent a near-collision event, and an outer boundary represents the last chance to mitigate the risk of collision. However, it is not trivial to define what these boundaries should look like for UAS operations. One idea presented in [29] to compute the inner boundary is to scale it according to vehicle size. If we assume the inner boundary for a manned aircraft to be 100 ft vertical and 500 ft horizontal separation, and its wingspan to be 68 m (Boeing 747-8 [33]), we can compute the inner boundary for the Joby Aviation S2 eVTOL (9 m wingspan [58]) to be 13 ft vertical and 66 ft horizontal separation approximately. The outer boundary is more complex to calculate because it varies with several factors such as the geometry of the conflict, vehicle characteristics, speeds, and weather conditions. Moreover, this outer boundary also relies on the ability of the target aircraft to maneuver cooperatively or not [29].

6. "**Tactical Separation Assurance and Recovery Maneuvers**" [29]. Separation assurance and recovery maneuvers must be designed for when an aircraft encounters the outer boundary threshold. There are several challenges to the design of such actions. First, the uncertainties in performing such complex maneuvers: "These include navigation and surveillance uncertainties in the current state of the vehicles involved, uncertainty in the timing of the separation assurance maneuver, uncertainty in the maneuver execution, uncertainty in the wind field, uncertainty in the target vehicle trajectory..." [29]. Determining the initiation time under these uncertainties can be complex, and an early initiation can be costly if forecasting errors are present. Secondly, follow-on conflicts may be created by a maneuver to resolve one conflict, against one of the primary rules for ATCos mentioned before. This is especially

likely in unstructured heavy-traffic scenarios, where one maneuver can trigger a domino of new conflicts [29].

7. "**Strategic Deconfliction and Path Planning**" [29]. Strategic deconfliction and path planning compute paths with few to no conflicts based on weather, traffic, origin, destination, and obstacles such as buildings or terrains. This approach also comes with a few setbacks. First, unlike in traditional aviation settings, there is no legal authority responsible for separation assurance in UAS operations. This raises the question of who establishes the ground rules, and whether they will be followed by everyone. [31] proposes a 4D contract consisting of route and crossing times for route deconfliction. Secondly, although 4D contracts could be a solution, they do not adhere to the unpredictability and flexibility needed for UAS operations, such as on-demand delivery or response to emergent situations. Furthermore, given the uncertainties mentioned before in UAS maneuvers, a pre-defined flight plan could not be the solution. Finally, such anticipated planning is very complex in terms of computational power [29].

Traditional Manned Aviation	**Small UAS Challenges**
Consistent vehicle performance	Diverse vehicle performance
Good maneuvering capability	Limited maneuvering capability
Performance robust in weather	Performance poor in weather
High situational awareness	Limited situational awareness
In situ decision making	High levels of autonomy
Highly reliable communications	Comm link failures common
Emerging, ADS-B, surveillance	ADS-B not scalable to dense ops
Air data and weather in situ	Little or no in situ weather data
Ground-based surveillance radars	No independent surveillance
Ground-based navigation aids	No navigational aids
Structured routes and airspace	Little airspace structure
High-altitude flight, good LOS	VLL, often cluttered LOS, clutter
NAS-wide ATC systems	No ATC services
Homogeneous O-D missions	Different mission types
Ops segregated from public	Ops integrated with public
Scheduled predictable ops	Unscheduled, unpredictable ops
SAA in time-tested and mature	DAA can fail in high-density ops
Simple separation criteria	Complex separation assurance
Clear lines of legal responsibility	Legal responsibility unclear

Table 2.1: Small UAS separation assurance challenges (Hunter et al., 2019) [29].

By analyzing the challenges and key drivers of the separation assurance problem for UAS, the authors in [29] suggest an autonomous ATC on structured airspace as an important component of the solution. Thus, the challenge of this research is to propose an autonomous controller that can keep aircraft separated both en route and at intersection points. Reinforcement Learning (RL) algorithms represent a possible framework to find optimal decisions in such an uncertain environment, by learning from interaction by trial-and-error without prior knowledge of the environment [64]. The next chapter focuses on reviewing the main Reinforcement Learning algorithms that can be used to learn an optimal controller technique based on a reward function.

3

Reinforcement Learning

Reinforcement Learning (RL) algorithms are well suited for tasks involving sequential decision making requirements [64]. In this chapter, the principal model-free RL techniques are studied. After a brief introduction to value-based and policy gradient methods, the combination of RL and Deep Learning (DL) is explored, as Deep Reinforcement Learning (DRL) algorithms have powered multiple solutions in the ATC space, both for single-agent and multi-agent problems [64].

3.1. Artificial Intelligence

Artificial Intelligence (AI) learning algorithms usually fall somewhere in the spectrum between supervised and unsupervised learning. Supervised learning algorithms learn from structured and labeled data to predict output labels for input features. These learning methods are trained using input-output pairs. Some supervised learning techniques are Support Vector Machines, Artificial Neural Networks, Logistic Regression, Naive Bayes, K-Nearest Neighbor, Random Forests and Decision Trees [15]. On the other hand, unsupervised learning algorithms learn from unstructured and unlabeled data to identify patterns in data. Some examples of unsupervised learning methods are data compression, clustering, and generative models [37]. In Reinforcement Learning (RL), no correct outputs are given, but only rewards. Thus, RL stands between supervised (more informative feedback) and unsupervised (less informative feedback) learning. An RL agent takes actions to maximize their goal (exploitation) while exploring the environment to collect rewards in unknown areas (exploration) [37]. Based on these rewards, the agent defines its optimal policy. This means that once learning is successfully completed, the agent has learned which action to take based on its current state to maximize the likelihood of achieving its goal.

3.2. Environment

A typical single-agent RL environment is defined as a finite Markov Decision Process (MDP). It is characterized by <X, U, f, ρ>, where X represents the state space, U the action space, $f : X \times U \times X \rightarrow [0,1]$ the transition probability function, and $\rho : X \times U \times X \rightarrow \mathbb{R}$ the reward function. In case the system is deterministic, the transition probability function can be replaced by $f : X \times U \rightarrow X$, and the reward function by $\rho : X \times U \rightarrow \mathbb{R}$. At each step k, the environment is in state $x_k \in X$. After observing its current state, the agent takes action $u_k \in u$ on the environment. According to the current state, the action, and the transition function f, the environment returns the next state $x_{k+1} \in X$ and a scalar reward $r_{k+1} \in \mathbb{R}$, depending on the reward function ρ [13].

Through trial and error on its interactions with the environment, the agent learns the action u_k to take at step k, given the current state x_k that maximizes its cumulative reward. This action choice is called a policy (π), and the maximum possible cumulative reward to be achieved from the current state is called the value function (V). Each agent explores the state space to learn which action to select at any given state that will maximize the value of the next state [37].

3.3. Policy

A policy is the method used by the agent to choose an action at any given state, by maximizing the cumulative reward. This means that immediate rewards are considered to be less important than long-term rewards. This way, the agent is forced to stick to its long-term goals, rather than to look for ways of maximizing its rewards on the next step [37].

Figure 3.1 represents the optimal policy for the following 3x3 grid problem: an agent is placed at a random cell in the grid. The origin of the grid is the top left corner cell, the x-axis points right and the y-axis points down. The reward for reaching the star is +10 and the reward for stepping into the crossed cell is -10. The state space for this problem is X = $\{0,1,2\} \times \{0,1,2\}$. The action space is U = {(1, 0), (-1, 0), (0, -1), (0, 1)}, representing movements to right, left, up and down respectively. For instance, if the agent is placed at the origin of the grid, there are two optimal actions, (1, 0), and (0, 1) that both lead to the same cumulative reward. In this case, the agent chooses randomly between them [37]. On the other hand, if the agent is placed in any other cell (except for the crossed and starred cells), there's only one optimal action to take, as represented in the policy diagram.

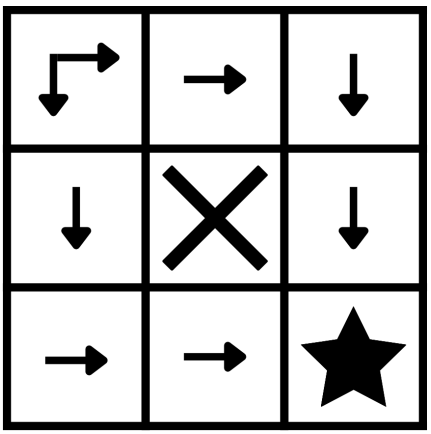


Figure 3.1: Example of an RL policy.

In case both the state space X and the action space U are quantized and discrete, the policy can also be represented in the form of a look-up table. As it is observable from Table 3.1, for each state of the environment, the agent can easily pick up the action that maximizes the cumulative reward according to the optimal policy π.

	0	1	2
0	(1, 0) or (0, 1)	(1, 0)	(0, 1)
1	(0, 1)	*	(0, 1)
2	(1, 0)	(1, 0)	*

Table 3.1: Look-up table of an RL policy.

One way to characterize RL algorithms is based on the path to an optimal solution. RL algorithms can be divided into off-policy and on-policy methods. Off-policy algorithms learn an optimal policy independently of the agent's action. This set of algorithms involves two policies, one acting on the environment - the behavior policy that depends on the exploration-exploitation settings, and another being updated but not being used - the target policy, whose updates do not depend on the actions taken by the agent. On the other hand, on-policy learning relies on using the policy that is being learned to act on the environment. [37].

3.4. Return

Figure 3.2 represents the traditional time-step in a Reinforcement Learning algorithm. Given the current state x_k and action taken by the agent u_k according to its policy π, the environment returns the next state x_{k+1} and a reward r_{k+1}. If the reward is provided with high frequency, it is called a dense reward. In case it is only provided a few times per episode, or even only at the end of it, it is called a sparse reward [37]. A good example of a case where a sparse reward is suitable is the Mountain Car Markov Decision Process (MDP). This problem was first introduced in [42]. This example consists of a car placed at the bottom of a valley. The goal is to strategically accelerate the car to the top of the right hill. If we consider the discrete problem, the state space is defined by position speed interval pairs and the action space by the different acceleration intervals that can be applied. In this case, a positive reward is only provided if the cart reaches its goal destination after bouncing back and forth in the valley to gather acceleration (a sparse reward).

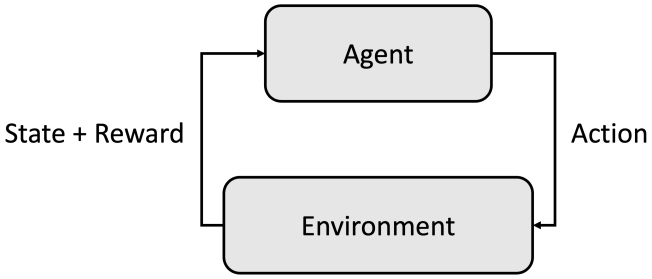


Figure 3.2: Example of interaction between RL agent and its environment.

The goal of an RL agent is to find a policy π that maximizes the discounted return (Equation 3.1) for every state. "The discount factor γ can be regarded as encoding an increasing uncertainty about rewards that will be received in the future, or as a means to bound the sum which otherwise might grow unbounded" [13]. The discount factor varies between 0 and 1. If γ is closer to 0, the agent will give more importance to immediate rewards, which may compromise future goals. On the other hand, if γ is closer to 1, the agent will be willing to delay the reward, as it gives more importance to future rewards. Therefore, a value of γ closer to 1 is usually preferable. Back to the comparison with supervised learning, rewards are the way of supervision in Reinforcement Learning, as they are the only feedback the agent receives from the environment [37].

$$R(x) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \tag{3.1}$$

3.5. Categorizing RL Algorithms

Reinforcement Learning algorithms can be separated into two different groups: model-based and model-free algorithms. Model-based algorithms require a model of the environment, information that can be very useful to find desired policies. However, modeling the environment can be very complicated most times. On the other hand, model-free algorithms can learn how to act on the environment without this external model [37].

"The first distinction is between model-free and model-based. Model-free RL algorithms can be further decomposed into policy gradient and value-based algorithms. Hybrids are methods that combine important characteristics of both methods" [37].

Model-free RL algorithms cannot rely on the dynamics of the model to make decisions. Thus, they have to run through a series of trajectories (τ) according to a policy (π) to gather experience through the collection of rewards from the environment. Model-free RL excels at handling complex system dynamics, whose behavior is not fully known beforehand [16]. There are three principal subcategories of methods in this set of algorithms: value-based, policy gradient and actor-critic methods [37].

Value-based algorithms typically use the Bellman equation, studied later in this chapter, to learn a Q-function, which means they learn a Q-value for every state-action pair. If we want to deal with high dimensional state spaces such as a raw image input from an Atari game, or the current state in a game of Go, Neural Networks (NNs) are very often used as Q-value function approximators [37].

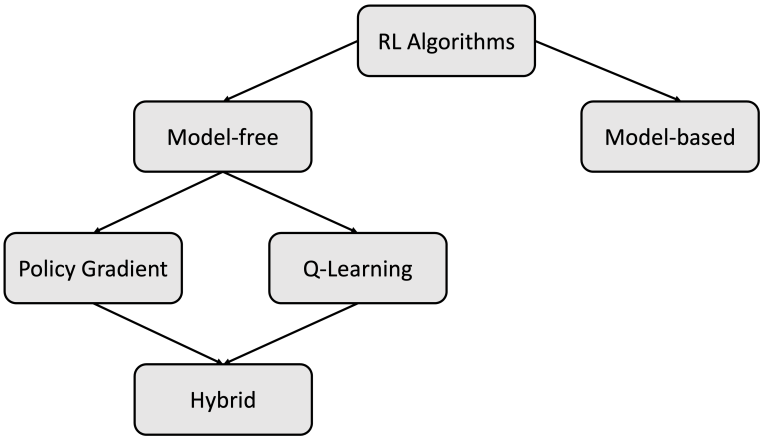


Figure 3.3: RL algorithm categories (adapted from [37]).

Policy gradient algorithms learn actions from a parameterized policy, via updates to the parameters in the direction of the improvements. Policy gradient methods are useful when the action space is very large or continuous, a task that cannot be achieved by value-based methods relying on value-action functions Q(s, a) [37].

Lastly, actor-critic algorithms are situated between value-based and policy gradient algorithms. These "algorithms are on-policy policy gradient algorithms that also learn a value function (generally a Q-function) called a critic to provide feedback to the policy, the actor" [37].

On the other hand, in model-based RL, the mathematical model of the environment is known, which means that the next states and rewards can be determined without any interaction with the environment [37]. This makes these algorithms much more sample-efficient, meaning that the amount of data required to fit the model is lower. The combination of model-free and model-based approaches has also shown promising results in rapid, efficient, and secure real-world task learning, with a reduced amount of experience or demonstration data, required [16].

Despite all the benefits of model-based RL, it is a challenge to acquire an accurate model of the environment for many complex real-world applications [37].

3.6. Value-Based Methods

3.6.1. Value Function

"The value function represents the long-term. quality of a state. This is the cumulative reward that is expected in the future if the agent starts from a given state" [37]. This means that the following relation can be established between the value function (V) and the discounted return (R):

$$V^{\pi}(x) = \mathbb{E}[R^{\pi}(x)] \tag{3.2}$$

3.6.2. Q-Function

Although it would be very memory efficient to store only the discounted returns for every given state, deriving a greedy policy from a value function is nontrivial. "The task of the agent is therefore to maximize its long-term performance (return), while only receiving feedback about its immediate, one-step performance (reward)" [13]. This can be achieved with the Q-function [17].

$$\begin{aligned} R^{\pi}(x_0) &= \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \\ &= \rho(x_0, \pi(x_0)) + \sum_{k=1}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \\ &= \rho(x_0, \pi(x_0)) + \gamma \sum_{k=0}^{\infty} \gamma^k \rho(x_{k+1}, \pi(x_{k+1})) \\ &= \rho(x_0, \pi(x_0)) + \gamma R^{\pi}(x_1) \end{aligned} \tag{3.3}$$

The Q-function makes the first action a free variable u_0.

$$Q^{\pi}(x_0, u_0) = \rho(x_0, u_0) + \gamma R^{\pi}(x_1) \tag{3.4}$$

The difference between a value function and a Q-function is that the value function indicates the expected return given a state, while the Q-function provides the expected return for a value action pair.

$$V^{\pi}(x) = \mathbb{E}[R_k | x_k = x, \pi] \tag{3.5}$$

$$Q^{\pi}(x, u) = \mathbb{E}[R_k | x_k = x, u_k = u, \pi] \tag{3.6}$$

The goal is to find the optimal Q-function. For this, the Bellman equation is used.

$$\begin{aligned} Q^{\pi}(x_0, u_0) &= \rho(x_0, u_0) + \gamma R^{\pi}(x_1) \\ &= \rho(x_0, u_0) + \gamma[\rho(x_1, \pi(x_1)) + \gamma R^{\pi}(x_2)] \\ &= \rho(x_0, u_0) + \gamma Q^{\pi}(x_1, \pi(x_1)) \end{aligned} \tag{3.7}$$

This means that Q_{pi} is improved using bootstrapping, i.e., using the current estimate of the value-action function. Considering that $x_1 = f(x_0, u_0)$, the Bellman equation for Q^{π} can be written as:

$$Q^{\pi}(x, u) = \rho(x, u) + \gamma Q^{\pi}(f(x, u), \pi(f(x, u))) \tag{3.8}$$

Considering a deterministic system. The goal is to derive a greedy policy from the optimality solution Q*. Thus, we take Bellman optimal equation (for Q*).

$$Q^*(x, u) = \rho(x, u) + \gamma max_{u'} Q^*(f(x, u), u') \tag{3.9}$$

Where u' is the action for the next state. As long as we have an optimal Q*, an optimal policy can be computed by choosing the action with the maximum E value for every state [13].

$$\pi^*(x) = argmax_u Q^*(x, u) \tag{3.10}$$

"A policy that maximizes a Q-function in this way is said to be greedy in that Q-function. So, an optimal policy can be found by first determining Q^* and then computing a greedy policy in Q^*" [13].

3.6.3. Q-Learning

Q-learning, introduced by Watkins in 1989 is a simple, yet powerful off-policy value-based model-free RL algorithm that approximates the optimal control dynamic programming. This algorithm can be viewed as asynchronous dynamic programming [66]. It builds upon the previous explanation of the Q-function. If we take the Bellman Equation 3.9 at some (x, u) and turn it into an iterative update substituting the transition function f and the reward function ρ by a transition sample $(x_k, u_k, x_{k+1}, r_{k+1})$:

$$Q(x_k, u_k) \leftarrow r_{k+1} + \gamma max_{u'} Q(x_{k+1}, u') \tag{3.11}$$

Finally, we make the update incremental by adding a learning rate $\alpha_k \in [0,1]$ multiplied by the temporal difference error term:

$$Q(x_k,u_k) \leftarrow Q(x_k,u_k) + \alpha_k \cdot [r_{k+1} + \gamma max_{u'} Q(x_{k+1},u') - Q(x_k,u_k)] \tag{3.12}$$

The convergence of Q-Learning for deterministic MDP is achieved if each state-action pair is visited an infinite number of times. This means that the agent "...can learn the Q function (and hence the optimal policy) while training from actions chosen completely at random at each step, as long as the resulting training sequence visits every state-action transition infinitely often" [39]. This is accomplished, in practice, by a trade-off between exploration (choosing actions at random) and exploitation (choosing greedy actions according to the current knowledge). Exploration allows the agent to explore new state-action pairs, while exploitation lets it optimize the state-action pairs that have already been explored before, making the process more sample-efficient.

In practice, the balance between exploration and exploitation is accomplished using the ϵ-greedy strategy, where $\epsilon \in [0,1]$ sets the exploration probability.

$$u_k = \begin{cases} argmax_{u'} Q(x_k,u') & \text{with probability} \quad (1-\epsilon_k) \\ \text{a random action} & \text{with probability} \quad \epsilon_k \end{cases} \tag{3.13}$$

3.6.4. SARSA

SARSA (state-action-reward-state-action) is another value-based RL algorithm similar to Q-Learning. However, in the case of SARSA, the method is on-policy. The update rule is given by:

$$Q(x_k,u_k) \leftarrow Q(x_k,u_k) + \alpha_k \cdot [r_{k+1} + \gamma Q(x_{k+1},u_{k+1}) - Q(x_k,u_k)] \tag{3.14}$$

Instead of considering the action with the highest possible value (that might be different from the actual agent action due to exploration) for the temporal difference (TD) update, SARSA considers the actual action performed by the agent.

3.7. Policy Gradient Methods

Despite being able to represent complex policies to control agents, value-based methods present limitations when dealing with a very high number of actions of continuous action spaces. In such cases, when the model cannot be represented using a low dimensional action space, Policy Gradient (PG) algorithms exhibit great potential [37].

Policy gradient algorithms, introduced by Sutton in 1999, learn to represent a parameterized policy π_θ as a function approximator, by maximizing the expected return $E[R|\theta]$ with respect to the policy parameters θ. [61] [3].

In the case of discrete action space, the parameterized policy yields a value (z) for each action. These values are then normalized and converted to probabilities of taking each action using the softmax function, such as the sum of the probabilities of every possible action is equal to 1 [37]. By sampling at random over the probability distribution, the exploration-exploitation balance is achieved.

$$\pi_\theta(u|x) = \frac{e^{z(x,u)}}{\sum_i e^{z(x,u_i)}} \tag{3.15}$$

3.7.1. Policy Gradient

Considering θ the parameters of a given policy, the goal of an RL agent is, as studied before, to maximize the expected return over the trajectory. Thus, the objective function is given as:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{3.16}$$

The parameters are optimized in the direction of the gradient $\nabla J(\theta)$ using gradient ascent, similar to the classical gradient descent, but looking to find the maximum in the objective function. When the maximum

is found, π_θ generates trajectories (τ) with the highest possible returns. According to the policy gradient theorem, introduced by Sutton et al., it is possible to calculate the derivative of the objective function with respect to the policy's parameters, without calculating the derivative of the state distribution. [61].

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(u|x) Q_{\pi_\theta}(x,u)] \tag{3.17}$$

One can estimate the expectation by sampling trajectories from the policy.

$$\nabla J(\theta) \approx \frac{1}{N} \sum_{i_0}^{N} [\nabla_\theta log \pi_\theta(u_i|x_i) Q_{\pi_\theta}(x_i,u_i)] \tag{3.18}$$

Using gradient ascent, the policy can be optimized using parameter update with learning rate α. The plus sign in the formula of gradient ascent means the objective is being maximized (i.e. the parameters are being updated in the direction of the gradient) [37].

$$\theta = \theta + \alpha \nabla_\theta J(\theta) \tag{3.19}$$

3.7.2. REINFORCE

REINFORCE is the simplest policy gradient algorithm. The challenges raised by the simplicity of this algorithm will be tackled in the next sections, where we will develop an understanding of two more RL algorithms: REINFORCE with baseline and actor-critic (AC) [37].

The REINFORCE algorithm is an on-policy method, collecting experience for sampling based on its interactions with the environment. To compute the gradient of the objective function, we have to define how to calculate the action-value function. In REINFORCE, $Q_{\pi_\theta}(x,u)$ is estimated using Monte Carlo (MC) returns. "Monte Carlo methods estimate the expected return from a state by averaging the return from multiple rollouts of a policy" [3]. This means that Equation 3.17 can be replaced by:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(u_k|x_k) R_k] \tag{3.20}$$

Where R_k is the MC return at time step k. This is also called the reward to go, and is defined as [37]:

$$R_k = \sum_{k'=k}^{K} \gamma^{k'-k} \rho(x_{k'}, u_{k'}) \tag{3.21}$$

3.7.3. REINFORCE with Baseline

The simplest REINFORCE algorithm has a problem of variance, that increases with the length of the trajectory. This happens because of the stochasticity of the policy, which means that executing the same policy multiple times may lead to different outcomes, and thus, the value assigned to a certain state-action pair may not be correct [37].

To reduce the variance and improve the stability and performance of the REINFORCE algorithm, a baseline, b, is introduced [37]. The gradient of the objective function in the REINFORCE with baseline algorithm is then given as:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(u_k|x_k)(R_k - b)] \tag{3.22}$$

The simplest way of defining baseline b is by subtracting the average return. However, this way, the baseline could be conditioned on the state.

$$b = \frac{1}{N} \sum_{n=0}^{N} R_n \tag{3.23}$$

A better alternative is to estimate the value function V^{π_θ}, which is, on average, the return obtained by following the policy π_θ.

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(u_k|x_k)(G_k - V^{\pi_\theta}(x_k))] \tag{3.24}$$

To learn the value function, the best strategy is to fit an ANN with MC estimates.

$$V_w^{\pi_\theta}(s) = \sum_{k'=k}^{K} \gamma^{k'-k} \rho(x_{k'}, u_{k'}) \tag{3.25}$$

Where w represents the weights of the Neural Network (NN). The NN is trained on the same data used for learning π_θ, and thus, it does not require any extra interaction with the environment. MC estimates will serve as the target values to optimize the network's parameters [37].

3.7.4. Actor-Critic

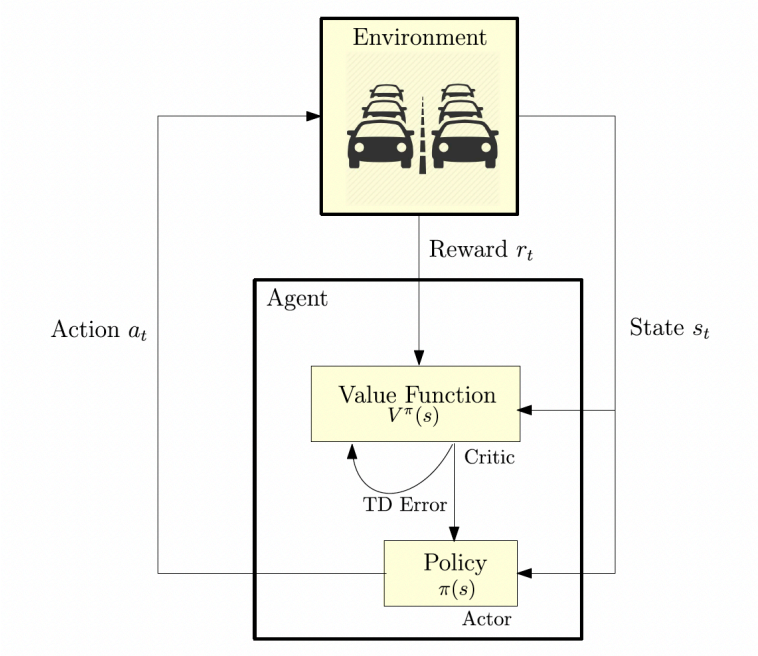


Figure 3.4: Actor-critic algorithm layout [25].

As we have seen before, REINFORCE had a high variance shortcoming. While adding a baseline improved it, the convergence of the algorithm is very slow. To speed up learning, the bootstrap method can be used. As in Q-Learning, we can use subsequent state values in the estimate of returns. The hybrid policy-gradient algorithm that uses this is called actor-critic (AC), introduced by Konda et al. in 1999, and it combines the benefits of value-based and policy gradient methods [37]. While most RL methods rely solely on a parameterized policy or a value-function approximation, AC methods aim at combining the strong points of both methods. [35].

Using one-step bootstrapping, the action-value function is defined as:

$$Q(x,u) = \rho(x,u) + \gamma V(x') \tag{3.26}$$

Where x' is the next state. In AC algorithms, the actor controls a parameterized policy π_θ and the critic represents a parameterized state-value function (V_w) [37]. "The actor (policy) learns by using feedback from the critic (value function)" [3]. Using a one-step AC update:

$$\theta = \theta + \alpha(\rho_k + \gamma V_w(x') - V_w(x)) \nabla_\theta log \pi_\theta(u_k|x_k) \tag{3.27}$$

As opposed to MC methods, TD learning has low variance but high bias. Thus, the combination of both algorithms is most of the time the best solution. To acknowledge this trade-off, an n-step return can be used in AC. Using the n-step MC return $G_{k:k+n}$, the actor parameters' update can be rewritten [37].

$$\theta = \theta + \alpha(R_{k:k+n} + \gamma^n V_w(x_{k+n}) - V_w(x_k))\nabla_\theta log\pi_\theta(u_k|x_k) \tag{3.28}$$

The quantity $R_{k:k+n} + \gamma^n V_w(x_{k+n}) - V_w(x_k)$ can be defined as an estimate for the advantage function:

$$A(u,x) = Q(u,x) - V(x) \tag{3.29}$$

Usually, this function is easier to learn; "more intuitively, it is easier to learn that one action has better consequences than another than it is to learn the actual return from taking the action" [3].

To optimize the parameters w of the critic, Stochastic Gradient Descent (SGD) with Mean Squared Loss (MSE) is used. The target values y_i are computed using the following formula:

$$y_i = R_{t:t+n} + \gamma^n V_w(x_{k+n}) \tag{3.30}$$

Finally, the MSE loss function to optimize is defined as:

$$L = \frac{1}{2}\sum_i (V_w(x_i) - y_i)^2 \tag{3.31}$$

3.8. Deep Reinforcement Learning

So far, we have been mainly studying RL problems where the state space is quantized. However, for very high-dimensional problems, standard RL algorithms cannot scale the computation of value and policy functions for all states as the size of the table grows exponentially. Moreover, even if we had no memory constraints, this table could only be sparsely filled, and information could not be efficiently propagated between state-action pairs. Deep Reinforcement Learning (DRL) is very efficient at processing this curse of dimensionality. Classical DRL approaches use Convolutional Neural Networks (CNNs) as function approximators to compute optimal value-action functions [25] [37] [3].

3.8.1. Deep Q-Network

The simpler DRL method is Deep Q-Network (DQN). In Deep Q-learning, a Q value would be stored for every state-action pair. As mentioned before, this approach is not scalable to high-dimensional or continuous state spaces such as raw image inputs. Traditionally, DQN uses CNNs, taking a raw image as an input and directly predicting the Q-values for every possible action, without the need to store them. The only memory being used in this case only has to store the network's weights, which are shared across all the different stages of the environment. This makes the process much more memory-efficient [37].

In DQN, the Q-values are parameterized in θ, the weights of the neural network:

$$Q_\theta(x,u) \tag{3.32}$$

The objective of DQN is to optimize the Q-function approximator neural network's weights θ, in order to attain the best possible estimations. However, since the optimal Q-function is not provided to be used as a target in training the network, the best solution is to minimize the Bellman error for one step $r + \gamma max_{u'} Q_\theta(x',u') - Q_\theta(x,u)$, where x' and u' refer to the next step. Thus, the parameters θ of the NN can be optimized according to the following equation:

$$\theta \leftarrow \theta - \alpha[r + \gamma max_{u'} Q_\theta(x',u') - Q_\theta(x,u)]\nabla_\theta Q_\theta(x,u) \tag{3.33}$$

Where $\nabla_\theta Q_\theta(x,u)$ is the partial derivative of the Q-function with respect to the neural network's parameters. Although this update resembles tabular Q-learning, it does not yield a good approximation. To improve DQN, the Mean Squared Error (MSE) loss is used. Furthermore, rather than an online update (i.e. updating the parameters for every sample collected during training), a batch update is chosen. [37]. This is called experience

replay, and stores $(x_k, u_k, r_{k+1}, x_{k+1})$ in memory, such that sample batches can be selected at random to train the network. There are two main advantages to using a fixed memory buffer (it eliminates older samples to give place to new ones): first, because mini-batches are sampled from a large pool, temporal correlations that tend to deprecate RL agent's learning are broken and these samples can be considered independent and identically distributed (IID); secondly, the use of batches improves the efficiency of training by reducing the required number of interactions with the environment and the variance of learning updated [25] [3].

$$L(\theta) = \mathbb{E}_{(x,y,r,x')}[(y_i - Q_\theta(x_i, u_i))^2] \tag{3.34}$$

Where y is the Q-target value. The network's parameters θ are updated according to the gradient descent formula [37]:

$$\theta = \theta - \alpha \nabla_\theta L(\theta) \tag{3.35}$$

There is yet another cause of instability in the prior formulation of Deep Q-Network: the non-stationarity of the Q-Learning algorithm. As we have formulated the problem, the network that is updated at every step is the same which predicts the target values y_i. This is a source of instability in the learning process. To tackle this issue, the concept of a separate target network was introduced. The target network is used to predict the target value, but its weights are only updated in N-step intervals, contrary to the online network, whose weights are updated every single step. Usually, the value of N varies between 1000 and 10000 steps, meaning that with every fixed number of steps the parameters of the online network are copied to the target network. For all the other steps, the weights of the target network are kept frozen [3] [37]. With this in mind, the Q-target value can be defined as:

$$y_i = r_i + \gamma max_{u'_i} \overline{Q}_{\theta'}(x'_i, u'_i) \tag{3.36}$$

Where θ' are the parameters of the target network, θ are the parameters of the online network, and $\overline{Q}$ is the Q-function of the target network. Therefore, the loss function becomes:

$$L(\theta) = \mathbb{E}_{(x,y,r,x')}[(r + \gamma max_{u'} \overline{Q}_{\theta'}(x', u') - Q_\theta(x, u))^2] \tag{3.37}$$

Deep Q-Network was introduced as the first RL model to learn optimal policies directly from raw image inputs. Mnih et al. developed this method to play Atari arcade games and achieved super-human performance in some of them. The architecture implemented in [41] takes an 84x84x4 input, a grayscale image with 4 channels. Each channel represents one of the last 4 frames saved in memory. This allows the agent to not only know the state of the game but also its recent evolution, overcoming the partial observability problem of Atari games. If we consider the game of pong, it is not possible to know if the ball is traveling left or right based only on a single frame [37]. After a series of convolutions and nonlinearities are applied to the input, the output layer contains a prediction head for each possible action. This avoids having to run a different network for each action, given that one single network can predict the Q values for all discrete actions [41].

3.8.2. Double Deep Q-Network

Double DQN (DDQN) was first introduced by Hasselt et al. [62] as a more stable and reliable learning method to tackle the overestimation problem found in DQN that led to sub-optimal policies. Double Q-Learning appears to tackle the overestimation challenge found in Q-Learning. Since the maximum operator is used to choose and evaluate actions, it is more likely to compute overoptimistic Q-values [62]. In classical double Q-Learning, first introduced by Hasselt [23], two Q-functions are learned Q^A and Q^B. Each Q-function is updated with a value from the other Q-function trained on the same problem but with a different set of experiences. The greedy action is derived by averaging both Q-values [23]. DDQN aimed at reducing overestimations by decoupling the maximum operation into action selection and action evaluation. The online network in the DQN architecture is used to evaluate the greedy policy, while the target network is leveraged to estimate its value [62]. DDQN changes only the target calculation of DQN that we have previously discussed in Equation 3.36 [65].

$$y_i^{DDQN} = r_i + \gamma Q(x', max_{u'} Q(x', a'; \theta_i); \theta^-) \tag{3.38}$$

Where θ^- are the weights of the target network and θ_i the current weights of the online network. The update of the target network remains periodic as in DQN [62].

3.8.3. Prioritized Experience Replay

Prioritized Experience Replay (PER) built on top of DDQN was introduced by Schaul et al. [51]. As discussed before, experience or memory replay allows DRL agents to use experiences collected in the past during the learning process. Classically, the mini-batches used for training the online network were sampled uniformly from the experience buffer. Prioritized experience replay is aimed at improving learning efficiency by replaying state transitions with high expected learning progress more frequently. This importance of a given transition is measured by the TD error (temporal-difference error). Based on this priority metric, this method proposes a stochastic selection method, combining greedy prioritization with uniform random sampling. Using PER, the probability of sampling transition i is computed as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.39}$$

Where α determines how much prioritization is given, and thus, $\alpha = 0$ corresponds to uniform sampling. p_i is the priority computed by the TD-error of transition i [51]. Another concept introduced in this paper is the importance-sampling (IS) weights. Since the estimation of the expected value relies on the distribution of transitions, and prioritized experience replay changes this stochastic distribution with non-uniform sampling probabilities, inducing a bias, IS weights were introduced to compensate for this.

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \tag{3.40}$$

Where β is a hyper-parameter that represents the amount of importance-sampling correction. If $\beta = 1$, the non-uniform sampling probabilities $P(i)$ are fully compensated. Prioritized replay was found to speed up learning by a factor of 2 and lead to a new benchmark on the Atari games [51].

3.8.4. Dueling Deep Q-Network

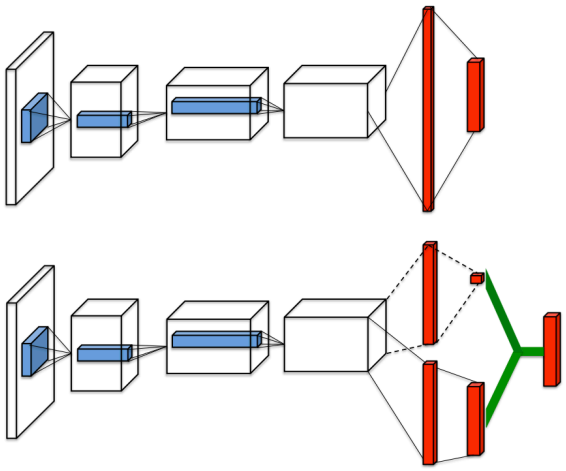


Figure 3.5: Stream Deep Q-Network (top) vs. Dueling Deep Q-Network (bottom) [65].

The idea behind the development of the Dueling Network Architecture in [65] is that in typical RL problems, it is not necessary to estimate the value of every possible choice of action. The authors provide the example of the Enduro game setting, where it is only important to know which action to take when a collision is imminent. They designed the dueling network, that changes the architecture of DQN by adding two modules on top of the convolutional layers, instead of only fully connected layers. These streams are built to have the capacity of computing separate value and advantage functions. This has the benefit that each of the modules will pay attention to different features of the input. For a race environment, the value streams learn to pay attention to the road, while the advantage streams pay attention to cars immediately in front of the agent. Both of these streams are combined to yield a value for the Q-function for every eligible action. Since the output of the network is a set of Q-values, it can be trained using one of the algorithms studied before, such

as DDQN or SARSA. Considering the Dueling Network settings, a parameterized estimate of the Q-function is computed as:

$$Q(x,u;\theta,\alpha,\beta) = V(x;\theta,\beta) + (A(x,u;\theta,\alpha) - \frac{1}{|U|}\sum_{u'} A(x,u';\theta,\alpha)) \tag{3.41}$$

Where α and β are the parameters of the two branches of FC layers, θ represents the parameters of the convolutional layers, and U is the finite set of actions. The $A(x,u;\theta,\alpha)$ vector is $|U|$-dimensional [65]. The combination of this dueling network architecture with the previously discussed PER is a state-of-the-art discrete RL technique [3].

3.8.5. Trust Region Policy Optimization

To improve the stability of the policy-gradient algorithms, Natural Policy Gradient (NPG) was introduced by Kakade in 2001 to represent the steepest descent direction based on the structure of the parameters, i.e., towards the greedy optimal action [32]. To understand NPG, one has first to understand two underlying concepts: first, the Fisher Information Matrix (FIM) is the covariance of an objective function and the Kullback-Leibler (KL) divergence is a metric that measures how different two distributions are. The higher this value, the larger the KL divergence metric. The goal is to limit the distance between the old and new distributions in a model update, which is why the KL distribution is used [37].

$$D_{KL}(P||Q) = -\sum_{x\in X} P(x) log(\frac{Q(x)}{P(x)}) \tag{3.42}$$

The "FIM defines the local curvature in the distribution space by using the KL divergence as a metric" [37]. The direction and length of the optimization step are adjusted with respect to the second-order derivative of the KL divergence and the first-order derivative of the objective function in order to stabilize the convergence of the objective [37].

$$\theta \leftarrow \theta + \alpha F^{-1} \nabla_\theta J(\theta) \tag{3.43}$$

Despite the usefulness of the NPG, computing F^{-1} is very computationally demanding when using modern DNNs. Trust Region Policy Optimization (TRPO) is an on-policy actor-critic method, proposed in 2015 by Schulman et al. It uses multiple approximations to compute the NPG, aiming at stabilizing the training of a DNN policy [37]. This algorithm has proven to be scalable and able to optimize complex policies such as swimming, hopping, walking, and playing Atari games from raw image inputs [52].

The algorithm uses a surrogate loss function constrained with the distance (KL divergence) between the new policy and the old policy, representing a trust region. This means that the optimization steps are bounded to a trust region where the cost function approximation is valid [3] [52]. In practice, the constraint problem on the policy parameters in TRPO is defined as follows:

$$\begin{aligned} &\text{maximize}_\theta \quad J_{\theta old}(\theta) \\ &\text{subject to} \quad \overline{D}_{KL}(\theta_{old},\theta) \leq \delta \end{aligned} \tag{3.44}$$

Where $J_{\theta old}(\theta)$ is the objective surrogate function, $\overline{D}_{KL}(\theta_{old},\theta)$ is a heuristic approximation which considers the average KL divergence between the old parameters θ_{old} and the new ones θ, and δ is a coefficient of the constraint [37]. The objective surrogate function is minimized with respect to the new parameters while using the state distribution from the old policy. To accomplish this, the authors use an importance sampling estimator. This is needed because the trajectory is sampled using the old policy, but we want to compute the distribution of the new one. The final optimization problem solved in TRPO is defined next [52].

$$\begin{aligned} &\text{maximize}_\theta \quad \mathbb{E}_{x\sim p_{\theta old}, x\sim q}[\frac{\pi_\theta(u|x)}{q(u|x)} Q_{\theta old}(x,u)] \\ &\text{subject to} \quad E_{x\sim \rho_{\theta old}}[D_K L(\pi_{\theta old}(\cdot|x)||\pi_\theta(\cdot|x))] \leq \delta \end{aligned} \tag{3.45}$$

Where p is the distribution of the initial state x_0, q is the sampling distribution, and $\pi(\cdot|x)$ indicates the actions distributions conditioned on the state x [37]. From the previous equation, we have to substitute the expectations by an empirical average using a batch of samples and the Q value by an empirical estimate [52]. An example of a possible sampling method, typically used for policy-gradient estimation is the single path. Using this scheme, a sequence of state are collected by sampling $s_0 \sim \rho_0$, and simulating a trajectory for T time steps using the old policy $\pi_{\theta old}$, which will give us a sequence of state action pairs. Thus, $q(u|x) = \pi_{\theta old}(x|u)$. The old Q-values $Q_{\theta old}(x,u)$ are estimated based on the sum of future discounted rewards along the trajectory [52].

3.8.6. Generalized Advantage Estimation

Policy gradient methods are usually very sample-consuming, requiring a large amount of experience to yield good learning results. To address this challenge, Schulman et al. proposed an exponentially-weighted estimator of the advantage function, reducing the variance of policy gradient estimates, but introducing some bias [53]. Denoting the estimate of the discounted advantage function at time step k $\hat{A}_k$, the policy gradient estimator can be written as:

$$\hat{g} = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=0}^{\infty}\hat{A}_k^n \nabla_\theta log\pi_\theta(u_k^n|x_k^n) \tag{3.46}$$

Where n indexes over a batch of episodes. The generalized advantage estimator, $GAE(\gamma,\lambda)$, is written as an exponentially-weighted average.

$$\hat{A}_k^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{k+l}^V \tag{3.47}$$

Where $\delta_k^V = r_{k+1} + \gamma V(x_{k+1}) - V(x_k)$. The parameter λ is usually set between 0 and 1 and regulates the trade-off between bias and variance. Finally, a value function estimator has to be defined. In case a nonlinear function approximator is used to represent the value function, the most straightforward approach to finding its parameters is to solve a nonlinear regression problem.

$$\text{minimize}_\phi \sum_{n=1}^{N}||V_\phi(x_n) - \hat{V}_n||^2 \tag{3.48}$$

Where $\hat{V}_k$ is the discounted sum of rewards as we have seen before, and n indexes all time steps in a trajectory batch. The authors added a KL divergence constraint in their trust region (TR) method to optimize the value function to avoid overfitting the parameters to the most recent batch of data [53]. Using TRPO together with GAE is a state-of-the-art continuous RL technique [3].

3.8.7. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy, actor-critic algorithm, introduced by Shulman et al. in 2017. It is based on the same ideas as the previously studied TRPO but takes a more simple, general and sample-efficient approach [37]. This work proposed a novel objective function that allows for multiple epochs of mini-batch updates. The experiments done by the authors have shown that PPO outperforms other online policy gradient algorithms in tasks such as robotic locomotion and playing Atari games [54].

The main concept behind PPO is a clipped surrogate objective function. Instead of constraining the objective as we have seen in TRPO, PPO clips it when it moves away to ensure that optimization steps are not very large [37]. The main objective function proposed by the authors is given as follows:

$$L^{CLIP}(\theta) = \mathbb{E}_{x\sim p_{old}, u\sim\pi_{old}}[min(r_k(\theta)A_k, clip(r_k(\theta), 1-\epsilon, 1+\epsilon)A_k)] \tag{3.49}$$

Where ϵ is a hyper-parameter and $r_k(\theta)$ is a probability ratio:

$$r_k(\theta) = \frac{\pi_\theta(u_k|x_k)}{\pi_{\theta_{old}}(u_k|x_k)} \tag{3.50}$$

The goal of this objective is to prevent r_k from getting outside of the interval $[1-\epsilon, 1+\epsilon]$, by penalizing changes to the policy that move r_k away from 1 [54]. The practical implementation of the PPO algorithm uses a truncated version of the GAE method we have discussed before to estimate the advantage function. At every time step, multiple trajectories are collected within a time span, and both the policy and the critic are updated multiple times using mini-batches [37].

3.8.8. Deep Deterministic Policy Gradient

Policy gradient methods, such as TRPO, are not sample efficient in finding an optimal policy, due to their on-policy nature. In this section, a new set of off-policy actor-critic algorithms is introduced [37]. All policy gradient algorithms we have studied so far consider stochastic policies $\pi_\theta(u,x) = P[u|x,\theta]$ that gives a probability of taking a certain action given the state x and the policy parameters θ. On the other hand, deterministic policy gradient algorithms use deterministic policies $u = \mu_\theta(x)$. Since the deterministic policy gradient (DPG) integrates over the state space only, it is more sample efficient than the stochastic case that integrates over both states and actions [56].

Finding an off-policy algorithm that is able to learn desired policies in high-dimensional environments is difficult. We have studied DQN before, which is able to do this for discrete action settings. If we want to use DQN in continuous environments, we must discretize the action space. However, for tasks that need more precise control, this strategy is not scalable. The goal of deterministic policy gradient is to learn a deterministic AC, i.e., a deterministic $\mu_\theta(x)$ policy that approximates $argmax_u Q(x,u)$. A deterministic policy makes it possible to work with continuous state-action environments by avoiding a global maximization at every time step [37] [56].

According to the deterministic policy gradient theorem, a deterministic policy is defined as $\mu_\theta : X \rightarrow U$, with parameter vector $\theta \in \mathbb{R}^n$. We define the objective function as an expectation [56].

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{x\sim p^\mu}[\nabla_\theta \mu_\theta(x) \nabla_u Q_\phi(x,u)|_{u=\mu_\theta}] \tag{3.51}$$

To make DPG off-policy, the objective function has to be slightly changed to take into account the value function of the target policy, parameterized by θ, averaged with respect to the state distribution of the behavior policy, parameterized by β [56].

$$\nabla_\theta J_\beta(\mu_\theta) = \mathbb{E}_{x\sim p^\beta}[\nabla_\theta \mu_\theta(x) \nabla_u Q_\phi(x,u)|_{u=\mu_\theta}] \tag{3.52}$$

Where β is the behavior or exploratory policy that generates the trajectories. In the off-policy deterministic actor-critic algorithm, the critic uses Q-learning updates to minimize the Bellman error [56].

$$\begin{aligned} \delta_k &= r_k + \gamma Q^w(x_{k+1}, \mu_\theta(x_{k+1})) - Q^w(x_k, u_k) \\ w_{k+1} &= w_k + \alpha_w \delta_k \nabla_w (x_k, u_k) \\ \theta_{k+1} &= \theta_k + \alpha_\theta \nabla_\theta \mu_\theta(x_k) \nabla_u Q^w(x_k, a_k)|_{u=\mu_\theta(x)} \end{aligned} \tag{3.53}$$

DPG is in practice very unstable when implemented with DNNs. Deep Deterministic Policy Gradient (DDPG), introduced by Lillicrap et al. in 2015 was the first deterministic actor-critic algorithm that uses DNNs to learn both the actor and the critic. This is a model-free, actor-critic, off-policy algorithm, extending DQN to the continuous action RL setting [37] [36].

To train neural networks in RL settings, one has to ensure the samples are IID, which does not hold true with sequential experiences. Moreover, to increase training efficiency, learning should happen through mini-batches, rather than online. Similarly to DQN, the DDPG authors decided to use a replay buffer to train the actor and the critic by sampling a mini-batch from the buffer using a uniform distribution [36].

To tackle the instability faced when directly using NNs with Q-learning, due to the fact that a single network is being updated while computing the target values, the authors also used a separate target network inspired by DQN. However, they adapted it for actor-critic settings, using two online networks, one for the critic and the other for the actor, and two target networks. Moreover, they used "soft" target updates, instead of directly copying the weights from the online network to the target network, as it happens in DQN every N steps [36].

A "soft" update means that the actor's parameters θ' are partially updated every time step with the current parameters θ of the online network, as defined in Equation 3.54.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{3.54}$$

With $\tau << 1$. The main benefit of this approach is the increased stability added to the algorithm. The same happens for the critic online and target networks parameterized by ϕ and ϕ' respectively [37]. A third challenge faced when learning from low-dimensional feature vector observations is that different components can have units in different ranges. This may make it more difficult for the network to adjust the weights effectively. The authors of DDPG borrowed the concept of batch normalization from deep learning and applied it to ensure that each dimension across samples in a mini-batch is normalized, making learning faster and more efficient [36].

To overcome the challenge of exploring a continuous action space environment, DDPG takes advantage of its off-policy nature, introducing an exploratory policy β_θ that is built by adding noise sampled from a noise process N [37] [36].

$$\beta_\theta(x_k) = \mu_\theta(x_k) + N \tag{3.55}$$

In summary, the DDPG algorithm consists of three steps performed repeatedly: first, the exploratory policy β_θ collects experience, i.e. state observations and rewards from the environment and stores them in memory (replay buffer). Secondly, the critic and the actor are updated using a mini-batch sampled from the replay buffer. The critic is updated using MSE loss between the values predicted by the online critic Q_ϕ, and the target values, y_i, predicted using the target critic $Q_{\phi'}$ [37].

$$y_i = r_i + \gamma Q(x', \mu'(x'|\theta_{\mu'})|\theta_{Q'}) \tag{3.56}$$

Where μ' is the target actor-network and Q' is the target critic network [36]. The actor is updated using Equation 3.52. Lastly, the target network parameters are updated using the "soft" update explained before [37]. The experiments performed by the authors of DDPG have shown that the algorithm was able to solve more than 20 simulated physics tasks and to learn end-to-end policies directly from raw image inputs [36].

3.8.9. Twin Delayed Deep Deterministic Policy Gradient (TD3)

Twin Delayed Deep Deterministic Policy Gradient (TD3) is a state-of-the-art off-policy actor-critic algorithm that builds upon DDPG to make it more stable, performant, and less sensitive to hyper-parameters [37]. This word proposed a clipped double Q-learning in an actor-critic setting to tackle the overestimation bias we have discussed when introducing DDQN [20].

In DDQN, the authors used two neural networks, one for choosing the action and the other to calculate its value. Although this idea has been successfully applied for value-based settings, it does not work in actor-critic because policies change very slowly. Thus, the online and target networks are too similar to estimate different values. Instead, the authors of TD3 propose the use of a pair of actors $(\pi_{\theta_1}, \pi_{\theta_2})$, and a pair of critics (Q_{ϕ_1}, Q_{ϕ_2}), where π_{θ_1} is updated with respect to Q_{ϕ_1} and π_{θ_2} with Q_{ϕ_2} [20]. The learning target proposed considers the minimum estimate between the two separate critics:

$$y = r + \gamma min_{i=1,2} Q_{\phi'_i}(x', \mu_{\theta'}(x')) \tag{3.57}$$

This novel update rule may generate an underestimation bias, but the authors consider this to be preferable in comparison to an overestimation bias. This is because the propagation of the underestimation bias does not occur through the policy update, as opposed to the overestimation bias. To reduce computational costs, a single actor can be used and optimized concerning Q_{ϕ_1}. TD3 has outperformed other state-of-the-art methods in every OpenAI gym environment tested [20].

4

Multi-Agent Deep Reinforcement Learning

As stated in the introduction of this report, the goal of this research is to assess the feasibility of an autonomous controller to ensure separation in high-density airspace mergers. Model-free RL methods are one of the possibilities to tackle this problem. However, multiple problems arise when we extend single-agent RL algorithms to multi-agent settings, namely non-stationarity or the moving target problem, partial observability, and the credit assignment problem. In this chapter, we discuss solutions found in the literature for these problems.

4.1. Single-Agent vs. Multi-Agent Systems

In a single-agent system, there is only one agent whose goals, actions, and knowledge are modeled. It is however possible to have more agents if we consider a centralized system in which the decision power is done by the only agent interacting with the environment and controlling the other agents that may be viewed as part of the environment, and not as having its own goals, actions, and knowledge [59]. A multi-agent system, on the other hand, is a group of autonomous entities perceiving and actuating on a shared environment [12]. The main difference between single and multi-agent systems is that in a multi-agent scenario, the environment dynamics may be altered due to actions taken by other agents. Figure 4.1 depicts the general case where multiple agents are part of the environment but modeled as different entities, with possibly different goals, actions, and domain knowledge. A specific environment may be modeled using a different number of agents, levels of heterogeneity (by having different goals, domain models, or actions), and with or without the capacity to interact directly via communication [59].

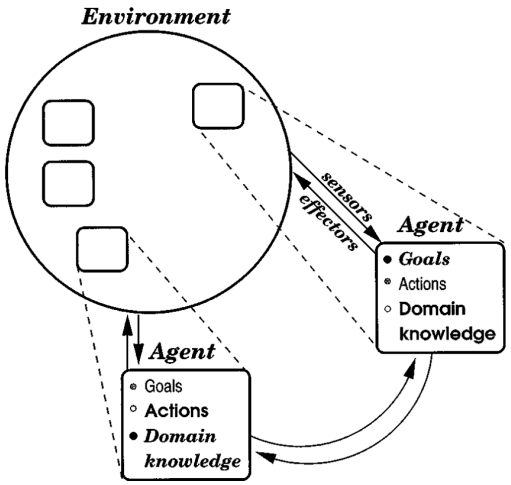


Figure 4.1: A general multi-agent system [59].

4.2. Multi-Agent Environment

As we have seen before, the typical single-agent RL scenario is defined as an MDP. A stochastic game is defined by generalizing the MDP to a multi-agent setting. It is defined as the tuple $< X, U_1, ..., U_n, f, \rho_1, ..., \rho_n >$, where n is the number of agents, X is the finite state space, U_i, is the finite set of actions available to the agents. The interaction between the agents and the environment generates a joint action set $U = U_1 \times ... \times U_n$. The state transition probability is given by $f : X \times U \times X \rightarrow [0,1]$. Lastly, the reward function is given as $\rho_i : X \times U \times X \rightarrow \mathbb{R}$. In a stochastic game, the state transition is conditioned by the joint action of all agents, and the sum of each agent's individual policy forms a joint policy h [13]. The return can then be defined as:

$$R_i^h(x) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{i,k+1} | x_0 = x, h] \tag{4.1}$$

4.3. Challenges and Solutions in Multi-Agent Deep RL

Extending Deep Reinforcement Learning to a multi-agent environment comes with a set of challenges that MADRL algorithms have to address in order to be efficient at learning and executing complex tasks. In this section we will address some of the most relevant problems that arise in multi-agent RL settings, starting with the non-stationarity caused by having multiple agents learning at the same time, creating a moving-target learning problem for each of the agents, and the partial observability problem. Then, we will move on to study which are the possible training schemes in MADRL. The possibilities for coordination and communication between agents will also be discussed. Later, the problem of attributing credit for joint action to individual agents will be elaborated on. Finally, we will touch on scalability issues and how to overcome them using different techniques [13] [43] [21].

4.3.1. Non-stationarity

The non-stationarity problem arises due to simultaneous learning and the fact that the environment dynamics appear to be non-stationary from an individual agent point of view [21]. For this reason, every agent faces a so-called moving target problem, as their optimal policy changes, when the policies of other agents are altered [13]. In a multi-agent domain, rather than only be concerned with the outcome of its own actions, an agent must cope with the behavior of other agents that constantly reshape the environment, creating the non-stationarity problem. The convergence theory that we have discussed before applied to Q-Learning does not hold anymore in multi-agent environments. The DQN algorithm and its variations that we have studied before were not designed to deal with non-stationarity. Luckily, more recently, some authors proposed variations that do [43].

Multi-agent concurrent DQN

Multi-agent concurrent DQN was proposed by Diallo et al. in 2017. Their goal was to let a couple of agents learn to cooperate in the game of pong. To achieve this, agents must be able to co-adapt to each other's behavior. Figure 4.2 depicts the architecture used, considering global goals but independent decision making capabilities [18].

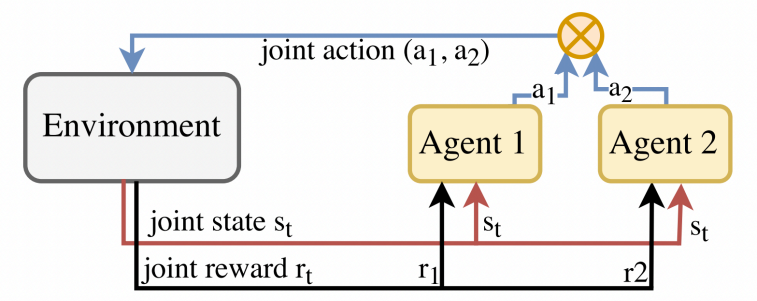


Figure 4.2: Multi-agent concurrent DQN architecture [18].

To let the agents learn to cooperate and correctly divide the area of responsibility, reward shaping is important. This model rewarded both agents if they won a game against a pre-trained AI opponent. In case one agent loses the ball, both agents get punished with a negative reward. Moreover, if they collide with each other, both get penalized as well. If both of them lose the ball and collide, they are penalized to double [18].

The deep Q-network structure used by Diallo et al. is similar to the one we have introduced before. The network takes 84x84 screenshots as input. To cope with the partial observability problem, they stack the last 4 frames, so the agents have information about the speed and direction of the ball. They use a traditional sequence of convolutional layers, followed by fully-connected layers to predict the Q-values for each one of the 3 possible actions: move up, down, or do not move [18].

The experience replay used is also similar to the one introduced in the DQN paper. However, rather than sampling at random to obtain the mini-batches used to train the online network, they use the concept of prioritized experience replay we introduced in the previous chapter. The results of this work have demonstrated that multi-agent concurrent DQN converges even if the environment is non-stationary [18].

Lenient DQN

Lenient-DQN was introduced by Palmer et al. in 2017. To cope with the non-stationarity problem, they use "...leniency with decaying temperature values to adjust policy updates sampled from the experience replay memory" [43]. Lenient learning was first introduced by Potter and De Jong to prevent relative overgeneralization, which happens in multi-agent settings when an agent converges to a sub-optimal policy because of the noise introduced by the exploratory behavior of the other agent. Lenient agents ignore sub-optimal actions taken by teammates that yield low rewards. The level of leniency is reduced every time the agent visits a state-action pair. This factor is controlled by a temperature function [45].

$$l(x_k, u_k) = 1 - e^{-K * T_t(x_k, u_k)} \tag{4.2}$$

Where K is a constant that controls the influence the temperature function has on the leniency function. The temperature is decayed using a β discount factor between 0 and 1. Considering the TD-error $\delta = Y_k - Q(x_k, u_k; \theta_k)$, leniency is applied according to the following rule [45].

$$Q(x_k, u_k) = \begin{cases} Q(x_k, u_k) + \alpha\delta & \text{if } \delta > 0 \text{ or } x > l(x_k, u_k) \\ Q(x_k, u_k) & \text{otherwise} \end{cases} \tag{4.3}$$

Where x is a random variable sampled from the uniform distribution $x \sim U(0,1)$. Lenient DQN adds to the classical tuple of elements stored in the replay memory the current leniency value for that state-action pair. This function is computed similarly to Equation 4.2, but the temperature function is given by a dictionary that maps values of $(\phi(x), u)$ to temperature values. The state hash-key values $\phi(x)$ are calculated using an auto-encoder. An auto-encoder is a neural network that consists of a sequence of convolutional, dense, and transposed convolutional layers, meaning that it is able to learn a compressed representation of the input image. This network is trained using the experience collected by the agent in the replay memory [45].

$$l(x, u) = 1 - e^{-k \times T(\phi(x), u)} \tag{4.4}$$

Where u is the selected action. If the pair $(\phi(x), u)$ is not present in the temperature dictionary, a new entry is created, setting the temperature to its maximum value. On the other hand, if the entry is already present, the temperature will be used and consecutively decayed. To complete the understanding of lenient DQN, we have to explore its exploration strategy, called $\overline{T}$-Greedy Exploration. This action selection method replaces the ϵ in the ϵ-Greedy algorithm by the average temperature value for a state, comprised between 0 and 1. An exponent ξ is used to control the rate agent transitions from explorer to exploiter. Thus, an agent takes the greedy action with probability $1 - \overline{T}(x_k)^{\xi}$, and a random action with probability $\overline{T}(x_k)^{\xi}$ [45].

Weighted DDQN

More recently, in 2018, Zheng et al. proposed weighted DDQN to cope with non-stationarity and the stochastic rewards caused by the environment's characteristics and the coexistence of many learning agents [72]. WDDQN applies the idea of leniency to the reward estimator for each state-action pair. Moreover, arguing

that prioritized experience replay leads to poor performance in MAS because the transitions get outdated as agents update their policies simultaneously, the authors propose a novel scheduled replay strategy to adjust the priority of transition samples dynamically [72].

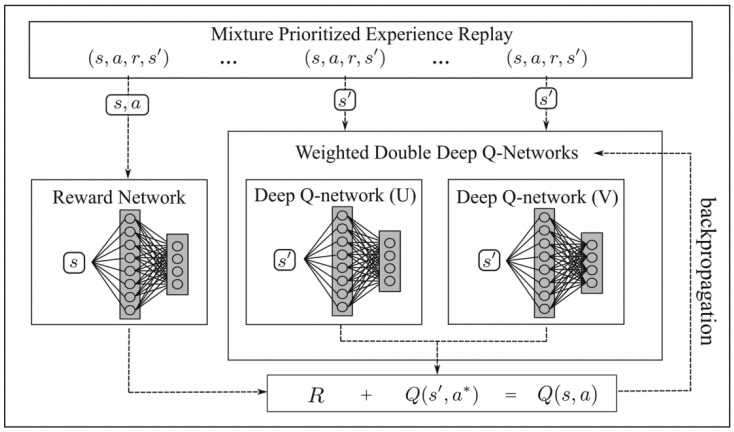


Figure 4.3: Network architecture of WDDQN [72].

WDDQN is adapted from an earlier algorithm presented by Zheng, the weighted double Q, by leveraging neural networks as function approximators. The goal of this WDQ algorithm was balancing between the overestimation in Q-learning we have studied before, and the underestimation in the double Q-learning solution by weighting between a single and double estimator using a parameter β [70]. Thus, in WDDQN, the action is selected by averaging the Q-values given by two estimators, Q^U and Q^V. When u' is chosen by Q^U, this network will be positively biased, and Q^V negatively biased, and vice versa [72].

$$a = max_{u'} \frac{Q^U(x,u') + Q^U(x,u')}{2} \tag{4.5}$$

Furthermore, the target value used in back-propagation is replaced by a weighted combination, balancing between overestimation and underestimation. As shown in Figure 4.3, WDDQN does not use the rewards stored in the replay memory. Instead, they use a lenient reward network (LRN) as an approximator of the reward function to reduce bias in the immediate reward given by stochastic environments and co-adaptive agents. Mis-coordination between agents may lower their rewards independently of the optimality of a given action. To overcome this problem, WDDQN borrows the concept of leniency to update the LRN [72].

$$R_{k+1}(x_k,u_k) = \begin{cases} R_k(x_k,u_k) + \alpha\delta & \text{if } \delta > 0 \text{ or } x < l(x_k,u_k) \\ R_k(x_k,u_k) & \text{otherwise} \end{cases} \tag{4.6}$$

Where $R_k(x_k,u_k)$ is the approximated reward and δ the TD-error between the $R_k(x_k,u_k)$ and the target reward, obtained by averaging all immediate rewards stored in memory for that state-action pair. The leniency function is decayed each time an agent visits a state-action pair as we have seen before [72].

Finally, the authors of WDDQN identified issues with prioritized experience replay (PER). First, PER defines the priority of a transition based on its TD error. The problem is that if a transition has a very biased reward due to the stochasticity of the MAS, the PER is likely to pick this transition as a high-priority one to update the network, when in fact, the reward is incorrect given the noise. To address this, the immediate reward r is replaced by the estimation using the LRN. Secondly, PER attributes the highest priority to all the samples in the new trajectory. However, in MAS, the number of trajectories in which agents succeed in cooperation is low, and when it happens, states closer to the terminal state are more valuable for learning. Moreover, states that are distant from the terminal state can worsen the estimation. Therefore, samples closer to the terminal state should often be used to train the network. For this reason, the authors propose a scheduled replay strategy (SRS) that assigns different priorities based on the sample's position within the trajectory. This way, the SRS assigns higher priority to samples that are closer to the terminal state. This has proven to speed up convergence and improve training performance [72].

4.3.2. Partial Observability

In many real-world applications, complete information about the state of the environment is not available to RL agents. Thus, it is important to build algorithms that cope with the partial observability problem. We have seen this problem before in the context of DQN applied to pong. If an agent has only access to a single frame, it cannot extrapolate what are the speed and direction of the ball, and thus, it will have difficulties learning the best action to take. The DQN authors took a very simplistic approach to solve the problem: stacking together a sequence of the last few frames. However, as one can imagine, this approach is limited in case more information about the past could be used. This problem is particularly relevant in the case of MAS, where environments tend to be more complex, and agents do not know the complete information about the environment when acting on it [43].

Deep Distributed Recurrent Q-Network

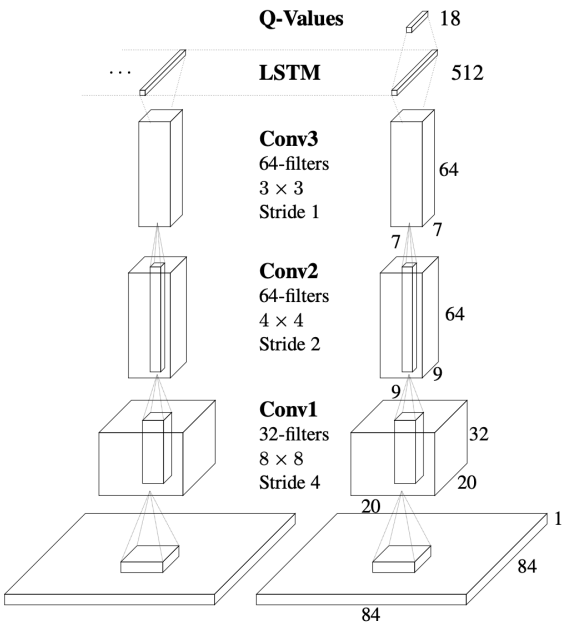


Figure 4.4: Network architecture of DRQN [24].

Deep Recurrent Q-Network (DRQN) was introduced by Hausknecht et al. in 2015. This work aimed at adding recurrency to DRL in an effort to deal with the partial observability problem. This was achieved using a long short-term memory (LSTM) network to approximate the Q function [24].
Instead of the stack of the 4 last frames, the DRQN only takes the current frame (84x84) as input for the convolutional layers, followed by the LSTM. Besides the output of these convolutions, the LSTM also takes the hidden state passed over by the last unit as input. This way, the information about the past is always present in a compact representation and considered by the agent [24]. To accomplish this, the LSTM uses a set of gates to learn which information is important or not, making sure only relevant information about the past is passed forward to the next unit.

To handle multi-agent partially observable MDP (POMDP), Foerster et al. extended the DRQN algorithm to the deep distributed recurrent Q-network (DDRQN) method in 2016. The authors propose three key modifications to the combination of DRQN with IQL, which would be the most straightforward approach to MAS, but comes at the cost of poor performance when compared to this novel DDRQN algorithm [19]. First, they add a last-action input to the recurrent neural network (RNN). As agents use stochastic policies during exploration, they benefit from knowing their action-observation history, rather than the observation history alone. Thus, by feeding the last action to the network along with the state of the environment, the RNN can approximate action-observation histories, improving the agent's decision-making ability. Secondly, they use inter-agent weight sharing, meaning that all agents share the same neural network with the same parameters. Different decisions come only from different observations of the environment and thus, different

hidden states. Lastly, they do not rely on the experience replay used in DQN. As we have seen before, the experience replay can raise challenges in MAS due to non-stationarity, and thus, the authors decided to avoid possibly misleading experiences during training [19].

Deep Recurrent Policy Inference Q-Network

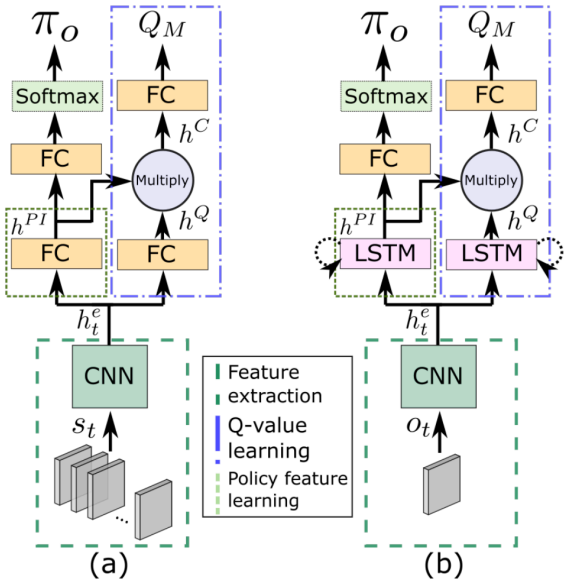


Figure 4.5: Network architectures of DPIQN (a) and DRPIQN (b) [28].

Deep recurrent policy inference Q-network (DRPIQN) was introduced by Hong et al. in 2017. DRPIQN is built on top of DPIQN, a method introduced in the same paper, to solve the partial observability problem. The main goal of DPIQN was to improve the state feature representation in MAS, particularly when modeling agents with different behaviors. This is done by letting a controllable agent learn the policy features of a target agent by observation over time. These features are encoded in a spatial-temporal representation vector h^{PI} of the target agent's policy π_o. DPIQN contains three main components: first, a features extraction module, which consists of a convolutional neural network. These features h_t^e are then used for the other two modules, a Q-function learning module Q_M, and a policy feature learning module π_o. Both the Q-value module and the policy feature module are built of fully-connected neural networks and take the features from the feature extractor as inputs. The Q-value module is trained to approximate the optimal Q-function, while the policy feature module is trying to predict the target agent's next action a_o [28].

DRPIQN is a variant of DPIQN inspired in DRQN to cope with the partial observability problem. We have seen before how this is important for the approximation of the Q-function. However, it is also important for the approximation of the target agent's policy. Considering a competitive environment, the target agent's policy can change from a defensive mode to an offensive mode in an episode, making it more difficult for the agent to adapt the Q-function and the feature policy approximators to the new behavior of the target agent. The problem is more severe in multi-agent cases as the policies of multiple agents are changing at the same time. In these cases, inferring the policy of a target agent is a POMDP problem, as the behavior of the target agent cannot be understood by using only a few observations. By adding recurrent units to both the Q-value and policy feature approximators, DRPIQN incorporates temporal information in the hidden states of the LSTM units, allowing the agents to capture long-time dependencies and better policy feature representations [28].

4.3.3. Training Schemes

As we have said before, the most straightforward extension of DRL to MAS would be to train each agent separately while considering that all other agents are part of the environment. This approach is called decentralized training. Agents share no information between them during the process. The main drawback of using decentralized training is that the environment appears non-stationary from an agent's perspective, as

they do not have access to others' knowledge, nor to the joint action. Several training schemes have been proposed in the literature that is better suited for MADRL. In this section, we will analyze centralized training centralized execution (CTCE), and centralized training decentralized execution (CTDE).

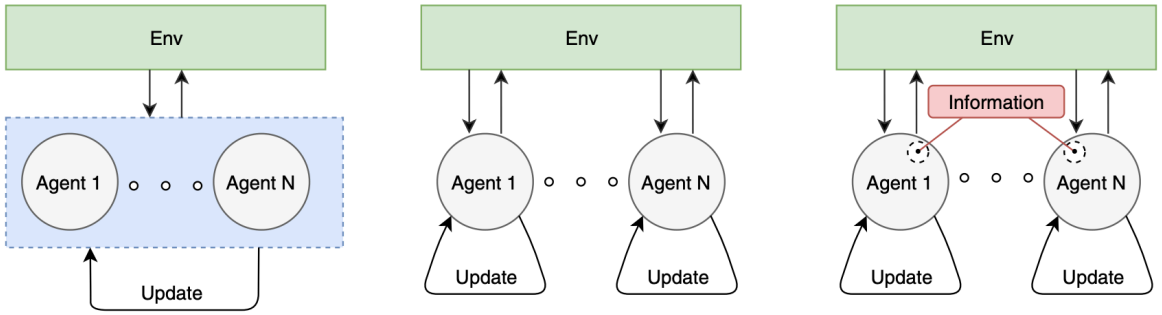


Figure 4.6: Training Schemes in MADRL: Centralized Training Centralized Execution (left); Decentralized Training Decentralized Execution (middle); Centralized Training Decentralized Execution (right) [21].

The centralized training paradigm relies on updating agent policies based on shared information. This can be then subdivided into centralized and decentralized execution schemes. CTCE uses a centralizer executor that takes action on behalf of all agents. The main advantage of this method is that classic single-agent training methods such as actor-critics or policy gradient methods in multi-agent settings. However, it comes at the cost of an exponentially growing state-action space if the number of agents is increased [21]. To cope with the curse of dimensionality problem in CTCE settings, Gupta et al. (2017) represented centralized control as a "...set of independent sub-policies that map the joint observation to an action for a single agent" [22]. For instance, considering a policy gradient approach, this means that the probability of the joint action u is the product of the probabilities of the individual actions u_i for each of the agents [22].

$$P(u) = \prod_i P(u_i) \tag{4.7}$$

This means that the policy of an agent is approximated by a set of output nodes in a neural network, reducing the dimensions from $|U|^n$ to $n|U|$, being n the number of agents and U the action space for each agent. Despite this size reduction of the state-action space, CTCE is still not applicable to complex problems due to scalability issues [22].

The state-of-the-art alternative to CTCE is CTDE. CTDE means that each agent holds an individual policy mapping from observations to a distribution over individual actions. In this paradigm, agents share computational resources and information during training, speeding up the learning process and overcoming non-stationarity, because information about the selected actions is available for every agent [21]. A good example of a CTDE setting was the DDRQN algorithm we have studied, where agents share the same neural network as a value-function approximator, but act differently on the environment depending on their local observations. Parameter sharing can be very beneficial in the case of homogeneous agents since they learn their policies from the experience gathered by all agents simultaneously [22].

4.3.4. Curriculum Learning

Curriculum learning was introduced by Bengio et al. in 2009. The underlying idea is to improve learning by starting with simple tasks and then accumulating knowledge to solve complex tasks. In MAS settings, the tasks of the curriculum become more complex as the number of agents in the environment increases [22]. Curriculum learning proposes reorganizing the training samples according to a set of criteria. By attributing weights to the training samples, the learning process can start with easier samples, moving towards more complex ones as the agent progresses. Considering z is the random variable representing a training sample, such as an input and a target value in supervised learning, and P(z) is the target distribution the agent should learn from. Then, $W_\lambda(z)$ is the weight applied to sample z at time step λ [6]. The training distribution at time step λ is given as follows:

$$Q_\lambda(z) \propto W_\lambda(z)P(z)\forall z \tag{4.8}$$

Q_λ is a curriculum if the entropy increases with time, meaning the training set becomes more diversified, and if the weights of particular examples increase as added to the training set. The most straightforward example of a curriculum is to start training with a set of easy examples, and then move on to the target training set. This allows the network to capture the global picture first, before specializing in more complex examples [6].

4.3.5. Credit Assignment Problem

Considering a cooperative multi-agent setting, the credit assignment problem is the challenge faced to assign joint rewards to individual agents, as the impact of their contribution to the team's success is difficult to estimate. Recent approaches to solving this problem have focused on the decomposition of the reward between agents, considering their individual contributions. In this section we will study two solutions to this problem, as part of the CTDE framework we studied before: first, the value decomposition networks (VDN) "...which factorizes the joint action-value function into a linear combination of individual action-value functions" [21]. Next, we will study QMIX, which was suggested as an improvement to VDN. "QMIX learns a centralized action-value function that is decomposed into agent individual action-value functions through non-linear combinations" [21].

Value Decomposition Networks

VDN was introduced by Sunehag et al. in 2017 as an approach to improve on the centralized approach, that reduces MARL to a single-agent RL problem, and also on independent learners. We have already studied the disadvantages of both these approaches before. This work aimed for an autonomous solution, in which the decomposition of the team action-value function over individual agents is learned [60].

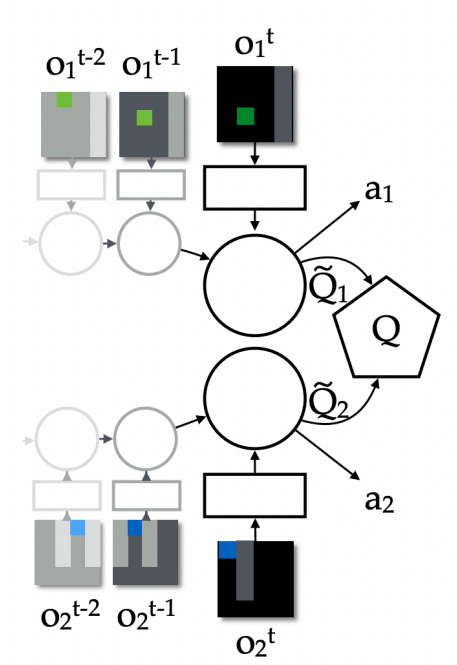


Figure 4.7: Network architecture of VDN [60].

The main assumption for VDN's approach is that the joint action-value function can be decomposed into the sum of the individual contributions of each agent.

$$Q_{tot}(\tau, u) = \sum_{i=1}^{N} Q_i(\tau^i, u^i; \theta^i) \tag{4.9}$$

Where τ is the joint action-observation history, u the joint action, and i the index representing each individual agent's components. The approximated action value for individual agents is learned implicitly through back-propagation using the joint reward as it can be observed in Figure 4.7. This image depicts three steps of the environment. The recurrent network brings valuable information about the past in the hidden state. Each of the two agents' observations passes through a linear layer onto the recurrent layer. Then, a dueling layer produces the values that are summed up to obtain the joint Q-function for training, while actions are selected based on individual outputs [60].

QMIX

QMIX was introduced by Rashid et al. in 2018. QMIX aimed at training decentralized policies in a centralized end-to-end way. They proposed the estimation of the joint action-value function through complex non-linear combinations of local agent values [48].

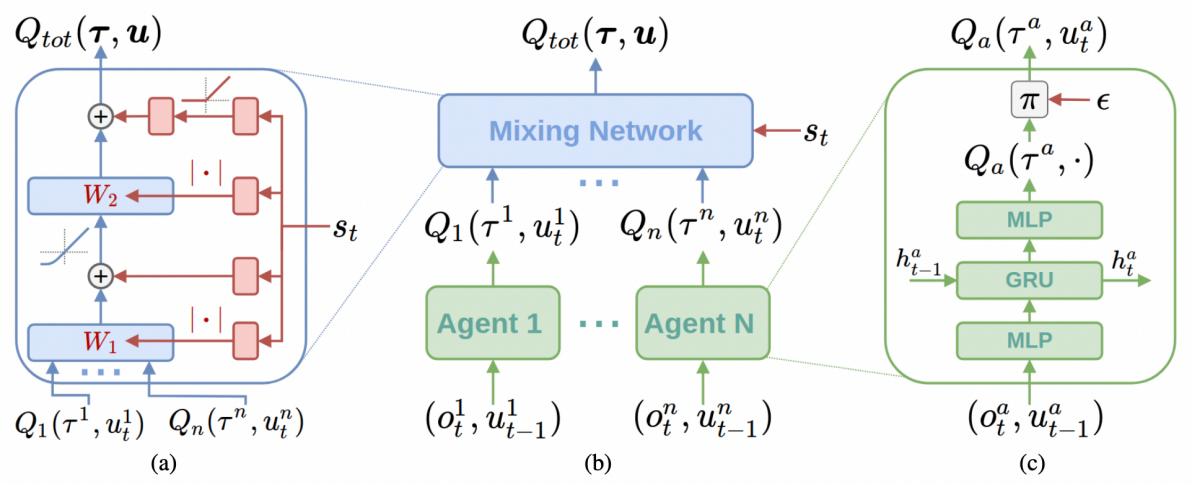


Figure 4.8: Network architecture of QMIX. Mixing Network (a); Overall QMIX architecture (b); Agent Network (c) [48].

QMIX represents Q_{tot} uses an architecture consisting of agent networks, mixing networks and hyper networks. Each agent a is assigned a network inspired on DRQN that approximates its individual action-value function $Q_a(t^a, u^a)$. The mixing network consists of a feed-forward neural network, taking the agent network outputs as inputs to compute the value of Q_{tot}. To ensure monotonicity, i.e., that the partial derivative of Q_{tot} with respect to Q_a is positive, the weights of the mixing network are always positive. These weights are calculated by the hyper networks, which take the state as input and output the weights for a layer of the mixing network. Finally, QMIX is trained end to end with respect to the following loss function [48].

$$L(\theta) = \sum_{i=1}^{b} [(y_i^{tot} - Q_{tot}(\tau, u, x; \theta))^2] \tag{4.10}$$

Where b is the batch size of transition samples taken from the replay buffer, $y^{tot} = r + \gamma max_{u'} Q_{tot}(\tau', u', s'; \theta-)$, and $\theta-$ are the parameters of the target network as we have seen in DQN [48].

4.3.6. Continuous Environments

Discrete environments are not sufficient to model the full complexity of some real-world applications. For instance, as we have seen before, the solutions for discrete environments are not scalable for a large number of possible actions. Thus, it is important to study what are the solutions for multi-agent continuous environments. In this section, we will extend both the DDPG and the PPO algorithm for multi-agent settings.

MADDPG

MADDPG was introduced by Lowe et al. in 2017 as an adaptation of actor-critic methods that successfully learn multi-agent cooperative policies. This was achieved with a CTDE framework, by allowing policies to use

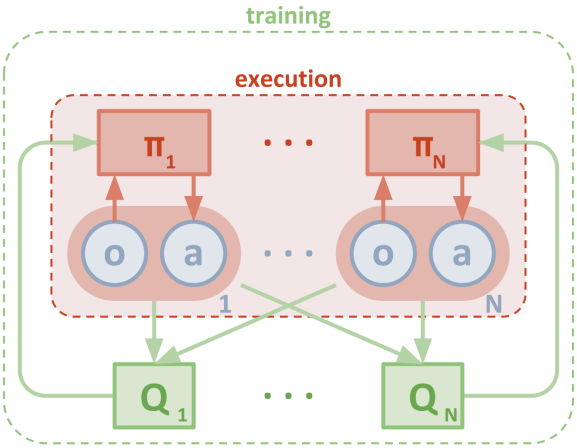


Figure 4.9: Network architecture of MADDPG [38].

extra information at training time. In this case, the authors extended an actor-critic policy gradient method where the critic is augmented using external inputs from other agents [38].
The gradient of the expected return for agent i $J(\theta_i)$ is given as:

$$\nabla_{\theta_i} J(\theta_i) = E_{x\sim p^u, u_i\sim\pi_i}[\nabla_{\theta_i} log\pi_i(u_i|o_i) Q_i^{\pi}(x, u_i, ..., u_n)] \tag{4.11}$$

Where π is the set of all agent policies, θ the parameters of these policies, and N the number of agents. $Q_i^{\pi}(x, u_i, ..., u_n)$ is a centralized action-value function that considers all agents' actions, observations, and other state information available as input and predicts the Q-value for agent i. These functions are learned separately by each agent, allowing for the possibility of conflicting rewards in the case of competitive settings, but also for similar rewards in cooperative cases. In deterministic policy settings, this gradient can be rewritten as [38]:

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{x,u\sim D}[\nabla_{\theta_i}\mu_i(u_i|o_i)\nabla_{u_i} Q_i^{\mu}(x, u_i, ..., u_n)|_{u_i=\mu_i(o_i)}] \tag{4.12}$$

Where μ_i represents a continuous policy μ_{θ_i} with parameters θ_i. D is the experience replay buffer containing state, action, and reward information from all agents- Q_i^{μ} is updated according to the following loss function [38]:

$$L(\theta_i) = \mathbb{E}_{x,u,r,u'}[(Q_i^{\mu}(x, u_1, ..., u_N) - y)^2] \tag{4.13}$$

Where y is the target value computed as $y = r_i + \gamma Q_i^{\mu'}(x', u_1', ..., u_N')|_{u_j'=\mu_j'(o_j)}$ and μ' is the set of target policies with delayed parameters θ'. One of the reasons MADDPG works is that by knowing all other agents' actions, the environment becomes stationary, even when the other agents' policies change. Furthermore, each agent can maintain an approximation of each of the other agents' policies, which is then used to compute the target value in Equation 4.13. This approximation is updated by maximizing the log probability of other agents' actions. Finally, to cope with the non-stationarity problem that arises in multi-agent settings due to the agents' changing policies, the authors propose to train K in different sub-policies. At each time step, a random sub-policy is selected for each agent, making agents more robust in handling changes in the policy of other agents [38].

R-MADDPG

R-MADDPG was introduced by Wang et al. in 2020. The goal of this recurrent algorithm is to handle MA coordination under partial observability settings with limited communication between agents. Beyond the recurrency added to both the actor and the critic, another key difference between this approach and the earlier MADDPG is that R-MADDPG learns two policies in parallel - one for navigation and another one for communication [63].

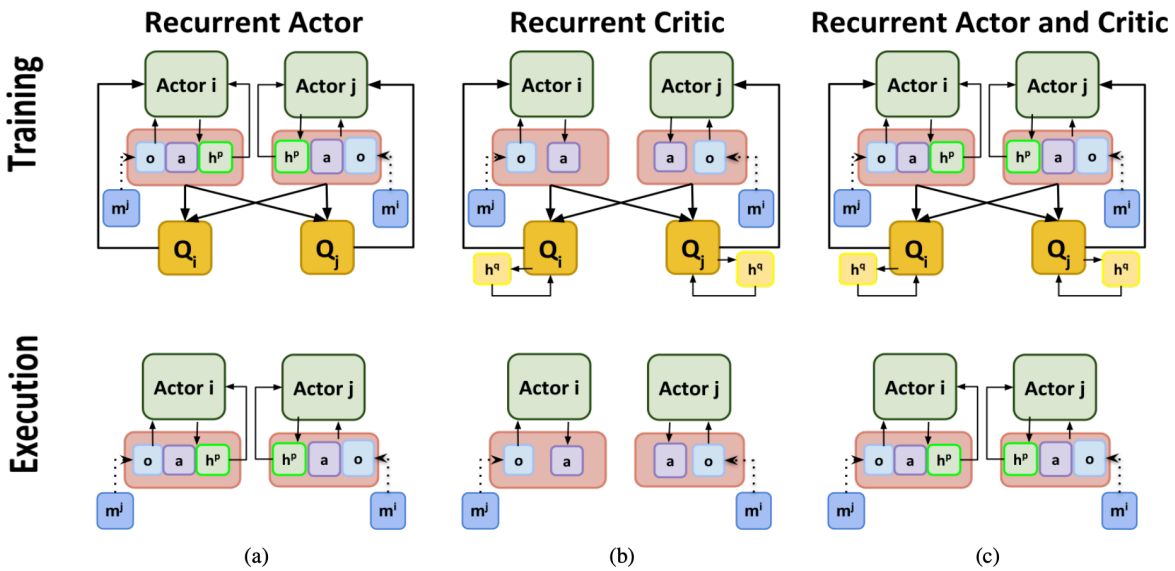


Figure 4.10: Network architecture of R-MADDPG [63].

Recurrency is added on top of MADDPG to handle limited communication constraints, by enabling agents to remember past communications and how they affect the capped communication budget. This work introduces three different models to cope with partial observability and limited communication models. First, a recurrent actor, where an agent's replay buffer D contains the following information vector: $(o_{i,k}, u_{i,k}, o'_{i,k+1}, r_{i,k}, o_{i,k}, c^p_{i,k}, c^p_{i,k+1})$. o represents the partial observations of agent i at time step k, u its action resulting from policy $\pi_{i,k}(o_{i,k}, c_{i,k})$, r the agent's reward, and c^p the cell state of the actor-network. Secondly, this work suggests a recurrent critic, where an experience at time step k contains the following information vector: $(o_{i,k}, u_{i,k}, o'_{i,k+1}, r_{i,k}, o_{i,k}, c^q_{i,k}, c^q_{i,k+1})$. The only difference for the recurrent actor is that here c^q denotes the cell state of the critic network. By combining the information present on both replay buffers described before, the authors propose a recurrent MA actor and critic. Assuming that the agent's current policy is μ and the target policy μ', and the experience sampled uniformly from the experience replay D, the policy gradient is calculated as:

$$\nabla_{\theta_i} J(\mu) = \mathbb{E}_{U(D)}[\nabla_{\theta_i} \mu(u_{i,k}|o_{i,k}, c^p_{i,k}) \cdot \nabla_{a_{i,k}} Q^\mu_i(x, u, c^q_t)|_{a_{i,k}=\mu_i(o_{i,k}, c^p_{i,k})}] \tag{4.14}$$

The action-value function Q^μ_i is updated using the following loss function:

$$L(\theta_i) = E_{U(D)}[r_i + \gamma Q^{\mu'}_i(x', u'_j, c^q_{t+1})|_{a'_j=\mu'_j(o_j, c^p_j)} - Q^\mu_i(x, u, c^q_t)^2] \tag{4.15}$$

MAPPO

MAPPO was introduced by Yu et al. in 2021. There have been two critical reasons for the lack of PPO applications in MARL settings: first, PPO is considered to be less sample efficient than off-policy methods such as DDPG or DQN; Secondly, the usual implementations and hyperparameters used in single-agent PPO would not yield substantial results in MA settings [68].

The approach chosen to extend PPO to MA settings is a decentralized POMDP with shared rewards. Similarly to single-agent PPO, agents learn a policy π_θ and a value function $V_\phi(x)$. The authors use parameter sharing for environments with homogeneous agents for the policy and value function networks. Since V_ϕ is only utilized during training, it takes as input information other than the agent's local observations, but still holds a CTDE structure. State-of-the-art techniques such as GAE are used in MAPPO as studied before in this report for the single-agent situation [68].

5

Deep Reinforcement Learning for Conflict Resolution

As previously mentioned, the objective of this research is to evaluate the performance of an autonomous controller trained by Reinforcement Learning in resolving conflicts in high-density traffic concentrations for en-route aerial vehicles. This chapter builds upon the previous knowledge on single and multi-agent DRL architectures to analyze DRL-based solutions for autonomous en-route aircraft conflict resolution proposed in the literature considering the three resolution techniques studied: speed, heading, and altitude adjustments. Five components will be studied for each solution: the environment, observation space, action space, reward function, and algorithm performance. This way, ideas, and conclusions will be extrapolated for later use in this report's next and last chapter: the research proposal.

5.1. Environment

[7] proposed a hierarchical RL solution for autonomous airspace control using the *NASA Sector 33* app. This learning environment provides the possibility of simulating multiple aircraft flying from origin to destination with numerous possible routes with the goal of maintaining safe separation and minimizing delay. Thus, in this work, there are two decisions to be made by the autonomous controller: choosing an aircraft's route (once per episode) and its speed (at every time step (Δt). One episode is defined as a simulation, i.e., either every aircraft achieved their goal destination (g_{x_i}) with no collisions and on time $|g_{x_i} - x_i| = 0, \forall i$, at least one aircraft got out of time, not arriving at the goal position given the time constraints $|g_{x_i} - x_i| > 0, \forall i$, or the aircraft collided with one another $\sqrt{(y_j - y_i)^2 + (x_j - x_i)^2} < \delta, \forall i \neq j$, where δ is the collision threshold. The optimal solution for the problem with two aircraft is found when the following equation is satisfied: [7]

$$|g_{x_1} - x_1| + |g_{x_2} - x_2| = 0 \tag{5.1}$$

The same authors of [7] proposed a multi-agent distributed solution for the same autonomous separation assurance problem using the BlueSky simulator in [8]. The goal of this work is to resolve conflicts and ensure a safe separation between all aircraft in a series of air route intersections by adjusting the speed. The main difference, when compared to the previous work, is that speed control is decentralized among multiple agents, instead of centralized control for all aircraft. The case study is designed using a realistic ATC simulator (BlueSky) with only three aircraft speeds possible: minimum, current, and maximum cruise speed. The terminal state is achieved when all aircraft had exited the sector $N_{aircraft} = 0$. A conflict happens when the distance between two aircraft is less than 3NM [8].

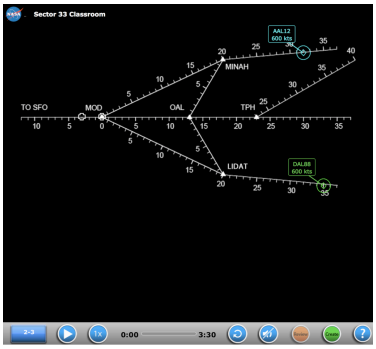


Figure 5.1: *NASA Sector 33* simulator with two aircraft [7].

Figure 5.2: BlueSky sector with three routes R_1, R_2 and R_3, and two intersections I_1 and I_2 [8].

The authors of [8] later suggested an alternative architecture to cope with a varying observation size in a similar Bluesky environment, instead of considering the N-closest agents only. This was proposed in [10] by using an LSTM over agents, instead of overtime to encode all intruder information in a fixed-length vector. This idea was then replaced by an attention mechanism in [9] that has the advantage of accessing all hidden states, and not losing any relevant information. Moreover, in [9], a new reward component was also introduced to minimize the number of speed changes taken [9].

The previously mentioned works consider en-route conflict resolution by varying aircraft speed only. In [71] the authors consider heading changes in addition to speed adjustments. This work uses a PyGame custom environment and the concept of solution space diagram (SSD) as prior ATM-specific knowledge to assist learning [71].

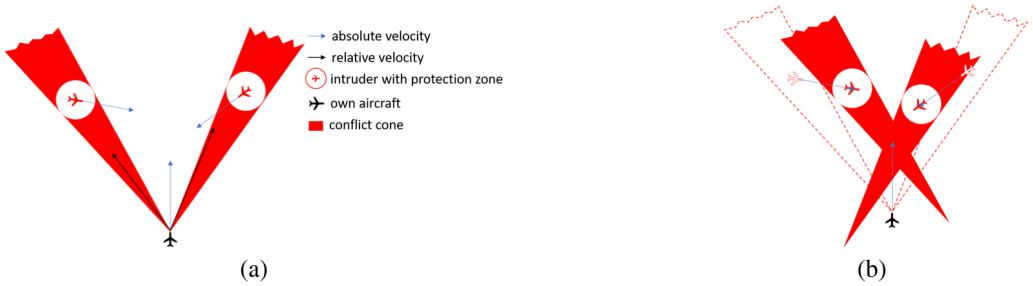


Figure 5.3: SSD assistive methods for conflict resolution [71].

Figure 5.3 depicts two conflict resolution aids provided by SSD. In (a), a conflict exists if the relative velocity vectors are within the conflict cones. This method can cause some difficulties to find the optimal speed and heading if multiple aircraft are present. In (b), by translating the conflict cones according to each of the intruders' speeds, a conflict is defined only if the ownship velocity vector is within the translated area in red. The goals of this work are safety assurance and minimized disruption and a conflict happens when the separation between two aircraft is less than 5NM. This work considers an additional vertical control layer of safety for when a conflict is imminent (i.e. happening in 2 minutes or less). The logic used to resolve conflicts is as follows: when a conflict is detected (i.e. the velocity vector of an aircraft is within the conflict cone area, the controller chooses a conflict resolution method. During the resolution period, the intention velocity (pointing towards the next way-point) is monitored, and as soon as it escapes the danger zone, the aircraft resumes this velocity vector to return to the path. To coordinate multiple aircraft a centralized controller is used according to the following rules: in a conflict between two aircraft, if one of them does not respond to the cooperation request, the other must take action; in a conflict between two aircraft, the one with a larger q-value takes action; in conflicts between more than two aircraft, the larger non-conflicting set is identified (i.e. aircraft with no conflict with each other), and aircraft that are not part of this set take action [71].

5.2. Model

5.2.1. Observation Space

To solve the aircraft routing and conflict resolution problem for *NASA Sector 33* simulator, [7] proposed a hierarchical RL approach involving two agents: a parent and a child agent. The parent agent observes the raw pixel app input and decides which route combination to choose based on its comprehension of the air traffic situation in the sector. The image is processed using a CNN. This action is taken once at the beginning of the episode. The child agent, on the other hand, uses a multi-layer perceptron with the current aircraft coordinates in the two-dimensional euclidean space (x_i, y_i), speed (v_i) and route combination $(c_j$, where j is a route combination) chose by the parent agent as inputs (observations). The child agent takes action every time step to control aircraft speeds. Therefore, the observation space for the parent agent is constant and equal to $m \times m$ pixels, where m is the size of the screen. Each pixel contains color information as a tuple. The child agent observation space is dependent on the number of agents (n), equals $2 \times n + n + 1$ [7].

The multi-agent decentralized approach in [8] requires communication between agents, and this is reflected in the observation space. The observation space includes information about the ownship as well as the N-closest agents that may create a conflict: these include agents on a conflicting route that did not reach the intersection yet or agents on the same route. By adopting these state space rules, the authors ensure the state space contains only relevant information and is constant in size. The observation space can be represented as [8]:

$$s_t^o = (I^{(o)}, d^{(1)}, LOS(o,1), d^{(2)}, LOS(o,2), ..., d^{(n)}, LOS(o,n), I^{(2)}, ..., I^{(n)}) \tag{5.2}$$

Where $I^{(i)}$ is the distance to the goal, aircraft speed, aircraft acceleration, distance to the intersection, route identifier, and half the loss of separation for aircraft i. The loss of separation is used to let agents develop optimal strategies even when different aircraft types are different. $d^{(i)}$ is the distance between aircraft i and the ownship. $LOS(o,i)$ is the loss of separation between the ownship o and aircraft i [8]. In [9], the state space is set according to the same rules of which aircraft may be represented in the ownship observations. The difference is that, because this work considers an attention layer, a variable number of agents can be considered, instead of the fixed N-closest defined in [8].

In [71], the authors consider an SSD-based image observation space containing all the required information for the agent to learn: number of aircraft, speeds, headings, and potential conflicts. This image-like representation as in the red areas in Figure 5.3 (b) is fed into a CNN that outputs a compact representation of the current traffic situation for learning [71].

5.2.2. Action Space

As mentioned before, the work on *NASA Sector 33* considers a parent agent that acts once per episode and a child agent that acts every $\Delta t = 4s$. Considering the environment case with two aircraft in Figure 5.1, the parent agent's discrete action space is given as $A_p = (C_1, ..., C_j), \forall j$, where j is a route combination. In this case, $j = 4$, as each aircraft can take two distinct routes, which makes a total of four unique route combinations. The child agent discrete action space is defined as $A_c = (U_1, ..., U_k), \forall k$, where U represents the set of all speed combinations and k is one unique combination. In the experiments of this work, the authors consider a speed envelope between 300 and 600 knots with 6 possible aircraft speeds in between. Thus, the size of the child's action space for two aircraft is 36 [7].

In [8] the authors propose a smaller discrete action space for the multi-agent approach: an aircraft can decide to change its speed every $\Delta t = 12s$. It can choose to travel at the current cruise speed, minimum allowed cruise speed, or the maximum allowed cruise speed:

$$A_t = [v_{min}, v_{t-1}, v_{max}] \tag{5.3}$$

The authors of [9] work on the Bluesky simulator with attention consider the same time difference between actions and an action space comprised of three actions: a_- (decrease speed), 0 (hold speed), and a_+ (increase speed), constrained to a minimum and maximum speed set by the simulator [9].

$$A_t = [a_-, 0, a_+] \tag{5.4}$$

In [71], four different action spaces are considered. First, a discrete heading controls considering three actions as shown in Figure 5.4.

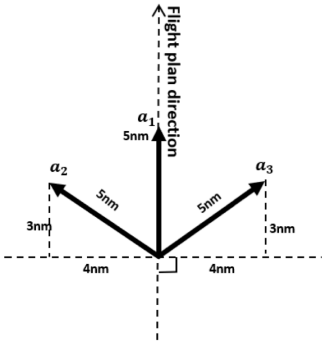


Figure 5.4: Discrete heading action space [71].

Secondly, a continuous heading control for a smoother resolution path. This action selects an angle between the original direction and the selected direction in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$. Thirdly, continuous speed control for an agent to choose what exact speed it wants to add with a range (-50knots, 50knots). Lastly, a combination of the heading angle change and speed control in a two-dimensional action space [71].

5.2.3. Reward Function

[7] defined a reward function for both parent and child agents according to the goals of the work: maintain safe separation, optimal route choice, and arrival delay minimization. The parent agent is rewarded based on the distance from each aircraft to their goal positions:

$$r_p = \frac{1}{\sum_{i=1}^{N} |g_{x_i} - x_i|} \tag{5.5}$$

The child agent is rewarded based on the speed of each aircraft:

$$r_c = 0.001 \sum_{i=1}^{N} v_i - 0.6 \tag{5.6}$$

The 0.001 constant is added to scale rewards between -1 and 1, while -0.6 is added to penalize slower speeds and reward higher speeds maintaining efficiency. There are three additional rewards provided: -10 for collision, -3 for out-of-time and $+10$ for optimal solution [7].

In [8], the same authors proposed a locally applied identical reward function for the distributed multi-agent approach. This means that if two agents are in a conflict they both get negatively rewarded, but the others do not. The reward function is defined as:

$$r_t = \begin{cases} -1, & \text{if } \ d_o^c < 3NM \\ -\alpha + \beta \cdot d_o^c, & \text{if } \ 3NM \leq d_o^c < 10NM \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

Where d_o^c is the distance between the ownship and the closest aircraft. α (0.1) and β (0.005) are constants that penalize agents for approaching the minimum separation distance of 3NM [8]. The reward function in [9] is a two terms sum: the first component being given by Equation 5.7, and the second (to minimize the number of speed adjustments) given as:

$$r_t(a) = \begin{cases} 0, & \text{if } \ a = Hold \\ -\psi, & \text{otherwise} \end{cases} \tag{5.8}$$

In [71] there are three separate reward functions that are combined in the different action space settings discussed before. First, a reward function with respect to safety is given:

$$R_c = \begin{cases} -1, & \text{if conflict} \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

Secondly, a reward function related to the deviation from the original flight plan is given with respect to θ, the angle between the direction selected and the intention direction:

$$R_h = 0.001 cos(\theta) \tag{5.10}$$

Lastly, a speed term is added to penalize deviations from the cruise speed, as it becomes more fuel-consuming to fly:

$$R_s = 0.001 e^{(\frac{v-v_0}{v_u-v_l})^2} \tag{5.11}$$

Where v is the current speed, v_0 the cruise speed, v_u the maximum speed and v_l the minimum speed. Finally, gathering all the previous rewards for the different action space cases:

$$R = \begin{cases} R_c + R_h, & \text{heading change} \\ R_c + R_h, & \text{speed change} \\ R_c + R_h, & \text{simultaneous heading and speed} \end{cases} \tag{5.12}$$

5.2.4. Algorithm Performance

To solve the *NASA Sector 33* game using hierarchical RL, [7] used DDQN applied for both the parent agent (using a CNN feature extractor) and the child agent (using an MLP feature extractor). To analyze the performance of the algorithm, the authors defined a game score as given by the rewards for collision, out-of-time and optimal solution $(-10, -3, +10)$. The evolution of the game score with the training process is depicted in Figure 5.5, where it is possible to see that the score increases as the AI agent learn [7].

In [8], the authors proposed an A2C advantage actor-critic algorithm with the loss function from PPO with parameter sharing between the actor and the critic. A CTCE framework with parameter sharing is used, where a centralized network learns and distributes the knowledge to be used in a decentralized way by all aircraft. This distribution happens at the beginning of every episode and the network is updated at the end. To evaluate the performance of this multi-agent approach, the authors define a goal as an aircraft exiting the sector with no conflicts. As there are 30 aircraft entering the airspace at a uniform distribution over 4, 5, and 6 minutes, the optimal solution must achieve a goal value of 30 [8].

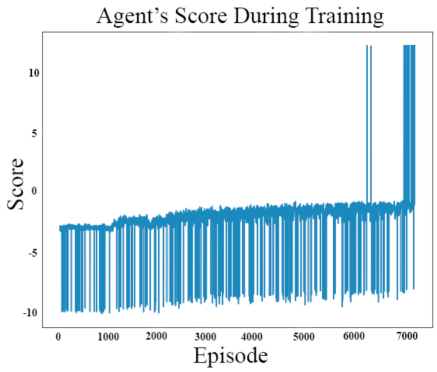


Figure 5.5: RL agent score during DDQN training process [7].

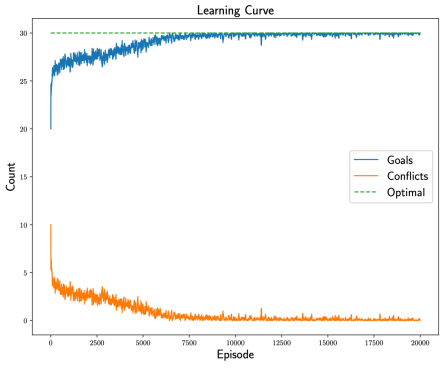


Figure 5.6: Learning curve in DD-MARL [8].

The authors in [9] tested the same case as in [9] but with a novel approach on top of the CTCE parameter sharing PPO algorithm: considering attention to cope with a varying number of agents. The attention layer that represents the understanding of the air traffic situation from an agent's point of view (a_s) can be defined by the following equations:

$$score(s, \overline{h}_i) = s^T W_1 \overline{h}_i \tag{5.13}$$

$$\eta_{s,\overline{h}_i} = \frac{exp(score(s, \overline{h}_i))}{\sum_{j=1}^{n} exp(score(s, \overline{h}_j))} \tag{5.14}$$

$$c_s = \sum_{i=1}^{n} \eta_{s,\overline{h}_i} \overline{h}_i \tag{5.15}$$

$$a_s = f(c_s) = tanh(W_2 c_s) \tag{5.16}$$

Where s is the pre-processed ownship state through fully-connected layers and $\overline{h}_i$ the pre-processed state of intruder i and n the total number of intruders. η_s is the attention weights of the ownship and each of the other intruders, and c_s is the context vector representing the surrounding air traffic.

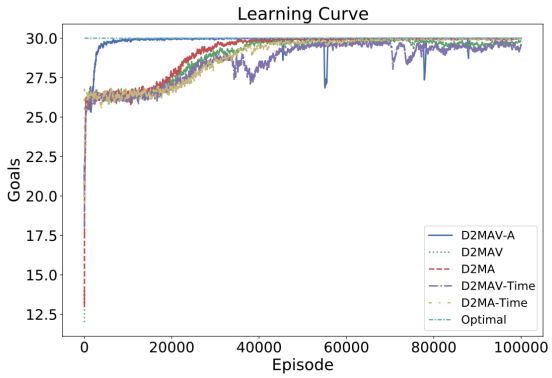


Figure 5.7: Learning curves in [9].

Figure 5.7 depicts the learning curve relative to the previously explained score indicator (30 is optimal) for the algorithms in [8] (D2MA), [10] (D2MAV with LSTM) and [9] (D2MAV-A with attention). In the original D2MA and D2MAV works, the N-closest or all applicable agents were sorted based on the distance to the ownship. Since the D2MAV-A algorithm does not require a pre-defined sorting strategy, a new sorting strategy was taken into consideration for fair comparison - the relative time to the intersection - with D2MA-Time and D2MAV-Time [9]. The figure makes it possible to conclude that using attention improves convergence and speeds up the learning process.

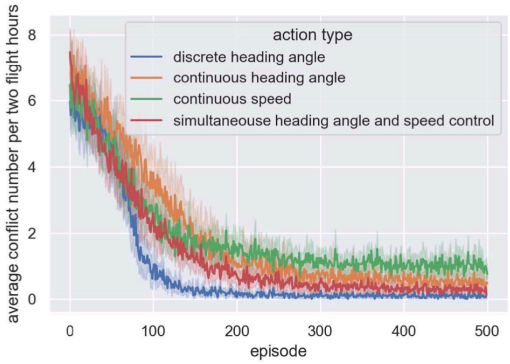


Figure 5.8: Conflict resolution performance of different action types during training [71].

In [71], the authors compare the evolution of the average number of conflicts during two flight hours during training for the four different action types using the PPO algorithm for training. In Figure 5.8 it is observable that the discrete heading angle action outperforms all the other options [71].

6

Research Proposal

The solutions for the conflict resolution problem analyzed before can be subdivided into two large groups: the ones that consider a centralized decision-making agent, such as in [7] and [71], and the ones that consider decentralized decision-making agents (multi-agent learning problem), explicitly sharing the same knowledge and communicating relevant information at every time step as in [8], [9] and [10]. Moreover, these solutions can also be categorized by the actions used to solve the conflict. While [7], [8], [9] and [10] use speed adjustments only, [71] considers heading adjustments as well. As society moves towards autonomous airspace control for high-traffic urban airspaces, solutions will have to first be supervised by human operators. For this reason, this study focuses on a centralized solution, whose actions are easier to understand. The work in [7] tests a hierarchical solution for a low number of agents only. In [71], the method relied on a centralized algorithm with pre-defined rules. This work aims at building a centralized autonomous DRL-inspired solution that controls agents directly in a variety of airspace intersections. Being agnostic to route configurations is important so that the agent can be generalized to multiple locations. As for the conflict resolution method, this proposal focuses on changing speed, as it is the less disruptive solution for pre-defined air routes. This section is organized as follows: first, the targeted research questions are enumerated; second, the model proposed to answer these questions is detailed, including the observation and action spaces, reward functions, and the reasoning behind the choice of learning algorithm; finally, some proposed scenarios for sensitivity analysis are developed.

6.1. Research Questions

1. To what extent can a centralized autonomous DRL controller ensure safe separation and efficiency in high-density urban aerial intersections with variable configurations?

    (a) How to model a realistic urban environment? What speed envelopes and collision boundaries should vehicles have?

    (b) What state information does the centralized controller require and how to obtain it?

    (c) What are the safety and efficiency goals? What reward function should be adopted to reflect them?

    (d) How to structure a feasible action space to cope with a large number of agents in the intersection?

    (e) What state-of-the-art DRL algorithm should be chosen to train the autonomous controller given the environment settings defined before?

    (f) How to quantify the agent's performance? How do the quantifiable metrics evolve with training?

    (g) What is the effect of different route configurations on the performance of the autonomous agent?

    (h) What is the effect of varying the aircraft sector arrival rates on the performance of the autonomous agent?

    (i) How does the distribution of actions of the centralized agent evolve over a simulation?

    (j) How does the typical velocity profile of a vehicle evolve over a simulation? Is the controller minimizing the number of commands an aircraft needs to perform?

6.2. Environment

The environment proposed is designed using OpenAI Gym library for RL research [11]. The option for a custom environment was chosen because it speeds up the development process and allows for a design targeted at the goals of this particular research. This work chooses to focus on a particular type of airspace intersection: mergers - meaning that two or more routes converge into a single route, causing potential conflicts between vehicles. The environment simulates random configurations for generalization where the angle between any entry route and the escape route after convergence is at least 90ž. The arrival distribution of agents at each route is set at random by a Poisson process. The goals of the environment are to ensure a safe separation between aircraft and maximize efficiency (i.e. to minimize the time it takes for them to exit the sector). At the beginning of the episode, a pre-defined constant number of aircraft are positioned to enter the sector according to the previously mentioned distribution. The episode ends if every aircraft has exited the sector without collisions, or if two aircraft collide with each other according to the euclidean distance $\sqrt{(y_t^{(j)} - y_t^{(i)})^2 + (x_t^{(j)} - x_t^{(i)})^2} < \delta, \forall i \neq j$, where δ is the collision threshold.

6.3. Model

6.3.1. Observation Space

The observation space contains all information for the centralized agent to learn. For every aircraft, this means knowing the route identifier (angle of the route), distance to the intersection, and current velocity. Thus, if n is the total number of aircraft in a simulation, the total size of the observation space is $n \times 3$. An agent's information is only present in the observation space if the agent is active, meaning that it has already entered and not yet left the sector. If an agent is not active, the corresponding observation entries are padded with zeros. Agents are ordered into the observation space according to their time of arrival.

$$O_t = (R_t^{(1)}, d_t^{(1)}, v_t^{(1)}, ..., R_t^{(n)}, d_t^{(n)}, v_t^{(n)}) \tag{6.1}$$

Where R_t is the route identifier between 0 and 2π, d_t is the distance to the intersection between 0 and the route length, and v_t the current speed between within the allowed speed envelope. The observation space is normalized between -1 and 1 to speed up learning.

6.3.2. Action Space

The action space for this research was thoroughly thought to cope with the expected high-density traffic in urban areas. For this reason, the action choice is limited to speed control and at each time step, the centralized controller chooses one vehicle to apply a speed change (no changes are also possible). The final action space is discrete and contains the possibility of not changing anything or speeding up or slowing down a given sector aircraft.

$$A_t = [None, A_-^{(1)}, A_+^{(1)}, ..., A_-^{(n)}, A_+^{(n)}] \tag{6.2}$$

An aircraft speed is constrained by its speed envelope and the + and - signs represent speeding up and slowing down by a fixed Δv. The action space is of size $2 \times n + 1$. When we compare the suggested action space with the combination of all possible speeds suggested in [7], where the action space size is given as S^n, where S is the size of the set of speed values an aircraft can take, we observe a much smaller action space size for large values of n, making training faster and more feasible for high congested areas. Possible disadvantages of the action space chosen are that it takes a few time steps to go from the lower to the higher speed and that it is not possible to change the speeds of two conflicting aircraft at the same time. However, if the frequency of actions is high enough, this should not compromise performance significantly, and it makes decisions more understandable for a human supervisor.

6.3.3. Reward Function

The reward function should provide the agent with an indication of what the goals are. In this case, they are minimizing the number of conflicts and the time it takes for an aircraft to exit the sector. The reward function is comprised of four individual components. First, an episode reward is given in case of success or failure:

$$R_e(t) = \begin{cases} L, & \text{if all aircraft exit without collisions} \\ -L, & \text{if } \sqrt{(y_t^{(j)} - y_t^{(i)})^2 + (x_t^{(j)} - x_t^{(i)})^2} < \delta, \forall i \neq j \end{cases} \tag{6.3}$$

A large positive reward is given in case of success and a large negative reward in case of failure as in [7]. Secondly, a dense reward is given to prevent conflicts from happening. This is based on the idea of an outer boundary where it is still possible to resolve the conflict as discussed in the first chapter of this report. As soon as two aircraft lose this separation, the agent receives a negative reward, that gets larger in absolute value as they get closer to the collision. This reward is averaged over the number of combinations of active pairs of aircraft (P).

$$R_s(t) = \frac{1}{|P|} \sum_{p^{(ij)} \in P} W_s \cdot (O_b - d_{p^{(ij)}}) \tag{6.4}$$

Where W_s is a small negative weight added to scale the reward, O_b is the outer boundary value, and $d_{p^{(ij)}}$ is the euclidean distance between aircraft i and j. Thirdly, a velocity reward is given to encourage larger speeds. This function is the average overall active aircraft (N).

$$R_v(t) = \frac{1}{|N|} \sum_{n \in N} W_v \cdot v_t^{(n)} \tag{6.5}$$

Where W_v is a small positive weight added to scale the reward. This makes the agent prioritize higher cruise speeds, increasing time efficiency that should be crucial in urban operations. Lastly, an action reward is added as in [9] to minimize the number of actions, improving safety and reducing the number of speed variations.

$$R_a(t) = \begin{cases} 0, & \text{if } A_t = None \\ -\psi, & \text{otherwise} \end{cases} \tag{6.6}$$

Where ψ is a small constant to penalize taking actions that are not meant to keep aircraft speeds as they are. Finally, the total reward can be computed as:

$$R(t) = \begin{cases} R_e(t), & \text{if episode is finished} \\ R_s(t) + R_v(t) + R_a(t), & \text{otherwise} \end{cases} \tag{6.7}$$

6.3.4. Algorithm

The DRL algorithm chosen to train a centralized agent according to the environment, observation and action spaces, and reward function described before was PPO. The implementation for training is based on Stable Baselines 3 (SB3) [47] and a CTCE framework. There were two main factors for this choice: first, PPO outperforms other policy gradient algorithms in multiple environments as shown in [54]. Although value-based methods could have an advantage in case the environment was expensive to sample from, this does not apply to this problem; secondly, most related works studied use PPO partially or totally, proving the efficiency of this recent DRL algorithm for the conflict resolution space.

6.4. Experiments

After training the centralized autonomous airspace controller, it is important to test it against a variety of scenarios. Figure 6.1 and Figure 6.2 depict two possible configurations. The agent in orange is the one currently being affected by the controller action. If there is no agent in red, it means the controller is not taking action on any of the active agents. The orange circle surrounding agents represents their collision threshold which is defined based on the vehicle's wingspan.
Tests on this environment include varying the Poisson rate at which aircraft arrive (this value is constant during training, ensuring an expected separation equal to the outer boundary). Larger rates should decrease performance, while lower rates should increase it as the agent gets more time and space to maneuver. Moreover, as the agent was trained for being agnostic to the intersection configuration, varying the angle between

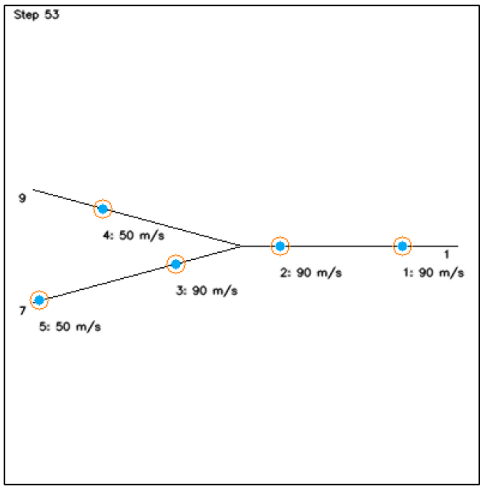


Figure 6.1: Custom research environment with a two-route merger.

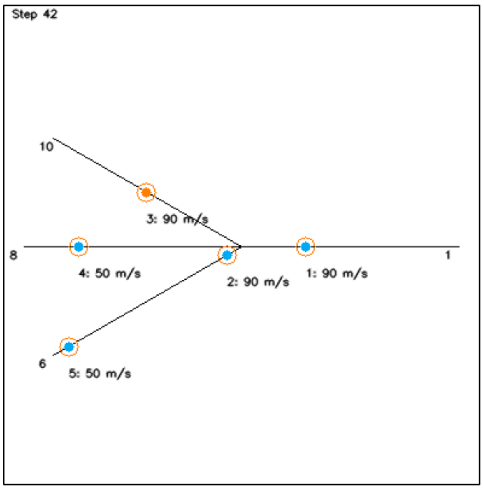


Figure 6.2: Custom research environment with a three-route merger.

entry routes, as well as their number is also interesting to understand how the agent deals with more challenging scenarios such as more routes or routes that are too close to each other, increasing the likelihood of conflict. Finally, it is also important to assess how the agent's actions are distributed in time. Is the agent sending opposite commands to the same aircraft, increasing the probability of error? Is the agent trying to send commands to aircraft that are no longer in the sector? Is the agent minimizing the number of commands concerning the reward function designed? All these questions are important to fine-tune the weights in the reward function.

III

Supporting work

1

Deep Learning Background

DL techniques are at the backbone of our DRL PPO implementation, as NNs are used to approximate both the actor and the critic. In this section, the main DL principles referenced in the paper are shortly explained. First, the activation functions used in the actor and critic networks are defined. Secondly, the Gradient Descent method used to update the network's parameters is covered. Finally, two more complex types of NNs mentioned in the paper, namely CNNs and LSTMs, are described. Typical NNs are constituted by an input layer (i), multiple hidden layers ($h_1, h_2, ..., h_n$), and one output layer (o) that predicts the outputs. A hidden or output layer in a neural network is a set of neurons (also called nodes). Each neuron can be described by a set of weights (w), biases (b), and an activation function (f) to be applied to the input (x):

$$v = f(w^T x + b) \tag{1.1}$$

Where v is the output of the neuron. In the most common type of neural network, the fully-connected ANNs (Artificial Neural Networks), the activation output of one layer serves as the input for the next one. Both actor and critic networks used in SB3-PPO implementation are fully connected with two hidden layers of 64 neurons.

1.1. Activation Function

There are multiple activation functions used in DL. In the context of the implementation considered in this work, we used tanh, and softmax activations. Activation functions are important in NNs to add the capacity of representing non-linear and complex processes. The sigmoid activation function converts input values into outputs between 0 and 1 and is S-shaped [55].

$$f(z) = \frac{1}{1 + e^{-z}} \tag{1.2}$$

The tanh function, or hyperbolic tangent, activation function, is applied in the fully-connected layers of both actor and critic networks and can be defined as:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1.3}$$

The tanh function shape resembles the sigmoid but is symmetric around the origin and the activation values are bounded between -1 and 1 [55]. The softmax function is applied to the $|U|$ nodes in the output layer of the policy network to yield the probability for each discrete action and can be written as:

$$f(z)_j = \frac{e^{z_j}}{\sum_{u=1}^{|U|} e^{z_u}}, j = 1, ..., |U| \tag{1.4}$$

For binary classification problems, the sigmoid function can be used to extract the probability of each class. However, if there are more than two possible output classes, the softmax is used for the same purpose. The

number of neurons in the output layer equals the number of classes. Another activation function that is typically used is the ReLU (or Rectified Linear Unit) function. This function works as a linear activation if the input is greater than 0, and equals 0 if the input is lower or equal to 0 [55].

$$f(z) = max(0, z) \tag{1.5}$$

1.2. Parameter Update

Neural network parameters are updated during training to optimize a loss function based on a method called Gradient Descent. One training cycle over the entire set of samples is called an epoch. One possible approach is Batch Gradient Descent, where the model error is computed with respect to the entire training set, i.e., the network's parameters (θ) are updated only at the end of each epoch by averaging the cost function over the complete dataset.

Algorithm 1 Batch Gradient Descent algorithm (adapted from [49])

for epoch = $1, 2, ..., K$ **do**
  Compute gradient of the loss function w.r.t. parameters θ ($\nabla_\theta J(\theta)$)
  $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta J(\theta)$
end for

A disadvantage of this method is that it is very memory-consuming, which can make it unusable for large sets of training samples. Moreover, it computes unnecessary gradients for similar samples before parameters are updated [49]. As opposed to Batch Gradient Descent, Stochastic Gradient Descent (SGD) updates parameters for every training sample.

Algorithm 2 Stochastic Gradient Descent algorithm (adapted from [49])

for epoch = $1, 2, ..., K$ **do**
  for sample $(x^{(i)}, y^{(i)})$ in data **do**
    Compute gradient of the loss function w.r.t. parameters θ
    $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$
  end for
end for

Therefore, SGD is faster and enables online learning (i.e. learning while new data is coming in). However, it suffers from a high variance and fluctuation of the cost function problem [49]. To combine the strengths of Batch Gradient Descent and SGD, Minibatch Gradient Descent considers the parameter update over a minibatch of n training samples.

Algorithm 3 Minibatch Gradient Descent algorithm (adapted from [49])

for epoch = $1, 2, ..., K$ **do**
  for minibatch $(x^{(i:i+n)}, y^{(i:i+n)})$ in data **do**
    Compute gradient of the loss function w.r.t. parameters θ
    $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
  end for
end for

Minibatch sizes typically vary between 50 and 256. This approach has two advantages. First, it reduces the variance in parameter updates, and second, it can leverage vectorized approaches in modern DL libraries to efficiently compute the gradient w.r.t. a minibatch of data. However, there are remaining challenges, such as avoiding local minima or correctly defining the learning rate during training, that are tackled by more advanced Gradient Descent optimization algorithms [49]. Some of the most relevant are Momentum ([46]), RMSprop ([26]), or the state-of-the-art combination of both, used in the implementation of PPO described in the paper, Adam ([34]).

Momentum is an optimization technique that improves Gradient Descent by reducing oscillations and speeding up convergence. This is done by adding a fraction of the previous update to the current update vector [49].

$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta J(\theta) \tag{1.6}$$

$$\theta = \theta - v_t \tag{1.7}$$

The momentum term γ is usually 0.9. If we compare this equation to a ball rolling down a hill, the analogy is that the ball rolls faster every step until it reaches its terminal velocity (due to $\gamma < 1$). If the hill then climbs, the ball will start losing its speed. Thus, the momentum term increases if the gradient has the same direction as in the previous step, and reduces if the direction of the gradient is altered [49].

Hinton coped with the gradient change during training and across weights by introducing an adaptive learning rate method, RMSprop (Root Mean Squared Propagation). It keeps a moving average of the squared gradient for every weight ($E[g^2]$) [26].

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{1.8}$$

It then improves learning by dividing the gradient by the squared root of this moving average. A small ϵ constant is used to prevent division by 0 [26].

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{1.9}$$

Adam (Adaptive Moment Estimation) is another adaptive learning rate algorithm that keeps an exponentially decaying average of past squared gradients (v_t) like RMSprop and the same type of average of past gradients (m_t) like momentum [49].

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{1.10}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{1.11}$$

Where m_t and v_t are estimates of the first and second moments, respectively. Since these vectors are initialized at 0, bias-correcting terms are defined to adjust in early training iterations [49].

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{1.12}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{1.13}$$

The final Adam update is given as:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{1.14}$$

The following table summarizes the implementation parameters of Adam used in the paper.

Adam Parameter	Value
α	3×10^{-4}
β_1	0.9
β_2	0.999
ϵ	1×10^{-8}
Minibatch size	64
Number of epochs	10

Table 1.1: Training parameters used with Adam and Minibatch Gradient Descent in SB3-PPO implementation [47].

Therefore, the implementation of PPO first collects 4 rollouts of data from 4 different environments. Then, it runs 10 epochs of Minibatch Gradient Descent on minibatches of size 64. For each minibatch, the parameters θ of the actor and the critic networks are updated using the Adam update described before.

1.3. Convolutional Neural Networks

CNNs are typically used when input features are represented as images. A squared black and white image is simply a $N \times N \times 1$ array of numbers. This is because, for each of the N^2 pixels, a value between 0 and 255 in the grayscale spectrum is set. In the case of RGB images, the number of channels is 3. A possible approach to using traditional ANN architectures with images is to flatten this array and consider each pixel entry value as an input feature. There are two main problems with this approach: first, the number of weights in ANNs scales significantly with image size, and second, as a consequence, complex ANNs are more likely to overfit the training data [44].

To understand CNNs, we first have to study the kernel convolution operation. Figure 1.1 depicts the convolution operation between a 3×3 input and a 2×2 kernel with a stride of $(1,1)$ and no padding, meaning the sliding window moves one step at a time and no padding is added to alter the shape of the output. The highlighted value in the output array is computed as $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$. Then, the kernel slides over the input to calculate the other values.

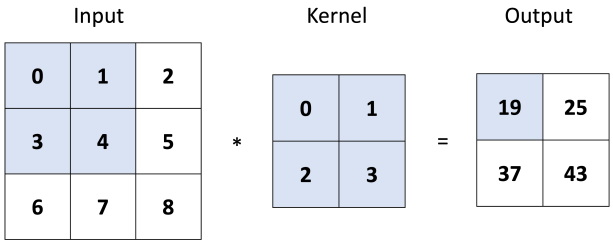


Figure 1.1: Convolution operation example.

In CNNs, the kernel may be viewed as the weight of the convolutional layer. In case the input is three-dimensional, the kernel must have the same depth dimension as the input and the operation is performed in the volume, so the output is always depth 1. Convolutional layers typically contain multiple filters, whose outputs are stacked up in an output volume. An advantage of this approach is that CNNs use parameter sharing across multiple regions of the image [44].

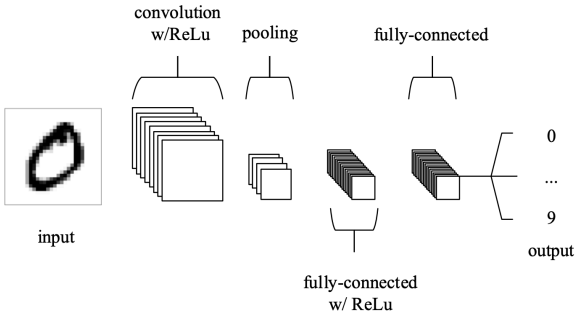


Figure 1.2: Architecture of a CNN [44].

Figure 1.2 depicts a typical CNN architecture. The convolution layer convolves multiple filters with the input, generating activation maps after applying a nonlinearity. Sometimes, zero padding is applied to control the dimensionality of the output. This is only adding 0-value pixels to the border of the output. The pooling layer objective is to reduce the dimensionality of the input and thus, the number of parameters of the output. Typically, max-pooling is applied, selecting the most significant features only. With a 2×2 pooling layer and a stride of 2, the input's height and width dimensions are reduced to 25% of their original size, while maintaining the depth of the volume constant [44].

1.4. Sequence Models

Sequence models can extract relevant patterns from sequences of data, converting single or sequences of inputs into single or sequences of outputs [69]. For instance, in the paper, it is mentioned that Brittain and

Wei used this type of approach to compress variable-length observation information into a fixed-size vector. The simplest sequence model is RNN (Recurrent Neural Network), proposed in [50], which consists basically of multiple copies of the same network, each receiving new sequential inputs and conveying information to the following unit. Each of these networks contains a set of weights and biases that are shared across time steps of the sequence. Thus, both the hidden state ($a^{(t)}$) passed forward and the predicted output of a unit ($\hat{y}^{(t)}$) are computed based on these parameters, the previous activation, and the current time input [69].

$$a^{(t)} = f_1(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a) \tag{1.15}$$

$$\hat{y}^{(t)} = f_2(W_{ya}a^{(t)} + b_y) \tag{1.16}$$

Where f_1 and f_2 are activation functions (typically f_1 is the tanh, and f_2 depends on the output type, it could be a sigmoid or softmax, for instance). W and b are the unit parameters, shared across units, $x^{(t)}$ is the sequence input at timestep t, $a^{(t-1)}$ is the hidden state received from the previous unit, $a^{(t)}$ is the hidden state to be passed over to the next unit, and $\hat{y}^{(t)}$ is the output of the unit, in case the output of the model is sequential. To train RNN, a technique called backpropagation through time (BPTT), introduced in [67], is used, which unenrolls all the unit values to perform the update on the shared parameters. One disadvantage of RNNs is that they cannot easily learn long-term dependencies [69]. In [27], a solution to the gradient vanishing problem or difficulty to remember distant past information is presented, the LSTM.

LSTMs are similar to RNNs, but instead of using the unit cells explained before, use memory cells, containing an internal state and gates, that allow the network to learn what information from the past to retrieve and what to forget. The internal state is can be considered as the memory of the LSTM that is passed forward. It depends, in the first place, on the candidate generated ($\tilde{c}^{(t)}$) based on the input data, the hidden state from the previous cell, and the cell parameters (also shared across cells in this case) [69].

$$\tilde{c}^{(t)} = tanh(W_c[x^{(t)}; a^{(t-1)}] + b_c) \tag{1.17}$$

This candidate is updated using the values of the input and forgets gates, based on how relevant the current input is in the long-term sequence [69]. The output gate is used to compute the hidden state to be passed forward. Considering input, forget, and output gates T_i, T_f, and T_o, respectively:

$$T_i = \sigma(W_i[x^{(t)}; a^{(t-1)} + b_i]) \tag{1.18}$$

$$T_f = \sigma(W_f[x^{(t)}; a^{(t-1)} + b_f]) \tag{1.19}$$

$$T_o = \sigma(W_o[x^{(t)}; a^{(t-1)} + b_o]) \tag{1.20}$$

The candidate update, as well as the hidden state and the output (in the case of a sequential output) values, are computed as:

$$c^{(t)} = (T_i * \tilde{c}^{(t)}) + (T_f * c^{(t-1)}) \tag{1.21}$$

$$a^{(t)} = T_o * tanh(\tilde{c}^{(t)}) \tag{1.22}$$

$$\hat{y}^{(t)} = f(W_y a^{(t)} + b_y) \tag{1.23}$$

Where $*$ represents the element-wise product. Therefore, in an LSTM, besides the hidden states, cells also pass internal states forward. T_i controls how important is the input in the long-term learning sequence, while T_f determines how important is the information about previous time steps [69]. In the specific case of handling variable observation space lengths, LSTM networks can be used over agents, instead of time, given that agents are sorted in such a way that the most relevant ones come last in the sequence [9].

2

Custom Gym Environment Verification

The verification of the environment developed took two steps: confirming it follows the Gym API rules and ensuring that the interaction between the agent and the model is mathematically correct, i.e., confirming the state transition and reward values of a step on the model: $(x_k, u_k, x_{k+1}, r_{k+1})$. The first step is straightforward as SB3 contains a function to perform this evaluation. The second step is performed by first computing the tuple values by hand and then checking them against the model outputs. For that, we considered a simple scenario where $|\mathcal{N}| = 3$. Aircraft ID 1 is on the exit route, $500m$ away from the merging point at a speed of $90m/s$. Aircraft 2 is on route 9 (check the sector image in the paper for reference on route numbers), $50m$ from the merging point at $80m/s$. Lastly, aircraft 3 is on route 7, $1200m$ from the merging point at $60m/s$. This information was added to the environment by setting aircraft coordinates and velocities. The action chosen is to increase the speed of aircraft 3 (a_3^+). At this point, we have all the required information to compute the next state and reward value and compare them to the output of the environment. Aircraft 3 normalized state is given as:

$$[r_3, v_3, d_{3,i}] = [\frac{165-0}{360-0} \cdot 2 - 1, \frac{70-30}{90-30} \cdot 2 - 1, \frac{1130-0}{1800-0} \cdot 2 - 1] \approx [-0.08, 0.33, 0.26] \tag{2.1}$$

Where 165° is the angle route 7 makes with the exit route (measured counter clock-wise), $70m/s$ is the updated speed, and $1130m$ is the updated distance. Following the same rules for aircraft 1 (0 route, $90m/s$ velocity, and $590m$ distance) and aircraft 2 (0 route - transitioned from entry to exit route - $80m/s$ velocity, and $30m$ distance), we obtain the final next state value:

$$X_{k+1} \approx [-1, 1, -0.34, -1, 0.67, -0.97, -0.08, 0.33, 0.26] \tag{2.2}$$

To verify the reward function an identical procedure is used. The action reward is -0.1 as the action is not $None$. The speed reward is computed by:

$$r_v = \frac{1}{3} \cdot 0.01 \cdot (70 + 80 + 90) = 0.8 \tag{2.3}$$

To compute the separation reward, we first have to determine the distances between all pairs of aircraft: $d_{1,2} \approx 560m$, $d_{2,3} \approx 1159m$, and $d_{1,3} \approx 1707m$. Since only $d_{1,2}$ is lower than the outer boundary, only this value is considered in the reward:

$$r_s = \frac{1}{3} \cdot 0.005 \cdot (560 - 1000) \approx -0.73 \tag{2.4}$$

Thus, the final reward is $r_{k+1} = 0.8 - 0.1 - 0.73 = -0.03$. It was verified that both the state and reward values matched the output of the environment, proving that the transition and reward functionalities are working properly.

3

Obtaining Stable Simulation Outputs

Obtaining stable simulation results both for the average success rate and average time to cross parameters is important so that our analysis is statistically relevant. For this, the stabilization of the coefficient of variation was used:

$$CV = \frac{\sigma}{\mu} \tag{3.1}$$

Where σ is the standard deviation of the data sampled and μ is the mean. This shows what is the level of dispersion around the mean with the current number of samples. We recomputed this value every time 100 new data points were sampled. Based on the graphs below, it was determined that the number of required samples to compute an accurate estimate of the average success rate with or without the agent deployed is $6k$, and $4k$ for the average time to cross the sector per aircraft.

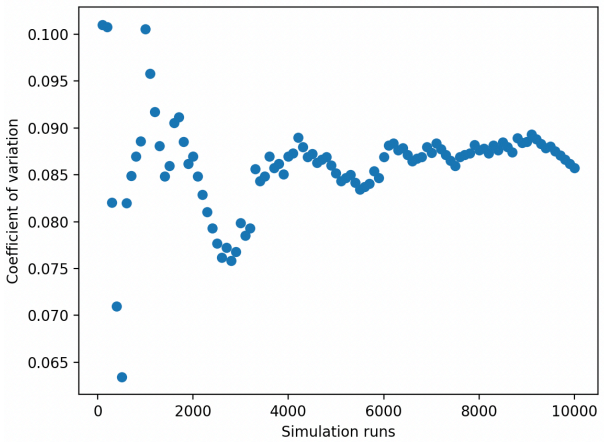


Figure 3.1: Evolution of the coefficient of variation for the success rate with the trained agent.

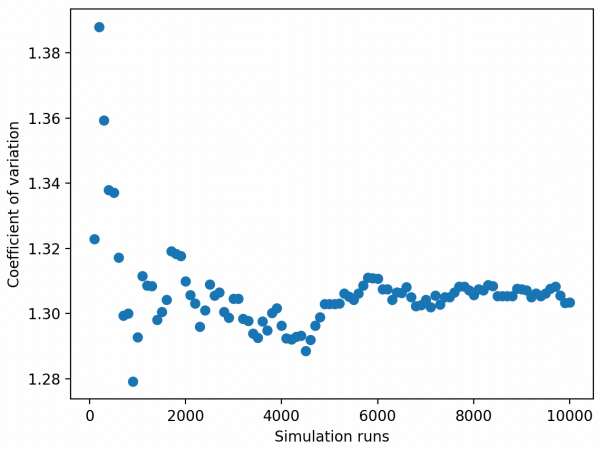


Figure 3.2: Evolution of the coefficient of variation for the success rate without the trained agent.

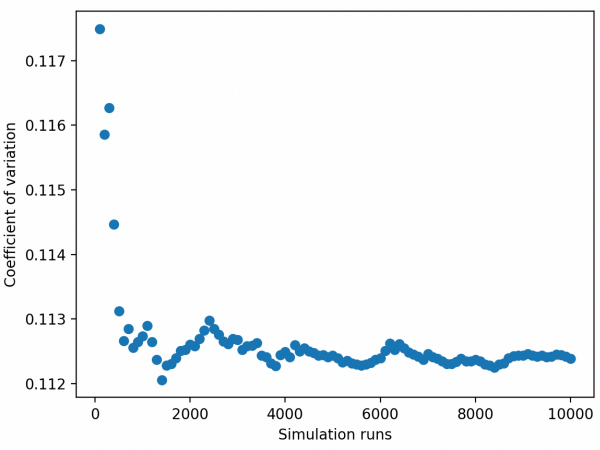


Figure 3.3: Evolution of the coefficient of variation for the time to cross the sector with the trained agent.

Bibliography

[1] Federal Aviation Administration. Faa aerospace forecast: Fiscal years 2020-2040; u.s. department of transportation. 2020.

[2] Unmanned Airspace. Airbus launches blueprint utm roadmap, predicts 19,269 drones an hour above paris in 2035. 2018. URL https://www.unmannedairspace.info/uncategorized/airbus-launches-blueprint-utm-roadmap-predicts-19269-drones-hour-paris-2035/.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[4] International Civil Aviation Association. Doc 4444: Air traffic management - procedures for air navigation services. 16th ed, 2020.

[5] Joby Aviation. Joby receives part 135 certificate from the faa. 2022. URL https://www.jobyaviation.com/news/joby-receives-part-135-air-carrier-certificate/.

[6] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[7] Marc Brittain and Peng Wei. Autonomous aircraft sequencing and separation with hierarchical deep reinforcement learning. *International Conference on Research in Air Transportation*, 2018.

[8] Marc Brittain and Peng Wei. Autonomous air traffic controller: A deep multi-agent reinforcement learning approach. *arXiv preprint arXiv:1905.01303*, 2019.

[9] Marc Brittain, Xuxi Yang, and Peng Wei. A deep multi-agent reinforcement learning approach to autonomous separation assurance. *arXiv preprint arXiv:2003.08353*, 2020.

[10] Marc W Brittain and Peng Wei. One to any: Distributed conflict resolution with deep multi-agent reinforcement learning and long short-term memory. In *AIAA Scitech 2021 Forum*, page 1952, 2021.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[12] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[13] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[14] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2): 57–83, 2002.

[15] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[16] Yevgen Chebotar, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine. Combining model-based and model-free updates for trajectory-centric reinforcement learning. In *International conference on machine learning*, pages 703–711. PMLR, 2017.

[17] Cosimo Della Santina, Jens Kober, Ivo Grondman, and Robert Babuška. Reinforcement learning. 2022.

[18] Elhadji Amadou Oury Diallo, Ayumi Sugiyama, and Toshiharu Sugawara. Learning to coordinate with deep reinforcement learning in doubles pong game. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 14–19. IEEE, 2017.

[19] Jakob N Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*, 2016.

[20] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[21] Sven Gronauer and Klaus Diepold. Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review*, 55(2):895–943, 2022.

[22] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International conference on autonomous agents and multiagent systems*, pages 66–83. Springer, 2017.

[23] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[24] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[25] Ammar Haydari and Yasin Yilmaz. Deep reinforcement learning for intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[26] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8):2, 2012.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Zhang-Wei Hong, Shih-Yang Su, Tzu-Yun Shann, Yi-Hsiang Chang, and Chun-Yi Lee. A deep policy inference q-network for multi-agent systems. *arXiv preprint arXiv:1712.07893*, 2017.

[29] George Hunter and Peng Wei. Service-oriented separation assurance for small uas traffic management. In *2019 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, pages 1–11. IEEE, 2019.

[30] Dae-Sung Jang, Corey A Ippolito, Shankar Sankararaman, and Vahram Stepanyan. Concepts of airspace structures and system analysis for uas traffic flows for urban areas. In *AIAA Information Systems-AIAA Infotech@ Aerospace*, page 0449. 2017.

[31] Antoine Joulia, Thomas Dubot, and Judicael Bedouet. Towards a 4d traffic management of small uas operating at very low level. In *ICAS, 30th Congress of the International Council of the Aeronautical Sciences*, 2016.

[32] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.

[33] Brad Bachtel Karen Dix-Colony. Operating the 747-8 at existing airports. URL https://www.boeing.com/commercial/aeromagazine/articles/2010_q3/pdfs/AERO_2010_q3_article3.pdf.

[34] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[35] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[36] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[37] Andrea Lonza. *Reinforcement Learning Algorithms with Python*. Packt Publishing, 2019.

[38] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[39] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[40] Matternet. Matternet receives faa production certificate for its m2 drone delivery system. 2022. URL https://mttr.net/images/Matternet_FAA_Production_Certificate_20221130.pdf.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[42] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, 1990.

[43] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE transactions on cybernetics*, 50 (9):3826–3839, 2020.

[44] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[45] Gregory Palmer, Karl Tuyls, Daan Bloembergen, and Rahul Savani. Lenient multi-agent deep reinforcement learning. *arXiv preprint arXiv:1707.04402*, 2017.

[46] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1): 145–151, 1999.

[47] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021.

[48] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 4295–4304. PMLR, 2018.

[49] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[50] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[51] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[52] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[53] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[54] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[55] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[56] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[57] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[58] Alex M Stoll, Edward V Stilson, JoeBen Bevirt, and Percy P Pei. Conceptual design of the joby s2 electric vtol pav. In *14th AIAA Aviation Technology, Integration, and Operations Conference*, page 2407, 2014.

[59] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[60] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.

[61] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[62] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[63] Rose E Wang, Michael Everett, and Jonathan P How. R-maddpg for partially observable environments and limited communication. *arXiv preprint arXiv:2002.06684*, 2020.

[64] Zhuang Wang, Weijun Pan, Hui Li, Xuan Wang, and Qinghai Zuo. Review of deep reinforcement learning approaches for conflict resolution in air traffic control. *Aerospace*, 9(6):294, 2022.

[65] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[66] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[67] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78 (10):1550–1560, 1990.

[68] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.

[69] S Zargar. Introduction to sequence learning models: Rnn, lstm, gru. *no. April*, 2021.

[70] Zongzhang Zhang, Zhiyuan Pan, and Mykel J Kochenderfer. Weighted double q-learning. In *IJCAI*, pages 3455–3461, 2017.

[71] Peng Zhao and Yongming Liu. Physics informed deep reinforcement learning for aircraft conflict resolution. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[72] Yan Zheng, Zhaopeng Meng, Jianye Hao, and Zongzhang Zhang. Weighted double deep multiagent reinforcement learning in stochastic cooperative environments. In *Pacific Rim international conference on artificial intelligence*, pages 421–429. Springer, 2018.