Martin Aalby Svalesen
Eilef Olsen Osvik

Learning-Based Collision Avoidance for Micro Aerial Vehicles in Confined Environments

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Martin Aalby Svalesen
Eilef Olsen Osvik

Learning-Based Collision Avoidance for Micro Aerial Vehicles in Confined Environments

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

The work for this master thesis was carried out in the spring semester of 2021 at the Department of Engineering Cybernetics at NTNU [1] in Trondheim, Norway.

This master thesis presents a novel solution to the local path planning problem in geometrically constrained and GPS-denied environments like caves and subterranean infrastructure. The problem is formulated as a data-driven Imitation- and Reinforcement learning algorithm. Data is gathered in simulation, using a LiDAR mounted on a quadrotor MAV. The learning based methods are executed by imitating an expert and through reinforcement learning with a specially tailored reward function. The thesis can be broken into two parts which have been explored by the authors in parallel: *perception* and *quadrotor control*. This master thesis as a whole represents an End-to-end method for doing collision-free navigation and flight control using learning-based methods. However, the two underlying components, perception and control, can also be seen as two independent pieces of work.

The following master thesis is a continuation of two project theses written by the authors during the fall of 2020. The title of these theses were: "Reward shaping in Quadrotor Control Using Deep Deterministic Policy Gradients"[2] and "Robustness in Deep Reinforcement Learning for Quadrotor Control"[3]. The experience gathered in these two projects plays a crucial role in developing the methods in this project.

The previous research on this topic has been led by Prof. Kostas Alexis and the research team at the Autonomous Robots Lab (ARL) [4], and it originates from the DARPA Subterranean Challenge [5].

Our supervisor Kostas Alexis has, throughout the semester, guided our research through bi-weekly meetings. In addition, Huan Nguyen has been a great help along the development process for understanding the existing code base from the NTNU Autonomous Robots Lab on top of which our own software is written.

The software package corresponding to the Convolutional Variational Autoencoder is written from scratch and is made available at [6]. The expert planner used for imitation learning is is part of the workspace from ARL [7], and the reinforcement learning and imitation learning algorithms have been forked from OpenAI [8]. The frameworks used for simulation is RotorS [9] and Gazebo [10]. All the neural networks have been built using the Tensorflow framework [11] version 2.0, and the whole system has been tied together with the Robot Operating System (ROS) [12].

Trondheim, 07.06.2021

Eilef Olsen Osvik
Martin Aalby Svalesen

Acknowledgment

We want to thank our supervisor Kostas Alexis for providing us with an interesting research project, and being a very helpful guide through the whole process from project thesis through to the end of the master thesis. His expertise in autonomous robotics has helped us reach very far in a short period of time during our last year at NTNU. A very special thanks to Huan Nguyen for being a patient and competent support through software development.

E.O.O and M.A.S

Executive Summary

This thesis explores a method of solving the local path planning and control problem in an End-to-end fashion using learning-based methods.

We present a novel compression algorithm that substantially reduces the dimensionality of the raw point cloud data from a 3D laser scanning sensor using a Convolutional Variational Autoencoder. The strength of such a compression algorithm is a radical reduction in dimensionality, which results in the data stream being reduced in size by three orders of magnitude. The weakness of the compression algorithm is its limitation to only work in a subset of object types and environments. Further work needs to be done to be able to generalize to more complex environments.

This compressed representation of the immediate environment is fed to a learning-based agent, parametrised as a neural network, together with the direction of the desired setpoint and the MAV speed. This agent is then taught through both Imitation Learning and Reinforcement Learning to decode this compressed representation of the environment to be able to execute collision-free trajectories. Our work indicates that the agent is able to infer a geometric understanding of the environment from the heavily compressed representation. Furthermore, the algorithm executes at a constant frequency of about 12 Hz, as opposed to traditional graph-based methods where the frequency varies greatly with both time and the complexity of the environment. However, several challenges are yet to be solved in both the perception and the control part of this thesis before this algorithm is able to provide collision-free trajectories as consistently as the graph-based local planner we use as baseline. We used simulation to train and evaluate both the Convolutional Variational Autoencoder's performance and the agent's ability to execute collision-free trajectories. This let us continually monitor the performance of the CVAE by looking at the reconstructed point clouds, as well as observing the trarjectories of the agent.

Sammendrag

I denne mastergradsoppgaven presenterer vi en ny måte å navigere et ubemannet flygende fartøy i trange omgivelser ved hjelp av laserbaserte avstandsmålere. Vi bryter denne navigasjonsprosessen inn i to deler, persepsjon og kontroll.

I den første delen som omhandler persepsjon, viser vi en ny måte å komprimere punktskydata ved hjelp av en lærings-basert metode. Denne punktskyen definerer hvordan de nære omgivelsene til fartøyet ser ut topologisk, fra synspunktet til fartøyet selv. Den ubehandlede datastrømmen fra den laserbaserte avstandsmålingen blir komprimert med 3 størrelsesordener etter å ha blitt behandlet av vår kompresjonsmetode. Målet med denne kompresjonen er å hente ut de viktigste topologiske egenskapene til de nærliggende omgivelsene slik at en kontrollalgoritme ikke skal trenge å bruke tid på å finne disse egenskapene selv. Svakheten til denne kompresjonsalgoritmen er at den foreløpig kun har høy virkningsgrad i et begrenset knippe med typer miljøer og objekter. Videre utvikling trengs for å få denne algoritmen til å generalisere til mer komplekse og usette miljøer.

Når punktskyen har blitt komprimert av persepsjonsmodulen, blir dataen videreført til kontrollalgoritmen. Denne modellen er parametrisert som et nevralt nettverk, og sammen med den komprimerte punktskyen blir den matet fartøyets relative posisjon i forhold til målet den skal til, samt dens egen hastighet. Modellen blir så lært å fly kollisjonsfritt gjennom to separate prosesser; læring gjennom imitasjon og belønningsbasert læring gjennom utforskning. Gjennom disse læringsprosessene blir modellen implisitt lært til å unngå kollisjoner, ved å optimere kontrollfunksjonen for å fly uten å kollidere med overflater i miljøet. Vårt arbeid indikerer at denne modellen er i stand til å hente en forståelse om de nærliggende omgivelsene fra den komprimerte punktskyen fra persepsjonsmodulen. Navigasjonsalgoritmen vi presenterer i denne oppgaven kjører med en konstant hastighet på rundt 12 Hz, i motsetning til tradisjonelle metoder der hastigheten varierer mye med både tid og kompleksiteten til miljøet fartøyet befinner seg i. Imidlertid er det fortsatt et sett med utfordringer som må løses før denne navigasjonsalgoritmen konsekvent leverer like gode resultater som de tradisjonelle metodene vi bruker som sammenligningsgrunnlag.

For å trene og evaluere agentens evnen til å utføre kollisjonsfrie baner, samt generere datasett for å trene kompresjonsalgoritmen, brukte vi simuleringsverktøy. Dette lot oss kontinuerlig monitorere ytelsen til kompresjonsalgoritmen ved å kunne følge med på den rekonstruerte representasjonen av omgivelsene, i tillegg til å kunne observere hvilken bane den lærings-baserte agenten valgte.

Contents

Preface . . . i
Acknowledgment . . . ii
Executive Summary . . . iii
Sammendrag . . . iv

1 Introduction **2**
1.1 Background . . . 2
1.1.1 Motivation . . . 2
1.1.2 Background of the Thesis . . . 3
1.1.3 Problem Formulation . . . 3
1.1.4 Related Work . . . 3
1.2 Objectives . . . 5
1.3 Approach . . . 5
1.4 Contributions . . . 6
1.5 Limitations . . . 6
1.6 Experimental Setup . . . 6
1.7 Outline . . . 8

2 Fundamental Theory **9**
2.1 Machine Learning . . . 9
2.1.1 Artificial Neural Networks . . . 10
2.1.2 Overfitting and Underfitting . . . 13
2.2 Kulback-Leibler Divergence . . . 15

3 Feature extraction from point cloud data **16**
3.1 Motivation . . . 16
3.2 Problem Formulation . . . 16
3.3 Related Work . . . 17
3.4 Theoretical Background . . . 18
3.4.1 Convolutional Neural Networks . . . 18
3.4.2 Point Cloud Measurements . . . 21
3.4.3 Occupancy Maps . . . 23
3.4.4 Signed Distance Fields . . . 24
3.4.5 Compression Methods . . . 26

3.4.6 Variational Autoencoders . . . 28
3.5 Proposed Method . . . 33
3.5.1 Convolutional Variational Autoencoder . . . 34
3.5.2 Computational Graph . . . 37
3.6 Results . . . 37
3.6.1 Training Methodology . . . 37
3.6.2 Metric . . . 39
3.6.3 Performance of Different Network Configurations . . . 39
3.6.4 Effect of Compression on the Data Stream . . . 39
3.6.5 Generalisability of Networks . . . 40
3.7 Discussion . . . 42
3.7.1 Model Architecture . . . 42
3.7.2 Visual Evaluation of Model Performance . . . 43
3.7.3 Implementation Evaluation . . . 45
3.8 Conclusion . . . 46

4 Learning-based methods for collision avoidance 48
4.1 Motivation . . . 48
4.2 Related Work . . . 49
4.3 Problem Formulation . . . 50
4.3.1 End-to-end Versus Modular Training . . . 51
4.3.2 Objective of the Chapter . . . 52
4.4 Theoretical background . . . 52
4.4.1 Quadrotor Dynamics . . . 52
4.4.2 Reinforcement Learning . . . 54
4.4.3 Imitation Learning . . . 62
4.4.4 Expert Planner . . . 64
4.4.5 The Data Aggregation Algorithm (DAgger) . . . 65
4.4.6 Differences Between the Imitation Learning Problem and the Reinforcement Learning Problem . . . 66
4.4.7 Desired Behaviour of the MAV . . . 67
4.5 Proposed Method . . . 67
4.5.1 The Agent . . . 67
4.5.2 Environments . . . 71
4.5.3 Autoencoder . . . 71
4.5.4 Evaluation Metric . . . 72
4.5.5 Reward Function Design and Implementation . . . 73
4.6 Limitations in the Implementation Process . . . 75
4.6.1 Topics of Discussion Regarding the Expert Planner . . . 75
4.6.2 The Imitation learning Problem and the Reinforcement Learning Problem . . . 81
4.7 Results . . . 82
4.7.1 Computational Complexity . . . 82
4.7.2 Imitation Learning . . . 83

4.7.3 Reinforcement Learning . . . 84
4.8 Discussion . . . 85
4.8.1 Computational Complexity . . . 85
4.8.2 Imitation Learning . . . 86
4.8.3 Reinforcement Learning . . . 92
4.9 Conclusion . . . 97

5 Conclusions 99
5.1 Summary . . . 99
5.2 Conclusion . . . 100
5.3 Discussion and Recommendations for Further Work . . . 100
5.3.1 Ensure Generalisability of the Encoder . . . 100
5.3.2 Memory of Objects Disappearing From Direct View . . . 101
5.3.3 Challenges with Respect to the Imitation Learning and the Reinforcement learning Problems . . . 101
5.3.4 Validation Loss for the Agent . . . 102
5.3.5 Ambiguous Learning Samples From the Expert Planner . . . 102
5.3.6 Randomising the Environment . . . 102

A Acronyms 103
A.1 Algorithms . . . 105
A.2 Specifications . . . 107

Chapter 1

Introduction

Autonomous exploration of GPS-denied environments is an active field of research. The two main challenges in these types of environments are: how to map the surroundings in such a way that a local representation of position is achieved, and how to use this representation to navigate safely without colliding. Since we assume no prior knowledge of the map is available, the robotic agent itself needs to understand the environment and be as being able to remember where it has gone in the past. Thus, we can break the problem of exploration down to two parts; *global exploration planning* and *local path planning*. In the *global exploration planning*, the agent remembers the whole trajectory it has traversed. This makes it able to re-plan, should the robotic agent come to a dead end and need to find a path towards a previously detected area far from its current location. The planner can then remember a previous intersection where a branch remains unexplored and navigate the robotic agent back through its traversed path to begin exploring this branch. In the *local path planner* however, the objective is finding the immediate path or control action for the robotic agent, which fulfils some goal in the short term. This goal may be to either reach a waypoint while avoiding obstacle on the way, or finding a path that maximises the environment's explored volume.

1.1 Background

1.1.1 Motivation

Autonomous exploration of geometrically constrained and GPS-denied environments has a broad appeal outside the academic sphere. It enables exploration of environments that may either be dangerous for humans or hard to reach. In addition, the autonomy ensures that the robotic agent can independently explore without needing communication with a human operator, which further facilitates deeper and more efficient exploration. This may be very useful for first-responders looking for human survivors, or inspecting underground or enclosed infrastructure like subway tunnels or mines.

The algorithms used for these scenarios today are very computationally expensive. This naturally limits their speed and the rate at which the exploration is conducted. However, recent advances in the fields of machine learning and its adaptation into high-performance edge devices motivate us to try new approaches to this challenge. Instead of sampling the environment and running expensive path optimisation algorithms, a learning-based way of navigating and controlling an Micro Aerial Vehicle

(MAV) collision-free must exist. After all, even tiny insects with tiny brains can infer a geometrical understanding about their immediate environment using just their eyes.

1.1.2 Background of the Thesis

The work in this thesis is based on two related projects conducted during the fall of 2020. These projects aimed to understand and construct Reinforcement Learning-based methods for MAV control in a collision-free environment. The first author investigated the impact the network structure had on performance and robustness, and the second author investigated the impact the reward shaping had on the learning process. The problem solved for both projects was basic waypoint navigation in a collision-free environment. The knowledge obtained from these two projects was essential in developing the strategies and methods used in this master thesis.

1.1.3 Problem Formulation

The problem of this master thesis is two-fold:

1. How do we infer a geometric understanding of the immediate environment using learning-based methods?
2. How do we use this understanding of the environment to design a control algorithm which executes a collision-free trajectory?

1.1.4 Related Work

We can broadly classify classic 3D path planning algorithms into categories, examples of which are sampling-based methods, trajectory optimisation methods and motion primitives-based methods.

In [13], a simple on-line real-time approach to collision avoidance in an uncertain environment is proposed. The path planning is based on polar coordinates, where the desired direction angle is used as an optimisation index. At every planning window, the robot senses obstacles and orients the robot direction of travel to generate a sub-goal. After the obstacle has been avoided, the robot will control its direction of travel to coincide with the direction to the desired waypoint. This method keeps the robot travelling straight towards the desired waypoint, and will only make detours to avoid objects.

In [14], the authors present two sampling-based algorithms for optimal path planning, namely the PRM* and the RRT*. These algorithms are shown to be asymptotically optimal which means that the cost of the return solution almost surely converges to the optimum, in contrast to their predecessors PRM and RRT. Using a known global map, these algorithms are able to efficiently calculate paths towards a desired waypoint.

In [15], the author presents a search-based path planning approach using motion primitives to plan resolution complete and sub-optimal trajectories. This makes the proposed paths cognisant of the dynamics of the MAV, which ensures that the proposed paths are feasible when navigating complex 3D environments. To explore the robot's control space, this method employs a random sampling which it then uses to determine possible paths in the known 3D map of the environment.

In [16], the authors present an algorithm called SPARTAN, which exploits the different levels of spatial distributions of obstacles in the environments in which it is operating. The algorithm creates a

sparsely connected graph across the tangential surface around obstacles, which lets it plan its trajectory especially fast whenever traversing environments with sparsely distributed obstacles. Using only the limited sensing capabilities on the MAV, and without a previously known map, the algorithm can find paths in real-time through unstructured outdoor environments.

In [7], a graph-based global and local planner is developed for autonomous exploration. It consists of a local planner which samples the surrounding space, constructs graphs, and calculates the path in which no collisions occur while the discovered volume is maximised. A global planner maps and incrementally builds a global map of the environment while remembering the traversed trajectory. This makes it able to guide the robotic agent to a new area of exploration should it reach a dead end.

In [17], a local path planner based on motion primitives is developed. Instead of sampling points in space, the planner samples possible control actions and their resulting trajectories. This enables the planner to take advantage of the agility of flying robots, further increasing the range. For each sampled control action, the paths generated are checked for collisions and the gain in newly explored volume is maximised.

In [18], a local learning-based path planner is developed. Instead of sampling trajectories, it uses an imitation learning framework for teaching a convolutional neural network agent to suggest exploration directions. The main contribution to this path planner, is removing the expensive operation of path sampling, collision checking and calculating the newly explored volume, as done in [17] [7].

In [19], the authors present a continuous-time polynomial formulation of a local trajectory optimisation problem for collision avoidance. This motion planning method is used as a local re-planner which recomputes safe trajectories as new obstacles are discovered.

In [20], a local path planner uses the concept of a Safe Flight Corridor, a collection of convex overlapping polyhedra that models free space and provides a connected path from the robot to the goal position. This representation is built from a piece-wise linear skeleton obtained using a fast graph searching technique and is input into a Quadratic Program (QP) for real-time re-planning on a receding horizon planning problem.

In [21], the authors implements an anytime planner based on the Anytime Dynamic A* algorithm that performs a graph search on a four-dimensional (x,y,z,heading) lattice. The graphs are constructed using motion primitives to ensure feasibility and exploit the dynamics of micro UAVs, while the collision checking may be done with arbitrary vehicle shapes.

In [22], the authors present a novel Nonlinear Model Predictive Controller (NMPC) for collision avoidance and local path planning. The novelty of their implementation is the modelling of dynamic obstacles using linear models, which permits the UAV to plan both its own trajectory and the obstacle trajectory forward in time. Furthermore, the dynamic collision avoidance is coupled with the control layer, which means that the trajectory of an obstacle is fed directly to the optimiser as a parameter, together with the control parameters.

In [23], the authors model the UAV as a floating object and are thus not relying on full odometry information as position information may drift over time in harsh environments. They develop a LiDAR-based navigation system with collision avoidance capabilities using Nonlinear Model Predictive Control, and use a geometric approach to obtaining the correct heading rate towards the open area to explore. The resulting system is a local path planner.

In [24], the authors present a novel Nonlinear Model Predictive control framework for autonomous navigation in indoor enclosed environments. Using the NMPC framework, the nonlinear dynamics of

the UAV and the nonlinear geometric constraints of the objects in the environment may be taken into account. To extract features from the LiDAR point cloud, a subspace clustering method is used. These features are then incorporated into the nonlinear constraints of the NMPC for collision avoidance.

In [25], the authors present a framework for dynamic path planning for Unmanned Aerial vehicles. This work combines the RRT* algorithm with the Dynamic Window Approach to solve dynamic obstacle avoidance in a partially unknown environment. Using the RRT* as the global path planner to avoid static objects in the map, commands are issued to a local path planner using the Dynamic Window Approach, which works as the local path planner. As a result, the local path planner senses dynamic obstacles and makes sure that the trajectory is collision-free.

In [26], the authors present a path and motion planner based on a local perception of the UAV to autonomously generate safe 3D paths in dynamic cluttered environments. The path generation algorithm is formulated as a multi-criteria real-time optimisation process that aims to minimise and objective function while satisfying the dynamic constraints of the UAV as well as the safety requirements. The safety constraints are implemented as a minimal-security distance.

In [27], the authors present a 3D path planning algorithm for UAV formation based on a comprehensively improved Particle Swarm Optimisation (PSO) algorithm. The initial distribution of particles is determined using a chaos-based logistic map, and the particles also undergo a mutation strategy where undesired particles are transformed into desired ones. The method is validated using simulation in environments with the terrain and treat constraints.

In [28], the authors present an algorithm for collision-free trajectory planning of multi-rotor UAVs in windy conditions based on modified potential fields. To overcome the local minima issue in artificial potential fields, an augmented objective function is considered to find an optimal trajectory for the UAV during point-to-point maneuvers in a region with apriori known wind speeds.

In [29], the authors show an efficient path planning algorithm for high-speed MAV in unknown environments. Their work proposes a computationally efficient algorithm using online sparse topological graphs, which also integrates a notion of long-term memory into the planner. It may choose to navigate to locations that are no longer included in the local map. To ensure that a path is collision-free, a motion primitives-based local receding horizon planner that uses a probabilistic collision avoidance methodology is implemented.

1.2 Objectives

The goal of this master thesis is to develop and test a local learning-based motion planner which allows for fast navigation in confined environments. It should detect and safely avoid obstacles while navigating to the desired waypoint while reducing the computational complexity substantially.

1.3 Approach

The algorithm will receive raw LiDAR point clouds generated by a lightweight mid-range LiDAR [30]. These point clouds will then be integrated into a local projective Truncated Signed Distance Field (TSDF) representation using the fast method in Voxblox [31]. This TSDF representation will be compressed using a Convolutional Variational Encoder, which has been trained as part of a Convolutional Variational Autoencoder(CVAE). During training, the CVAE is optimised to reconstruct the TSDF

point cloud, indicating the quality of the compression in the bottleneck layer. During inference, one extracts this compressed representation from the encoder and feeds it as part of the observation space to the learning-based agent.

This learning-based learning agent has been taught to navigate through imitation learning (IL) with an expert planner and a Reinforcement Learning (RL) process. The agent makes decisions only based on its inertial states, like distance to the goal point and speed, and using the *latent space* from the encoder. The output of this agent is an acceleration vector which is then fed to the low-level controllers of the MAV.

1.4 Contributions

The first contribution of this thesis is a novel framework [6] for training Convolutional Variational Autoencoders on point cloud data from a Truncated Signed Distance Field (TSDF) representation. This includes the full pipeline from the training process to the actual ROS-node running the encoding operation, and multiple tools for monitoring and visualising the encoder performance.

To the best of the authors' knowledge, this thesis presents the first attempt at doing learning-based navigation using a 360° 3D TSDF representation from LiDAR scans and a Convolutional Variational Autoencoder. This thesis will thus contribute to the understanding of how well a learning-based agent is able to learn and generalise from the latent space of such a system.

1.5 Limitations

Our implementation is specific to a given class of environment. Therefore, our work is only applicable to MAVs with similar LiDAR systems, and may have to be revised if a different hardware configuration is to be used.

The Convolutional Variational autoencoder is trained using data samples collected in a very specific environment with a limited set of different types of obstacles. Therefore, there is no guarantee that this model will generalise to environments that are very different. However, as this is a very early proof of concept for this system of algorithms, it is expected that future research will build upon our results and experience and be able to generalise further.

1.6 Experimental Setup

For both the training process of the learning-based agent and the training of the CVAE, simulation tools were fundamental for us to be able to gather enough data. Since these are both learning-based methods, we needed a significant amount of realistic data samples to emulate a real-world situation. Using simulation let us speed up gathering training data for the CVAE, and simulating the flight and collisions of a quadrotor without damaging any equipment in the process. The software tools used for simulation and visualisation are the following:

Robot Operating System (ROS)

ROS [12] is an open-source middleware framework used for simulating and controlling robotic applications, connecting the different software and hardware components. ROS integrates many features like message passing, control of low-level components and package management, which are all important for robotics. During the work on this thesis, we used the ROS Melodic Morenia version.

Rviz

Rviz [32] is a visualization toolkit that enables 3D renderings of environments, sensor data and coordinate transforms. Throughout this thesis, this tool has been used to visualise the trajectories of the learning-based agents, and visualise the point clouds in the CVAE pipeline.

Gazebo and RotorS Simulator

The simulation engine Gazebo[33] (version 9.0.0) was used together with RotorS for the simulation of the quadrotor flight, LiDAR operation and the environment. Gazebo is a sophisticated 3D robot simulation tool with an advanced physics engine and rendering capabilities. This simulator facilitates accurate sensor and actuator simulation, allowing for safe and realistic robotics development.

RotorS [9] is Gazebo simulator package specifically designed for MAV simulation. It is developed by the Autonomous Systems Lab at the ETH university in Zürich. The package contains mountable sensors, controllers and environments relevant for multirotor simulation.

Tensorflow

The open-source machine learning library Tensorflow version 2.0 [11] was used for this project. Tensorflow is developed by Google and is a highly optimised framework for training and running neural network models. Both the learning-based agent and the Convolutional Variational Autoencoder were implemented using this framework.

Voxblox

To make both global and local Truncated Signed Distance Field (TSDF) maps, we used the Voxblox software package [31]. The CVAE is trained to operate on the local TSDF maps from this software package, while the expert used for imitation learning used the global TSDF map.

Convolutional Variational Autoencoder library

Since no similar software library existed at the time of this research, a software library for training Convolutional Variational Autoencoders on point clouds was created [6]. This software library contains functionality for gathering datasets from point cloud streams with which it can later train a given autoencoder structure. A ROS-node for integration into the complete learning-based navigation pipeline was created. This node compresses the TSDF point cloud data stream from the Voxblox software package and sends the compressed representation to the learning-based agent. The reconstructed point cloud can optionally be continually monitored visually using Rviz.

OpenAI

OpenAI Gym [8] is a toolkit designed for RL research. During the research for this thesis, we have used version 0.13 to test different RL methods and to visualise their performance. Their implementation of the PPO algorithm from their baselines repository [34] was used in this project.

1.7 Outline

The master thesis is structured as follows: First, we have a chapter that establishes the theoretical understanding needed for both of the later chapters. Then the perception part of the project is presented in the second chapter, "Feature Extraction From point cloud Data". This chapter discusses how the raw LiDAR point cloud is intelligently compressed by extracting features and smoothing out sensor noise. The third chapter, "Learning-based methods for Collision Avoidance", discusses how these LiDAR features are used to train a learning-based agent. The agent's purpose is to propose low-level control commands, which leads to a reliable collision-free navigation to reach a pre-determined waypoint. This agent will be trained using both an Imitation Learning framework and a Reinforcement Learning framework.

Chapter 2

Fundamental Theory

2.1 Machine Learning

Machine learning is a field of computer science where the goal is to fit statistical models to a dataset. This will later make the computer able to do inferences with unseen data samples to generate a prediction or action. This process is often executed as an algorithm that only depends on human operators for basic parameters, and it thus has an interesting decoupling between the computer and the human operator. Within the field of machine learning, there are three main directions: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning

In supervised learning, we are fitting a statistical model to a labelled dataset. Thus the model is explicitly told what output a particular set of input data should generate. The result is a model that for an input sample, should be able to infer what the output should be [35, p.9].

Unsupervised learning

In unsupervised learning, we are not working with a labelled dataset[35, p.485]. This means that it is the job of the algorithm itself to find patterns in the dataset, which can then be used to classify or segment the input. Since this process is independent of human interaction, apart from setting up basic parameters, the process can yield interesting results previously unimagined by any human.

Reinforcement learning

Reinforcement learning (RL) is a sub-field of machine learning which differs from supervised and unsupervised learning. This is because the model we are training, often called the agent, is exploring the state space at the same time as it is learning[36, p.1]. This contrasts with the two earlier methods, where a complete and balanced data set may be used to train the model. To generate good samples to learn the model on, the model itself must explore the state space. This is a balance of *exploration* and *exploitation*, where the agent will alternate between locally optimal and sub-optimal strategies to ensure proper exploration.

2.1.1 Artificial Neural Networks

The name of Artificial Neural Networks (ANNs) is inspired by the fact that their function is loosely analogous to how a biological brain is constructed. At its core, ANNs are function-approximators that map a specific input distribution to a desired output distribution. What makes them so influential is their versatility in solving a very wide range of inference problems, and their complexity is easily controlled through varying the topological design of the network. They are finding their way into fields of science that historically have been dominated by clever hand-crafted mathematical models like computer vision and fraud detection. Given a properly distributed and descriptive dataset, the ANNs learn autonomously what features to look for in the data and may find subtle patterns, invisible to humans, which have a large predictive value. At the same time, they are considered "black box" models. This is because as the complexity increases from just a few layers and a few neurons, it is impossible to tell exactly what features and patterns the network is finding. We thus have little control over what goes wrong and what goes right when the networks are doing their predictions; i.e. if the network is confidently making a wrong prediction, it is impossible to know why it did this. The only way to guarantee a certain success rate of the networks is to test them on previously unseen data samples exhaustively.

Perceptron

The perceptron is the building block of the neural network model architecture, often called *neuron* in the fully connected context when several are chained together. It is often called a neuron because of its similarity to a *biological neuron*. In the real biological neurons, the *dendrite* receives electrical signals from the *axons* which are modulated in various amounts in the *synapses*. The neuron only fires an output signal if the total strength of the input signals exceeds a given threshold. All this behaviour is modelled in the mathematical perceptron model, using weightings of input signals and an activation function. This is illustrated in figure 2.1.

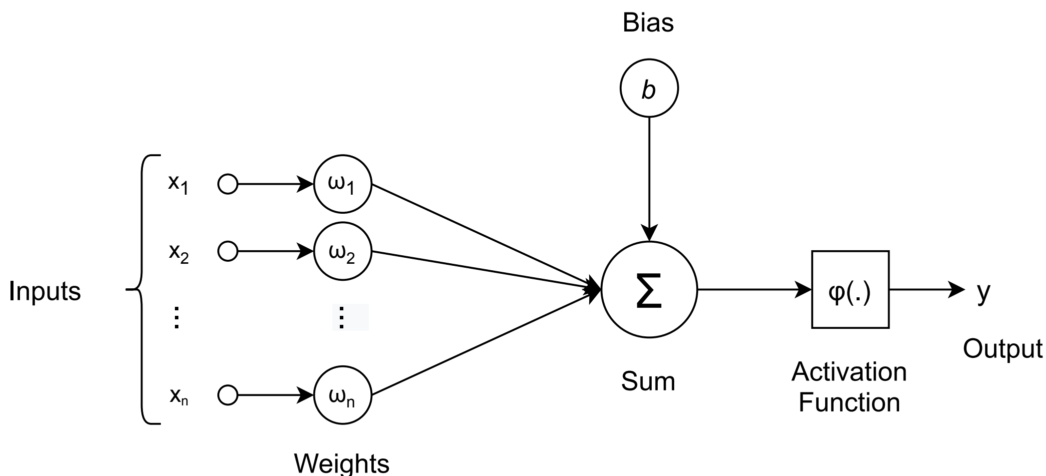


Figure 2.1: The basic components of a perceptron

As we can see in equation 2.1, we have a vector of inputs x_m with corresponding input weights ω_m and an additional *bias* term b. These weights are changed to reflect their predictive value to the

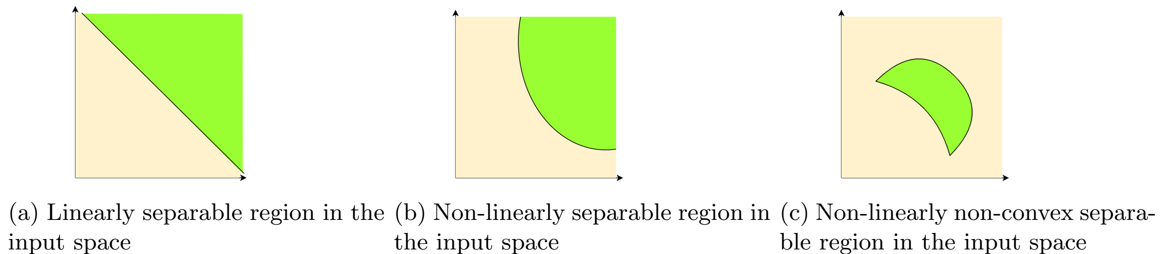
desired output. The bias variable is used as a free variable with which the training is able to explore a larger area of the solution space. When these terms are summed together, they are passed through an activation function $\phi(.)$. Two commonly used activation functions are the ReLU activation function, as seen in equation 2.2, and the Sigmoid activation function, as seen in equation 2.3. Depending on the use case, we may choose different types of activation functions.

$$y = \phi(x_1\omega_1 + x_2\omega_2 + ... + x_m\omega_m + b) \tag{2.1}$$

$$R(x) = \begin{cases} x & if \quad x > 0, \\ 0 & otherwise \end{cases} \tag{2.2}$$

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

In the original paper on the perceptron by Rosenblatt in 1958 [37], a simple step function is used as the activation function. By using only one perceptron, we can do classification based on *linearly separable regions* in the input space. An example of this can be seen in figure 2.2a, where the linear black line separate two regions in the input space.



(a) Linearly separable region in the input space (b) Non-linearly separable region in the input space (c) Non-linearly non-convex separable region in the input space

Figure 2.2: Separability of different complexities of input spaces

Even though this is a simplistic example, it shows the capabilities of a single *neuron*. Using a nonlinear activation function, we are able to make a nonlinear line in the input space for the activation of a single neuron, as seen in figure 2.2b. However, we are not able to solve problems where the solution region in the input space is non-convex, i.e. if we have several separate regions, or a non-convex region, in the input space that corresponds to the same output. For this, we will need the *fully-connected multilayer perceptron* model architecture.

Multilayer Perceptron

The multilayer perceptron model (MLP) [38], or the *neural network* as it is often called, consists of chaining together multiple *neurons* together and thus creating a fully connected network.

In figure 2.3, we can see an elementary neural network with one input layer, one "hidden" layer, and one output layer. Between each node, there is a connecting line that corresponds to a signal. Each node in the hidden layer and the output layer receives a signal from all previous nodes, and the

activation of a single node corresponds to the activation given by equation 2.1. Now we are able to make predictions based on non-convex and nonlinear patterns in the input space, an example of which can be seen in figure 2.2c.

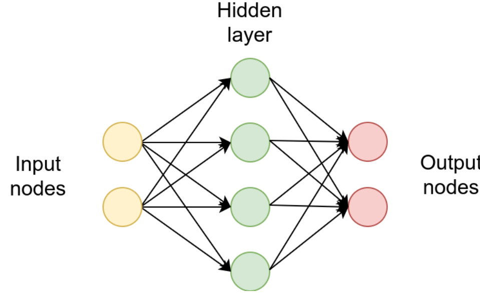


Figure 2.3: An example of a basic neural network architecture

Let us consider the ReLU activation function in equation 2.2. This piece-wise linear function has been extensively used throughout this thesis as it can overcome the vanishing gradient problem of deep networks and capture highly nonlinear behaviour. Even though it can facilitate for deep network, the ReLU function can have an issue of immobilising large parts of the network as the output of a neuron is zero for any inputs less than zero. Still, it is found to allow for easy optimisation using gradient descent as the gradients can flow through the layers for inputs larger than zero [39].

Gradient-based training

When we are "training" a neural network, all it means is that we are changing the weights and biases of all the inputs in the neurons. For a set of input samples, we want to generate an output that correctly corresponds to a given value. To do this, we need to make sure that the values of all the weights and biases are determined in such a way that if we pass a sample through the network, the mathematical calculation yield values close to the ones we want.

When the network is initialised, all the values are scrambled, and the output should be pure noise without much predictive value. The first step of training the network is comparing the output values of the network to the desired output values. For this we use a *loss-function*. Depending on the domain, we may use a loss function like the squared error as seen in equation 2.4 or the Binary Crossentropy as seen in equation 2.5. What all the loss functions have in common is that they produce small values if the output of the network $\hat{y}_i$ is close to the correct value y_i, and large values if the output deviates too much. This now means that we know for each sample what the correct output value should be, and thus in which directions the outputs of the network should move.

This algorithm is called *stochastic gradient descent*, and the *stochastic* part refers to the fact that we do not know the true gradient, but we are estimating the gradient using sampling [40, p.177].

$$L_{MSE}(\vec{\omega}, \vec{b}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.4}$$

$$L_{BinaryCrossentropy}(\vec{\omega}, \vec{b}) = -\frac{1}{n}\sum_{i=1}^{n} y_i log\hat{y}_i + (1 - y_i)log(1 - \hat{y}_i) \tag{2.5}$$

Backpropagation

Now that we know in which direction we want to change the output values of our neural network, we can begin changing the weights and biases in our network. To do this, we use the algorithm *backpropagation* [41]. This algorithm computes the *gradient* in the *weight space* of a neural network with respect to a loss function. Or more intuitively; in what way do we need to push the weights and biases inside the network to move the output values of the neural network closer to the desired ones. When evaluating the network, the inputs and the outputs change, and the weights and biases are fixed. When training, the inputs and outputs are fixed, and the weights and biases change. We denote the neural network as:

$$g(x) := f^L(W^L f^{L-1} \dots f^1(W^1 x) \dots)) \tag{2.6}$$

where:

- x: input data
- $W^L = (w^l_{jk})$: the weights between layer $l-1$ and l, where w^l_{jk} is the weight between the k-th node in layer $l-1$ and the j-th node in layer l
- f^l: activation functions at layer l

When training, we have a set of input-output pairs (x_i, y_i), with the y_i being the value we want the network to predict. The loss is then the difference between this desired output and the actual output:

$$C(y_i, g(x_i)) \tag{2.7}$$

Backpropagation calculates the gradient of this loss function with respect to the model weights, $\frac{\partial C}{\partial w^l_{jk}}$, for a given input-output pair (x_i, y_i). Instead of doing this iteratively through the chain rule, the algorithm computes the gradient for *each layer* which saves a lot of duplicate computation. The fully connected nature of the model lets us do this, as every neuron in a given layer depends on all neurons in the previous layer. This means that there will necessarily be many shared calculations.

After these gradients have been calculated using backpropagation, we apply a *gradient descent* step with step size η as shown in equation 2.8. This process then repeats until some convergence criterion is met.

$$-\eta \nabla C(\vec{\omega}, \vec{b}) = [\nabla\omega_1, \nabla\omega_2, ..., \nabla\omega_m, \nabla b_1, \nabla b_2, ..., \nabla b_t] \tag{2.8}$$

An important property of this process is that the magnitude of these gradients gets smaller and smaller as the gradients are propagated through the layers. Practically, this means that the last layers in the network will learn faster, and as the gradient propagates backwards through the network, the layers will learn at an increasingly slower rate. This property is called the *vanishing of gradients*, and this is the reason that large networks require longer training times than shallower ones.

2.1.2 Overfitting and Underfitting

Two very important concepts in machine learning are *overfitting* and *underfitting* [40, p.110]. Overfitting refers to the situation in which the neural network has learned the whole dataset by heart instead

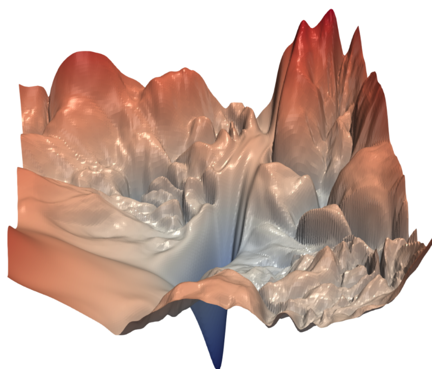


Figure 2.4: An illustration of a loss landscape from the paper [42]

of properly generalising. This means that it will be very good at predicting correct values to input samples it has seen before or input values that lie very close to values seen before in the input space. However, a sample which is not very close in the input space to something it has seen before will not be classified correctly.

The opposite may also happen. If a model is underfitted to its training data, it does not accurately predict the correct output values and thus has not captured enough of the fine-grained patterns in the training data.

An interesting property of neural networks is that these training behaviours are closely related to how the neural network is constructed. If we have a neural network with many layers and many nodes in each layer, the network is generally able to detect more fine-grained patterns in the input data. If the neural network is shallow, the granularity of the patterns and features it can learn to recognise is coarser. Therefore, it is important to make sure that the complexity of the neural network matches the complexity and feature granularity in the training data. This was explored and demonstrated in the project thesis *"Robustness in Deep Reinforcement Learning for quadrotor control"*.

To monitor under- and overfitting, we use a *validation loss* together with the training loss explained in the previous section. The validation loss is calculated using a dataset with which the network has not trained on. This serves as an indication of how well the model is able to generalise. As long as the training loss and validation loss are decreasing, we can be sure that what the network is currently learning is transferable to unseen data samples. However, when the two start to diverge, the network is beginning to overfit. This means that the network is learning the training data by heart.

The reason that deeper networks are more prone to overfitting, is that as the complexity of the architecture increases, so does the complexity of the loss landscape. Practically, this means that a complex network will often have several local minima corresponding to the same loss value. It is very difficult to know whether the minimum the network optimisation process has reached is a global minimum. This is directly tied to the networks *granularity* as explained earlier. An example of a highly non-convex loss landscape can be seen in figure 2.4.

Another result of overfitting, is the problem of *overparametrisation*. This refers to the process in which too many parameters are describing a simple mathematical relationship. If a mathematical relationship between two variables is linear, describing that linear relationship with more than two numbers leads to overparametrisation. A consequence of this is that the learning outcome of the

training process may become unstable, as the network is not able to generalise simple mathematical relationships to outside certain regions it has trained for.

2.2 Kulback-Leibler Divergence

The Kulback-Leibler divergence D_{KL} [43] (often abbreviated *KL-divergence*) is a mathematical measure of the similarity between two given probability distributions.

Definition

Let us define two probability distributions P and Q, which exist on the same discrete probability space $\mathcal{X}$. We then say that the *relative entropy* from Q to P is defined as:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) log\left(\frac{P(x)}{Q(x)}\right) \tag{2.9}$$

This can thus be interpreted as the *expectation* of the logarithmic difference between the probability distributions P and Q, where the expectation is calculated using the probability distribution P. This relative entropy is only defined if the two random distributions are *absolutely continuous* with respect to each other. Practically, this metric can be used to precisely quantify the *similarity of two different discrete random distributions*.

Chapter 3

Feature extraction from point cloud data

3.1 Motivation

To enable autonomous navigation using a MAV in a confined space, we need a way to understand the environment. Since there is no prior map or a Global Positioning System, we only rely on the onboard sensors to develop some idea of where the MAV currently is in the 3D environment. Understanding the environment in which the MAV is situated will help it make decisions that enables safe trajectory planning and execution without collisions. There exist many discrete algorithms for creating a representation of the environment for collision avoidance, however, getting a good 3D understanding of the environment using these methods is very computationally expensive. This is why we turn to a learning-based approach. If there exists an intelligent way of understanding the immediate environment based on a learning-based method, this could save a lot of computation.

3.2 Problem Formulation

The data collected by the MAV is in the form of a point cloud from a LiDAR system. This data contains a discretised description of the continuous 3D environment, and the resolution of the LiDAR scan determines the coarseness in the detection of 3D textures. Each data point in the point cloud represents a coordinate to an obstacle in 3D space, seen from the body frame of the MAV. This data stream is a high-dimensional representation and is in itself computationally expensive to use directly as input to a navigating algorithm. The challenge then is to extract the necessary features that the navigating algorithm needs to fly without collisions. It is not necessarily the case that a reduced dimensionality representation of 3D space has the features we need for collision avoidance. Therefore, we need to extract the relevant features for collision avoidance, and discard all unnecessary information. How we do this, is precisely the problem we will be looking at in this chapter.

3.3 Related Work

In [44], the authors present a set of Visual-Inertial Odometry methods which estimates the position, attitude, and velocity of a robotic agent by using one or more cameras and the internal Inertial Measurement Units (IMUs). This enables the robotic agent to navigate without any global positioning system like GPS, given that the environment in which the robotic agent operates contains a sufficient amount of visually discernible features.

In [45], the authors propose an algorithm for 3D odometry and mapping for a robotic agent in real-time using range measurements from a 2-axis LiDAR. They divide the algorithm into two parts; the first subroutine performs odometry estimation at a high frequency but with low fidelity to estimate the velocity of the robotic agent, while the second subroutine does fine matching and registration of the point cloud for mapping at a lower frequency. The resulting system can provide the robotic agent with both information about its own intertial states like position and velocity, and provide an incrementally built global map that the agent can use for high-level path planning.

In [46], the authors present a multi-sensor fusion estimation algorithms for a MAV that will be entering and exiting buildings. To enable the perception of the MAV position in a global map, the estimation algorithm fuses data from an IMU, laser scanner, stereo cameras, pressure altimeter, and a GPS receiver. This makes sure that when the MAV enters a building and loses GPS reception, it can still estimate its global position as it will map its surrounding area using the laser scanner and stereo camera.

In [47], the authors present an efficient probabilistic 3D mapping framework based on octrees called OctoMap. It is an explicit representation that maps occupied, free, and unknown space. This representation is compressed using an octree representation to minimise memory consumption and efficiently update the 3D map incrementally. Since 3D range measurements have an inherent level of uncertainty, the method employs a probabilistic approach to integrating new observations, ensuring consistency in the incremental construction of the global map.

In [48], the authors present "VDB", a hierarchical data structure for efficient representation of sparse, time-varying volumetric data discretised on a 3D grid. The representation exploits spatial coherency of time-varying data to separately and compactly encode data values and grid topologies. This results in a representation that supports fast random access with an average of O(1) to insert, retrieve or delete data points.

In [49], the researchers propose a method called Voxblox. The representation of the map is a Euclidean Signed Distance Field (ESDF), where each point of the map is the distance from the MAV. This allows for efficient use path optimisation algorithms, while also being readable for humans. Furthermore, the maps are efficiently constructed from TSDF point clouds allowing real-time applications for MAVs.

In [50], the authors show how to teach a robotic hand to grasp objects in 3D space. They show that it is possible to understand the 3D environment in a 3D convolutional neural network (CNN) which applies to collision avoidance. Their CNN shows an ability to understand collision risks when input a Truncated Signed Distance Function (TSDF) voxel representation of the 3D environment.

In [51], the authors propose a framework for extracting features from a 3D point cloud using voxel-grid-based downsampling, kd-tree-construction for efficient 3D range searching and a novel modification of the Euclidean clustering approach for computationally inexpensive object detection.

In [52], the authors present a lossy compression method for compressing a TSDF voxel representation to a smaller dimension space. They use a Principal Component Analysis approach followed by an autoencoder for the compression. They show that the latent space at the bottleneck contains sufficient information to reconstruct a map enabling estimating the ego-motion of the camera.

In [53], the authors propose an efficient subspace clustering technique to extract planes or surfaces that are presented in a point cloud. These are then fed as geometric constraints to a nonlinear Model Predictive Controller for collision avoidance.

In [54], a compression method for 3D LiDAR point clouds is proposed. Residual blocks are used for a near-lossless compression of the point cloud. Minimising the Symmetric Nearest Neighbor Root Mean Squared Error (SNNRMSE) and the bits per point representation is used to train the encoder-decoder. The residual blocks are found to allow for higher performance than other point cloud compression techniques.

In [55], a framework is made to give hints to humans attempting to land a MAV. The framework consists of a Cross-Modal Variational Auto-encoder (CM VAE) and a learned policy generated by the TD3 algorithm. The policy receives as input the latent space of the CM VAE, and the authors show that the policy can extract a geometric understanding of the environment using this compressed representation.

In [56], a 2D point cloud representation is obtained using a LiDAR. This is then fed into a CNN, which directly controls the steering commands of a ground-based robotic platform. It is able to robustly navigate a cluttered environment to get to its goal.

In [57], the authors compressed a 3D voxel grid using first an octree representation and then later a Variational Autoencoder. The compression method was tested and proven efficient by its scene reconstruction as well as for its performance in path planning.

In [58], the authors present a Dynamic Graph CNN for learning point cloud representations called *EdgeConv*. Instead of working on individual points, the algorithm exploits local geometric structures by constructing a local neighbourhood graph and applies *graph neural networks* to the edges connecting pairs of points.

3.4 Theoretical Background

A point cloud from a LiDAR scan represents a very fine-grained, high-dimensional 3D representation of the immediate surroundings of the autonomous vehicle. When doing trajectory planning, operating directly on these point clouds is currently a very computationally expensive process to do in real-time, as the point cloud may include millions of points. This limits the speed at which a MAV can traverse the environment. This section will discuss popular methods for down-sampling this high-dimensional sensor input into representations that are useful for motion planning.

3.4.1 Convolutional Neural Networks

A central challenge faced by data scientist when working with visual data has always been the curse of dimensionality. This refers to the fact that as the dimensions of the input data increases, the number of needed training samples increase aggressively. Visual data is often of high dimensions, with a low-resolution VGA video signal of 600x400 spanning 240000 discrete values. Having that many nodes

(or more) in the first layer of a neural network makes it very hard and computationally expensive to extract features and generalise the input data into different categories, which is often what we want to do. This network would need to have many deep consecutive layers and would be cumbersome to work with.

A Convolutional Neural Network solves this problem by introducing the concept of convolutional layers. These layers have a set of filters that they learn. This represents a feature extraction process that actively selects the relevant features from the input data necessary to do the prediction. These convolutions can work on a range of different type of input data, from 1D time-series data to 2D images, and in our case, 3D convolution on a volumetric environment representation. When these features have been extracted, the fully connected neural network then learns to piece together what combinations of features constitute whatever it is trying to predict.

Let us have a closer look at convolutions:

Convolution

Mathematically, convolution is defined as:

$$(f * g)(t) = \int_{\infty}^{\infty} f(\tau)h(t-\tau)d\tau \tag{3.1}$$

where f and g are two separate functions. This represents the integral of the product of the functions after reversing and shifting one of the input functions. This operation has a very pleasing intuitive explanation. Imagine one of the functions being fixed in place and the other function being moved from $-\infty$ to ∞. The value of a convolution in any given point in time is just the shared area under their curves.

Discretising this into the 1-dimensional convolution case, we can represent this as a simple vector of numbers, called a kernel, being passed over a string of numbers. The resulting convolution is the dot product between the numbers of the kernel and the numbers the kernel has "landed on".

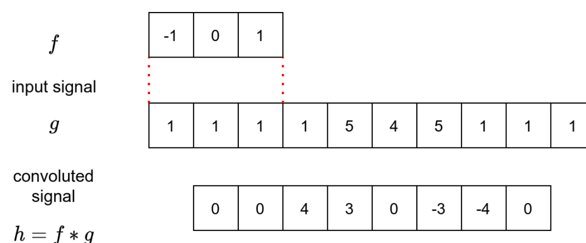


Figure 3.1: An example of a 1-dimensional convolution

In figure 3.1 we can see an example of one such convolution example. Notice how the kernel f is now working as a feature extractor on the input signal g; it generates a signal h which reflects the derivative of the input signal g. Notice how the output signal h is 0 everywhere the input signal g stays constant, positive when the input signal g increases, and negative when g decreases. This is therefore a simple edge detector kernel.

If we generalise this to 2 dimensions, we are able to detect edges in 2 dimensional data like an

image. A popular edge detection kernel is the Sobel operator:

$$\boldsymbol{G_x} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.2a}$$

$$\boldsymbol{G_y} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.2b}$$

This edge detector extracts the magnitude of the horisontal and vertical component of edges in images. If we want the total magnitude of an edge in an image we can use the following formula:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.3}$$

and for the angle of the edge, the following formula:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.4}$$

This is just an example of a popular kernel used for edge detection and shows how combining the output of several kernels can be used to extract higher-order features. This is precisely how the Convolutional Neural Network operates, the only difference being that these kernels are often learned through a gradient descent optimisation method instead of being hard-coded. The CNN will learn to recognise the features necessary to fit the network to the training data. This makes the process independent from prior knowledge about the input data as well as human intervention, which in itself is a significant advantage.

Stacking many spatial convolutional layers after one another has the effect of approximating higher-order features. The first layer of the network recognises small and finely-granulated features from the input data, which then can be assembled into representations of larger areas by the following convolutional layers. This concept of *local connectivity* ensures that features clustered close to each other in the input data may be associated with each other in the following higher-order features, while features collected at each end of the input image will not. This is in stark contrast to the case of the fully connected neural network, where all input features are related to each and every other input feature. This concept applies to both spatial (in the case of images) and temporal (in the case of time series) dependencies in the input data. CNNs are thus much more efficient at extracting and combining features.

As each filter is passed over the entire input image, the shared-weight architecture of the convolution kernels ensures that its weight vector and bias stays the same. This means that the generated feature map corresponds to a specific filter, and two equal features at different parts of the input image will give the same response. The practical implication of this is that the resulting feature map is equivariant under the shifts of the location of the input features. A specific feature may exist anywhere in the input image, but its activation will still be the same. This shared-weight architecture will also guarantee an implicit *regularisation* in the network architecture. Having a limited number of filters, the convolutional layers will have to be designed to capture the most significant features in the input data to make accurate predictions. Thus, they are much less prone to overfitting, as the model is

incentivised to not capture the noise in the training data set, but rather the main feature structures. The model is also incentivised to keep the magnitude of the values inside the filters bounded, as a drastically high magnitude in one of the filters would extract unwanted features in other parts of the image.

A vanilla CNN is illustrated in figure 3.2. An input image is convoluted with a set of filters, which results in a new set of *feature maps*. These maps contain the most basic features collected from the image through the first convolution layer. Subsequently, a new set of filters are applied to the first layer of feature maps to produce the second, and so forth. Notice how the feature maps decrease in dimension at every iteration, exactly like it did in our simple 1D example in figure 3.1. As the height and width of the feature maps decrease, the depth of the feature map stack increases. In the end, we have a vector of feature maps that only measure 1x1, and this can now be considered our extracted feature vector. It is then the job of the fully connected neural network to learn what combinations of features constitutes whatever it is trying to predict.

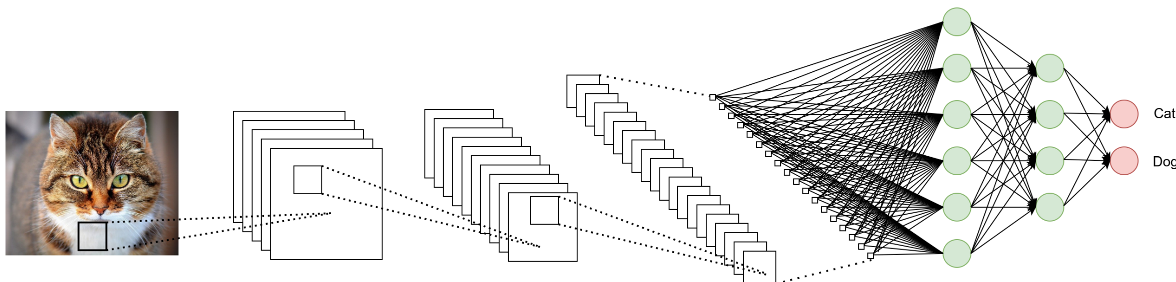


Figure 3.2: Standard setup of a Convolutional Neural Network

Pooling

As previously mentioned, when a convolution is applied to input data, the resulting convolved output data often has reduced dimensionality. This is beneficial as it decreases the computational power needed to process the data. We can also explicitly do dimensionality reduction through a process called *pooling*. The goal of the pooling layer is to reduce the dimensionality of the feature space while preserving the necessary features.

Practically, one first chooses a kernel size. In the example figure 3.3, we can see that the kernel size is 2x2. We distribute the kernel over the input image without overlapping. Then we can either choose the **maximum** value inside each box, or choose the **average** of the values. These operations are called *Max pooling* and *Average pooling*. In both cases, the input data dimensionality is reduced while preserving a portion of the data. In both cases, the noise levels are reduced. However, in practice *Max pooling* is the most common of the two, as it will select the strongest activation, i.e the strongest feature, to pass through. *Average pooling* however, will average the results inside each pooling area and may thus diffuse strong activations resulting in a weaker feature map.

3.4.2 Point Cloud Measurements

The input to our overall learning-based local planner framework is point clouds. As the name suggests, this is a data structure that consists of a set of discrete points in 3D space. These points may be encoded

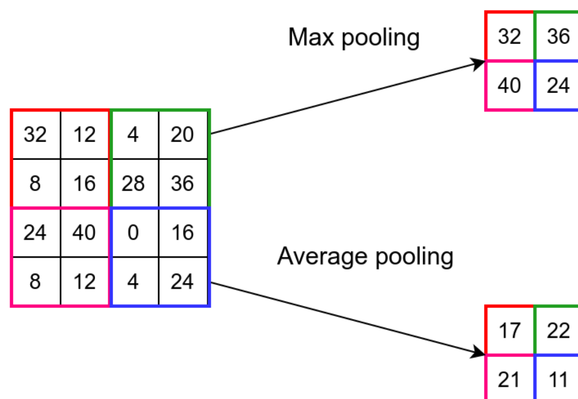


Figure 3.3: Illustration of a basic pooling operation

with information like intensity, or color in RGB. Sampling a point cloud of the immediate environment will give a sparse representation of the 3D environment that the agent is currently in. A wall, for instance, will produce a set of points that all span out a plane, while a rugged tunnel wall will give a set of points that appear more chaotic. A collection of point clouds may be used to create a mesh, which consists of creating polygon surfaces between neighboring points in the cloud. One can thus build a coarse reconstruction of the 3D geometry of the environment.

LiDAR Scanning

Generating the point cloud may be done in several ways. However, in this master thesis we will focus on generating point cloud data using a LiDAR ("light detection and ranging"). This device measures the time of flight for a laser beam between emission from the device to the reception. Knowing the time of flight of the laser beam, one can calculate the distance travelled. Knowing both the distance travelled and the angles at which the beam was emitted, one can calculate the position of a point in 3D space.

The LiDAR device cleverly integrates this into a unit as we see in figure 3.4. Several laser beams are placed on top of each other at different angles. These beams are then rotated 360 deg, and at a given angular interval, the laser beams are fired and return distances to the objects at which the beams are aimed. This ensures that the 3D environment around the robot is mapped. It is important to note that the LiDAR has a limited field of view. In our case, the beams exiting the LiDAR unit has a field of view of $\pm 22.5°$. This means that objects outside of this field of view are not directly observable, which has direct implications to how a robot may perceive its world.

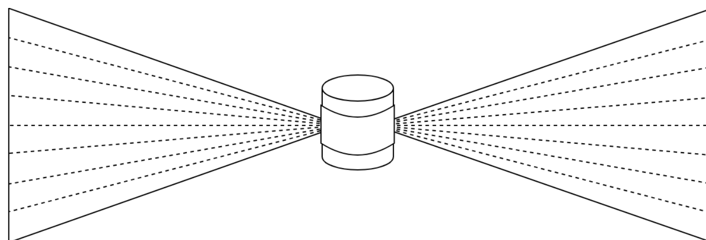
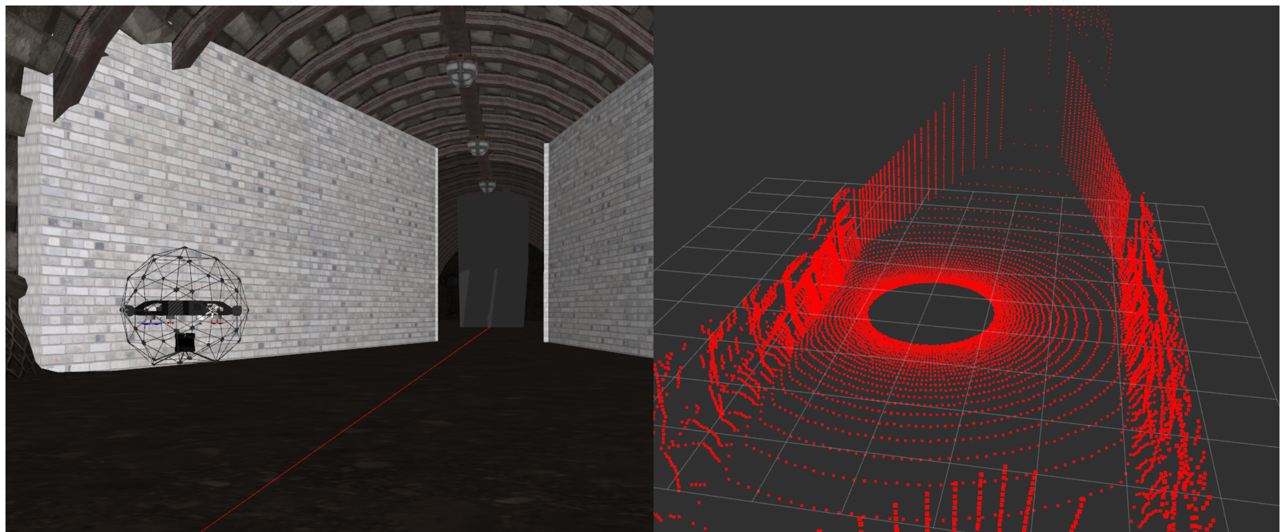


Figure 3.4: Basic geometry of a LiDAR range scanner



(a) A tunnel environments with walls

(b) The LiDAR point cloud scan

Figure 3.5: An example of how the point cloud is made

3.4.3 Occupancy Maps

The concept of occupancy maps is straightforward. First, a volume is discretised into cells that are called *voxels*. Then, as the sensor input is received, the cells that happen to be at the same spot as an obstacle or surface, are marked as black. These cells are thus "occupied", and after this process has been repeated for all the cells in the volume, a map is created that represents the 3D volume. The size of each voxel determines the resolution and granularity of the occupancy map, with high-resolution maps being able to capture detailed 3D structures with higher fidelity. This occupancy map will then be a much more useful representation of the 3D environment than the high-dimensional point cloud. It is an explicit surface representation of the 3D volume, meaning that each black voxel directly represents a surface.

In figure 3.6, we can see an example of an occupancy map in 2 dimensions. The undiscovered boxes are grey, the discovered free space is white, and the discovered surfaces are black. The object that is being discovered is dotted in red.

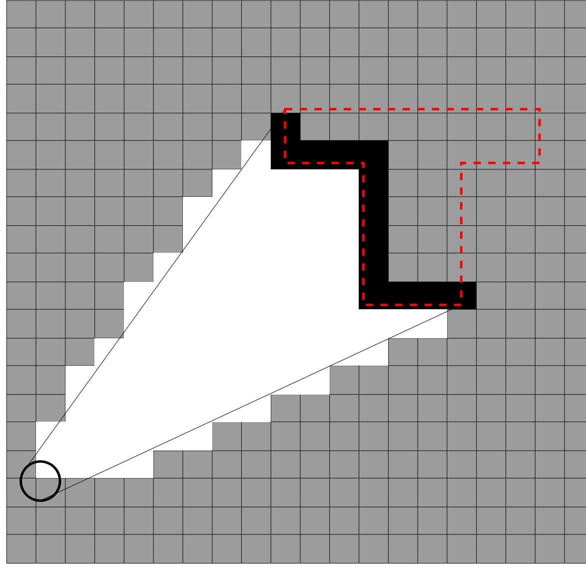


Figure 3.6: A 2-dimensional example of an occupancy map

3.4.4 Signed Distance Fields

A Signed Distance Field (SDF) on the other hand, is an implicit surface representation. It is based on a discretised volume of voxels just like occupancy maps, but each voxel is now colored according to the distance to the nearest surface. This representation is implicit because it describes the space around surfaces, meaning that surface positions and orientations can be inferred indirectly. While this may seem counterintuitive, it is a very useful representation for computer graphics and motion planning, as we will later see.

Truncated Signed Distance Fields (TSDF)

While an SDF is a very useful representation of 3D space, it is not trivial to construct. The reason for this is the limited sensing capabilities that an autonomous vehicle has as it is traversing space. From one point of view, it is impossible to know anything about an object other than what can be observed directly. This includes the interior of the object itself and the backside. Thus, generic signed distance fields need to be constructed using several sensor readings covering a sufficient amount of the 3D space.

An alternative to this is to make a projective signed distance field. This method consists of measuring the distance from the sensor itself to the surface point measurement. A common way to think about this process is that the sensor projects a ray towards the surface point, and all points along that ray are encoded with the signed distance to the surface point. This means that we have both positive values in front of the surface point and negative values behind the surface point. Each sensor ray is assumed to be a one-dimensional case, and the adjacent measurements are ignored.

We now truncate the field to only include a particular set of values between D_{min} and D_{max}, which produces the *projective truncated signed distance field.*

Let $\hat{D}(x)$ denote the approximation to the TSDF based on the projective line-of-sight distances, and let this be added to the n-th estimate of the TSDF, $D_n(x)$ with a corresponding measurement

weight $\hat{W}(x)$. The update rules for the TSDF are thus:

$$D_{n+1}(x) = \frac{D_n(x)W_n(x) + \hat{D}(x)\hat{W}(x)}{W_n(x) + \hat{W}(x)} \tag{3.5a}$$

$$W_{n+1}(x) = min(W_n(x) + \hat{W}(x), W_{max}) \tag{3.5b}$$

D_{n+1} denotes the updated truncated signed distance at x based on the projective estimate $\hat{D}(x)$. The weight $W_n(x)$ is the accumulated sum of the measurement weights $\hat{W}(x)$, up to some limit W_{max}. This measurement weight varies based on the implementation and may take the form of many different types of functions.

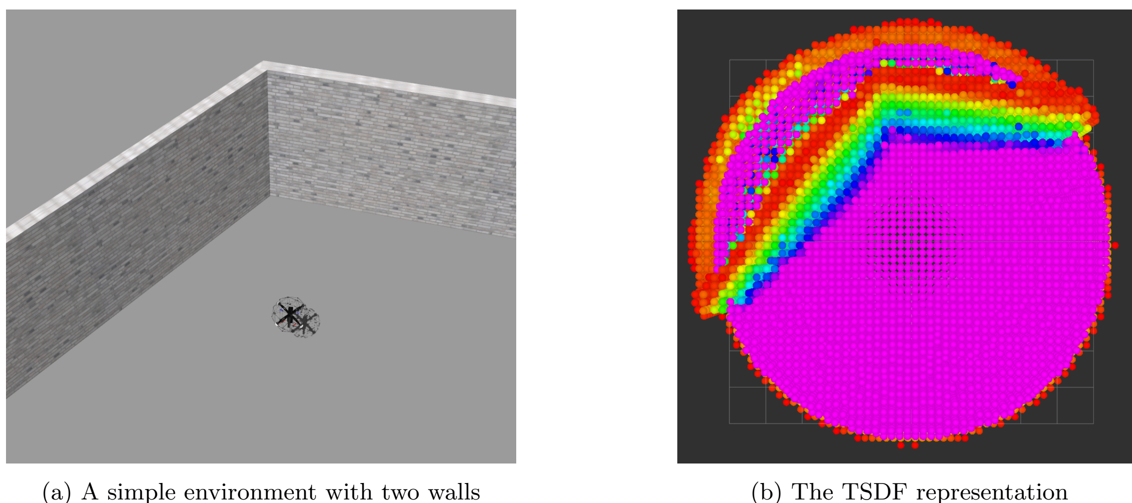
Voxblox [49], which is the library we are using in this master thesis, uses the following weighting function:

$$w_{quad}(x,p) = \begin{cases} \frac{1}{z^2} & -\epsilon < d \\ \frac{1}{z^2}\frac{1}{\delta-\epsilon}(d+\delta) & -\delta < d < -\epsilon \\ 0 & d < -\delta \end{cases} \tag{3.6}$$

where z is the depth of the measurement in the camera frame, the truncation distance $\delta = 4v$, $\epsilon = v$ and v is the voxel size. The intuition behind this weighting function is that the influence of voxels that have not been directly observed is reduced. These voxels may be the ones behind walls or inside objects, and their influence should be minimised as one cannot be sure that they are correct.

When merging a new sensor measurement into the TSDF, Voxblox groups all points in the sensor cloud belonging to the same voxel together, calculates the mean intensity value and distance across the grouped points. Instead of raycasting to every point in the sensor measurement to update the color of all voxels in between, raycasting is done only once to the grouped points to save computation.

From a practical point of view, the TSDF representation offers many advantages. It is fast to construct, which means that it can be run at a high frequency without significant computational penalties. This is crucial to the overall learning-based agent, as we, in general want to reduce the computational complexity and thus be able to run the algorithm with less latency and at a higher frequency. The second benefit of choosing a TSDF representation is that it filters out LiDAR sensor noise when merging new point clouds into the TSDF map. Reducing the noise of the observation input to the compression algorithm is a key benefit. The compression algorithm then does not need to learn how to filter it out explicitly. The third benefit of this representation is that we can make a radius around the robot where the local objects that fall within this radius are integrated. This means that we have a fixed-size observation space for the local vicinity around the robot, which is ideal for doing local motion planning.



(a) A simple environment with two walls

(b) The TSDF representation

Figure 3.7: An example of how the TSDF is made

Figure 3.7 demonstrates what the TSDF looks like in practice. Here we can see empty space encoded in purple, and the solid wall encoded in yellow. We see that most of the free space is not encoded with distance information but rather remains encoded as empty space. Inside the truncation radius from the walls, we can see a gradient of colours protruding from the surface of the walls. This indicates the distance from the wall at any given point inside the truncation radius. At the wall's surface, the signed values cross from positive to negative, which means that the wall is encoded with zeros. Note that in this example, the TSDF is in 3D.

An important consequence of the geometry of the LiDAR, as shown in figure 3.4, is that the space directly above the MAV is *not directly observable*. The TSDF-map does not include anything outside the visible region of the LiDAR, which means that objects passing over and under the MAV will be lost from view. However, because the TSDF *assumes object permanence*, objects that move inside the visible region of space, but behind other obstacles, will be remembered. This means that the TSDF can encode how objects look from the other side as it moves around the object itself. For the navigation of a robot, this means that there is an accumulation of data with which the autoencoder will be able to work with, instead of only being able to see what is directly observable in a given instance.

3.4.5 Compression Methods

As previously stated, the main challenge discussed in this chapter is how to reduce the dimensionality of the incoming LiDAR data stream so that intuition about the surrounding environment can still be inferred from the data. Generally, this dimensionality reduction is either done by selecting the relevant features and discarding the rest (*selection*), or by extracting new features which are created based on the old features (*extraction*).

To discuss dimensionality reduction, we can define some basic concepts. An **encoder** is the process that extracts the new features from the old representation (either by selection or extraction), and a **decoder** is the exact opposite. Thus, the dimensionality reduction process consists of the encoder compression of the data from the initial space to the **encoded space**, or **latent space**. From this compressed representation, the decoder decompresses the data and tries to reconstruct the original

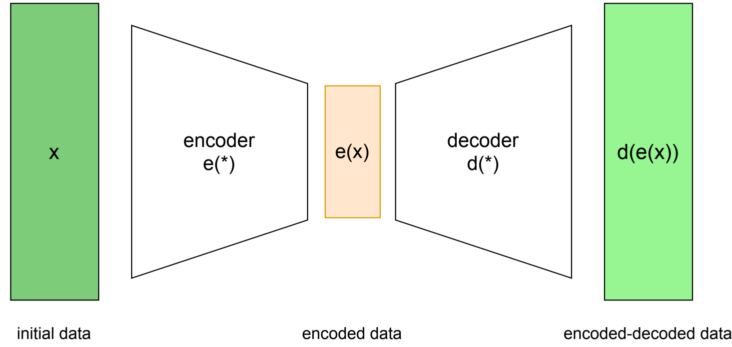


Figure 3.8: Dimensionality reduction using an encoder and decoder

representation. This process can either be lossless or lossy. In a lossless process, no information is lost during compression, and the full representation can be reconstructed later. In a lossy process, some of the information is lost during compression. Depending on the size of the latent space, and the method with which the data is compressed, the amount of lost data varies.

Let $e(*)$ define the encoder function, and $d(*)$ define the decoder function, as illustrated in figure 3.8. We can thus define lossless encoding as:

$$x = d(e(x)) \tag{3.7}$$

and lossy compression as:

$$x \neq d(e(x)) \tag{3.8}$$

In general, we can define the dimensionality reduction problem as finding the best pair of encoder/decoder which **keeps the maximum of information when encoding**, which then results in the minimum **reconstruction error** when decoding. This can then be formulated as:

$$(e^\star, d^\star) = \underset{(e,d)\in E\times D}{\arg\min}\ \epsilon(x, d(e(x))) \tag{3.9}$$

where $\epsilon(x, d(e(x))$ denotes the reconstruction error between the input data x and the compressed and decompressed data $d(e(x))$.

We can see that using the TSDF representation we have already heavily compressed the LiDAR observation space dimensionality significantly while retaining the most essential features.

Autoencoder

The autoencoder [59] is an encoder-decoder structure just like described in the previous section, and uses neural networks, as illustrated in figure 3.9. The encoder is a neural network that compresses the input space into a smaller latent space representation, and the decoder network tries to decompress this latent space into the original representation. This set of networks are trained using an iterative optimisation process, in which the autoencoder is fed a set of data, and the output of the autoencoder is trained to replicate the given data. This is done by measuring the reconstruction error and back-propagating the error through the network structure to update the weights of the networks. The loss

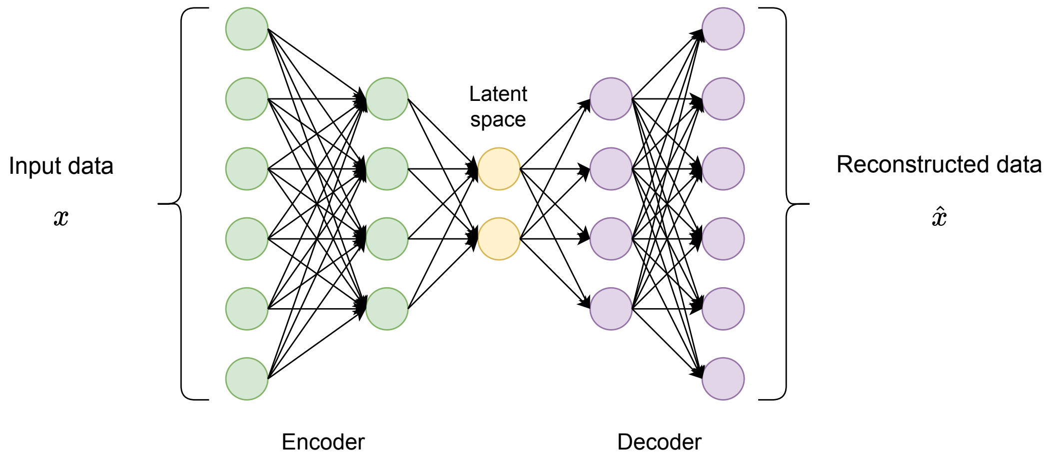


Figure 3.9: Neural network autoencoder

can thus be defined as:

$$loss = ||x - \hat{x}|| = ||x - d(e(x))||^2 \tag{3.10}$$

As seen in the figure 3.9, the autoencoder structure creates a latent space at the "bottleneck", where the dimensionality of the data is reduced. This bottleneck guarantees that the dimensionality of the data is reduced to a certain size through the compression in the encoder.

It is important to note that we do not necessarily need a perfect reconstruction of the input data at the end of the autoencoder. The goal is to reduce the dimension in the bottleneck layer in such a way that the *irrelevant information is discarded and the major structures within the data is preserved*. Depending on the use-case for the autoencoder, the optimal size of the bottleneck layer varies accordingly. For our use case, we want to reduce the size of the LiDAR point cloud data in such a way that the general geometric structures are coarsely represented and that the euclidean distance data is preserved in the bottleneck layer. *We do not need to know all the fine-grained details of an object to be able to do effective collision avoidance.*

3.4.6 Variational Autoencoders

The downside of the standard vanilla autoencoder, is that the latent space may be unstructured and overfitted. When the autoencoder is only trained by minimising the reconstruction error between the input and output, it may take advantage of all overfitting possibilities to achieve its goal. This means in practice that two points that are very close in the latent space might be decoded completely differently. This makes the latent space unpredictable and unorganised, and any network trying to learn anything useful from this latent space representation will have a more difficult job. Ideally, we want two similar input data sets to be represented as points close to each other in the latent space. To do this, Variational Auto Encoders (VAEs) [60] introduce explicit regularisation of the latent space. Intuitively, this creates a "gradient" over the information encoded in the latent space, which means that one can traverse a path in the latent space, and the output of the decoder network will be continually changing.

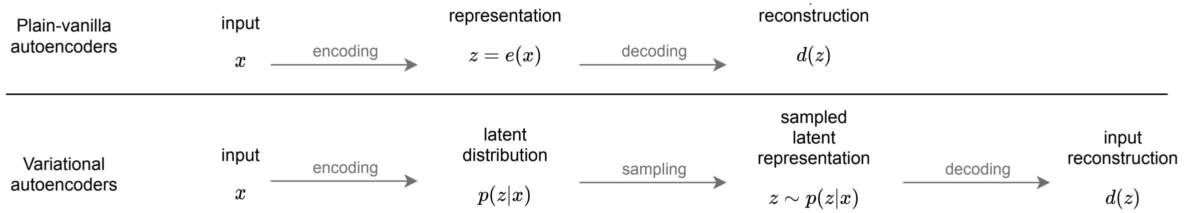


Figure 3.10: Process of the variational autoencoder

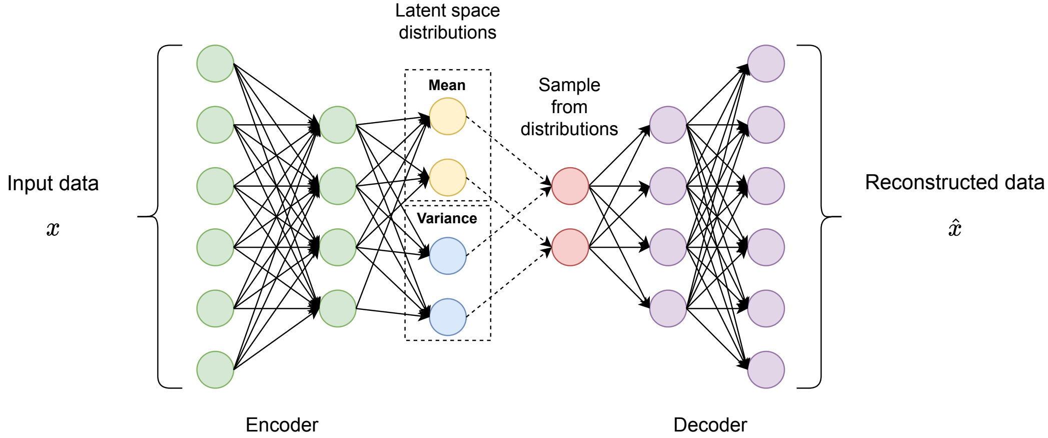


Figure 3.11: Structure of a variational autoencoder

The training process of a VAE is similar to a standard autoencoder. However, in order to introduce regularisation of the latent space, we *encode the input as a distribution over the latent space, rather than as a single point*. This is illustrated in figure 3.10, and the complete VAE structure is illustrated in figure 3.11.

The encoded distributions are chosen to be gaussian distribution, which means that the encoder can be trained to return the mean and covariance matrix that describes this gaussian distribution. This gives a natural latent space regularisation, as the distributions will be forced to resemble a standard normal distribution. The loss function that the training algorithm is minimising when training the VAE is thus comprised of two terms; the first is the reconstruction error like in the plain-vanilla autoencoder, and the second is the regularisation term expressed as the *Kulback-Leibler (KL)* divergence between the returned distributions and a standard Gaussian distribution, as explained in section 2.2.

$$loss = ||x - \hat{x}||^2 + KL[N(\mu_x, \rho_x), N(0,1)] = ||x - d(z)||^2 + KL[N(\mu_x, \rho_x), N(0,1)] \tag{3.11}$$

If the latent space distribution strays too much from the standard normal distribution, this cost will increase and thus increase the loss function. Therefore, during training, the network is therefore incentivised to keep the latent space distribution close to a normal distribution.

Variational Autoencoders From a Statistical Point of View

When we are training a model, we want it to be representative of the dataset that we are feeding it. This means that we want every data point X in our dataset to have at least one latent space representation, which then causes our model to generate an output very similar to the input X. Mathematically speaking, this is represented as a vector of latent variables z which exists in the high-dimensional latent space $\mathcal{Z}$. From this latent space, we can sample according to a given probability density function $P(z)$. Following this latent vector, let us say we have a set of deterministic functions $f(z;\theta)$ parametrised by the parameter vector θ in some parameter space Θ, which maps from the latent space to the input space: $f : \mathcal{Z} \times \Theta \rightarrow \mathcal{X}$. $f(z;\theta)$ is then a random variable in the original input space $\mathcal{X}$, as z is a random sample, and the function f is deterministic with a fixed parameter vector θ. Our goal is to optimise the parameter vector θ in such a way that we can randomly sample z from the distribution $P(z)$ and, with high probability, end up in the original input space $\mathcal{X}$ using our deterministic function $f(z;\theta)$.

Thus, during training, we aim to maximise the probability of each sample z to resemble the training data X:

$$P(X) = \int P(X|z;\theta)P(z)dz \tag{3.12}$$

where $f(z;\theta)$ is replaced by the distribution $P(X|z;\theta)$. The rationale behind this maximum likelihood formulation is that if the model is likely to generate training data samples, it will also be likely to produce similar samples to the training data samples and unlikely to produce dissimilar ones. It is common to use the Gaussian distribution as the choice of output distribution in Variational Autoencoders: $P(X|z;\theta) = \mathcal{N}(X|f(z;\theta), \sigma^2 * I)$, where I is the identity matrix.

By using the Gaussian distribution, we can use common optimisation techniques like gradient descent to increase $P(X)$ by changing the parameter vector θ in such a way that the function $f(z;\theta)$ approaches X for some sampled z.

To make the output of our Variational autoencoder similar to the data samples in the training set, i.e. solving equation 3.12, we have to solve two problems:

1. How do we define the latent variables z, and decide what information they represent?
2. How do we integrate over z, i.e. how do we make sure for all latent space samples z we end up with data samples that are similar to the ones in training set X?

Defining the latent variables in z boils down to choosing what features to look for in our input data. We know we cannot encode the complete input sample into z, and therefore some selection of features from the input data needs to be done. These features may be correlated and dependent on the structure of the latent vector z. What variational autoencoders do, is make sure all samples of z can be drawn from a simple Gaussian distribution $\mathcal{N}(0, I)$ where I is the identity matrix. This is exactly the regularisation we were talking about earlier. It makes sure that the latent space is organised so that the pool of possible latent space vectors is known.

This is possible because any distribution can be generated by taking a set of normally distributed variables and mapping them through a sufficiently complex. The same is then the case for the input; any input distribution may be mapped to a normal distribution using a sufficiently complex function.

Therefore we may use powerful function approximators to map the independent and normally distributed z values to a suitable set of latent variables and then be mapped back to the input data space X.

Reparametrisation Trick

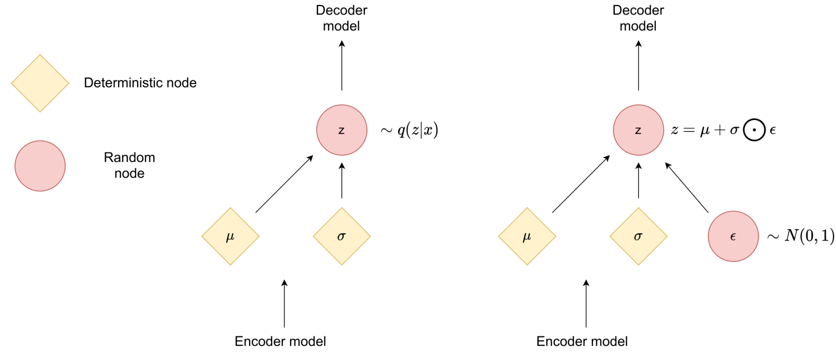


Figure 3.12: Illustration of the reparametrisation trick

A challenge with Variational autoencoders is that the architecture contains a node that does a sampling operation. This sampling node does not permit a backpropagation operation to run through it because of the stochastic nature of the sampling operation. Let us look into why this is the case and what can be done to remedy this.

We look at an example. Let us say that we are trying to calculate the gradient with respect to θ of the following expectation:

$$\mathbb{E}_{p(z)}[f_\theta(z)] \tag{3.13}$$

where p is a probability density function, and θ are the parameters with which the function f_θ is parametrised. If we can differentiate the function $f_\theta(z)$, the gradient is easily computed:

$$\begin{aligned} \nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] =& \nabla_\theta \Big[\int_z p(z) f_\theta(z) dz \Big] \\ =& \int_z p(z) \big[\nabla_\theta f_\theta(z) \big] dz \\ =& \mathbb{E}_{p(z)} \big[\nabla_\theta f_\theta(z) \big] \end{aligned} \tag{3.14}$$

We see that the gradient of the expectation is the same as the expectation of gradient. However, this is assuming an independent probability density function p. In our case, we have that p is also parametrised by θ.

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] =& \nabla_\theta \big[\int_z p_\theta(z) f_\theta(z) dz \big] \\
=& \int_z \nabla_\theta \big[p_\theta(z) f_\theta(z) dz \big] \\
=& \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \int_z p_\theta(z) \nabla_\theta f_\theta(z) dz \\
=& \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \mathbb{E}_{p_\theta(z)} \big[\nabla_\theta f_\theta(z) \big]
\end{aligned}
\tag{3.15}
$$

As we can see, we end up with two terms when calculating the gradient of the expectation of our function f. We do not have any guarantee that the first term is an expectation. If there existed an analytic solution to $\nabla_\theta p_\theta(z)$ this would not be a problem, but since we are working with discrete distributions, we have to use Monte Carlo methods. Monte Carlo methods require that it is possible to sample from $p_\theta(z)$, but not that it is possible to calculate its gradient.

Let us see what happens when we apply the reparametrisation trick to our example.

$$\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) \tag{3.16}$$

$$\boldsymbol{z} = g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \boldsymbol{x}) \tag{3.17}$$

$$\mathbb{E}_{p_\theta(\boldsymbol{z})}[f(\boldsymbol{z}^{(i)})] = \mathbb{E}_{p(\epsilon)}[f(g_\theta(\boldsymbol{\epsilon}, \boldsymbol{x}^{(\boldsymbol{i})}))] \tag{3.18}$$

where vectors are denoted in bold and the ith sample of a vector $\boldsymbol{v}$ is denoted $\boldsymbol{v}^{(i)}$ and $l \in L$ denotes the lth Monte Carlo sample. ϵ is now a random variable drawn from a distribution which does not depend on the parameter vector θ, while our $\boldsymbol{z}$ is now calculated using a function g_θ which depends on both the random ϵ and a parameter x. It is important that the function g_θ is differentiable for the following to work. Let us calculate the gradient of this expectation:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{z})}[f(\boldsymbol{z}^{(i)})] =& \nabla_\theta \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \boldsymbol{x}^{(i)}))] \\
=& \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla_\theta f(g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \boldsymbol{x}^{(i)}))] \\
\approx& \frac{1}{L} \sum_{l=1}^{L} \nabla_\theta f(g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}^{(l)}, \boldsymbol{x}^{(i)}))
\end{aligned}
\tag{3.19}
$$

We have thus used the reparametrisation trick to express a gradient of an expectation as an expectation of a gradient. A monte carlo method can then be used to estimate $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{z})}[f(\boldsymbol{z}^{(i)})]$ using the result of equation 3.19.

Let us put this into our Variational Autoencoder framework. The loss function in equation 3.11 can be re-written as follows:

$$\mathcal{L}_{VAE} = -KL[q_\theta(z|x^{(i)})||p_\theta(z)] + \frac{1}{L} \sum_{l=1}^{L} log p_\theta(x^{(i)}|z^{(l)}) \tag{3.20}$$

$q_\theta(z\|x^{(i)})$	Encoder
$p_\theta(z)$	fixed random distribution
$p_\theta(x^{(i)}\|z^{(l)})$	Decoder

We can interpret the equation 3.20 as follows:

Seen this way, the KL-divergence term will encourage the approximate posterior distribution $q_\theta(z|x^{(i)})$ to resemble the prior fixed random distribution $p_\theta(z)$. If the approximate posterior exactly matches both the real posterior and the prior, we would have that $p(x) = p(x|z)$ using Bayes' rule. Then one would only need to sample a z and condition on z to make a realistic sample x in the same space as the input distribution.

A Pass Through the Computational Graph of the VAE

1.	$\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x = M(\boldsymbol{x}), \Sigma(\boldsymbol{x})$	Pass $\boldsymbol{x}$ through the encoder
2.	$\epsilon \sim \mathcal{N}(0, 1)$	Sample noise
3.	$\boldsymbol{z} = \boldsymbol{\epsilon}\boldsymbol{\sigma}_{\boldsymbol{x}} + \boldsymbol{\mu}_{\boldsymbol{x}}$	Reparameterise
4.	$\boldsymbol{x}_r = p_\theta(\boldsymbol{x}\|\boldsymbol{z})$	Pass $\boldsymbol{z}$ through the decoder
5.	recon. loss = BinaryCrossentropy($(\boldsymbol{x}, \boldsymbol{x}_r)$	Compute the reconstruction loss
6.	var. loss = -KL[$\mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x)\|\|\mathcal{N}(0, I)$]	Compute variational loss
7.	Total loss = recon. loss + var. loss	Combine the losses

3.5 Proposed Method

The proposed method for point cloud feature extraction consists of learning a latent space using a Convolutional Variational Auto-Encoder (CVAE). The input of the CVAE is a binarised version of a local projective TSDF point cloud. The CVAE has then trained to re-project the binarised TSDF point cloud, optimising the feature representation in the latent space. In the larger scope of this project, the learning-based agent will map from this latent space to an acceleration vector. A TSDF representation of the point cloud encodes the distance information between the MAV and the objects within the cloud explicitly. Therefore, it is a valid candidate to ensure collision avoidance properties in the latent space.

3.5.1 Convolutional Variational Autoencoder

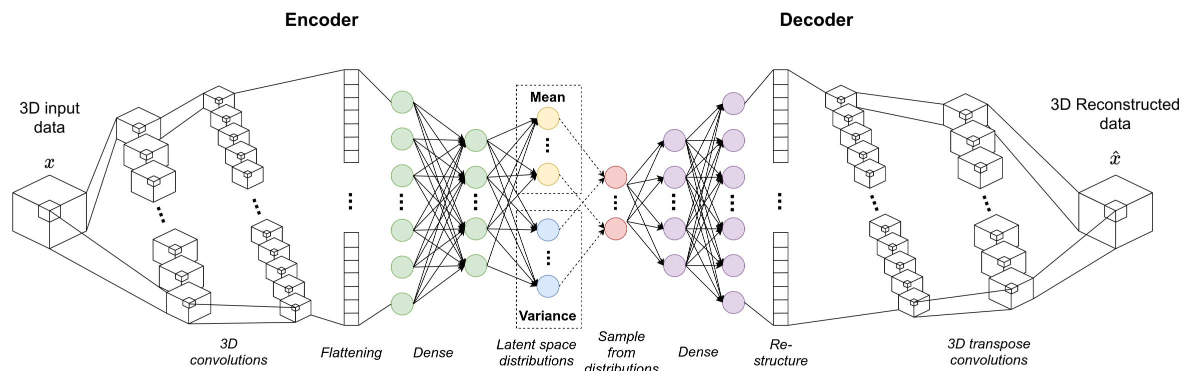


Figure 3.13: General structure of the 3-dimensional Convolutional variational autoencoder

The general structure of the Convolutional Variational Autoencoder can be seen in figure 3.13. Note that the number of convolutional and dense layers are varied, as well as the size of the *bottleneck*, or namely the *latent space dimension*.

The number of convolutional layers, as well as the number of learned filters in each layer, determine how fine-grained the features we are extracting are. We can generally regard this part of the network as a *feature extractor*, where the first convolutional layer recognises low-level features like edges, and the following convolutional layers pieces together higher-order features made up of the low-level features.

After flattening the feature map from the convolutional layers, we get a *feature vector* from which the following dense layers will learn from. The job of the dense layers is to recognise what combinations of features in the feature vector corresponds to a given type of object in the input data and compress this into a *latent representation* which retains only the most necessary information.

In the *bottleneck layer*, we have the latent space distributions. This is where we extract the latent space vector, which we will send to the RL agent. We expect that by increasing the size of this layer, more information can get through to the decoder. This will hopefully make the reconstructions better and the latent space more descriptive.

It is important to note that more and denser layers will not necessarily make the encoder any better. If we increase the complexity too much, we may end up with an *overparametrised* network. This boils down to the network being much more complex than the data it is trying to encode and will make the network very prone to overfitting. We need to make sure that the complexity of the CVAE matches the feature-granularity in our input data, or else we may end up in a situation where input samples that lie close to each other in the input space are treated completely different by the CVAE. As explained in section 3.4.6, introducing an explicit regularisation of the latent space helps with the overfitting problem, but it gives no guarantee.

Internal activation function	ReLU
Output activation function	Sigmoid
Kernel size	3
Strides	2
Padding	Same
Loss function	Weighted Binary Crossentropy
Loss function weighting	60/40
Learning rate	0.001
Optimizer	Adam

Table 3.1: The parameters of the Convolutional Variational Autoencoder

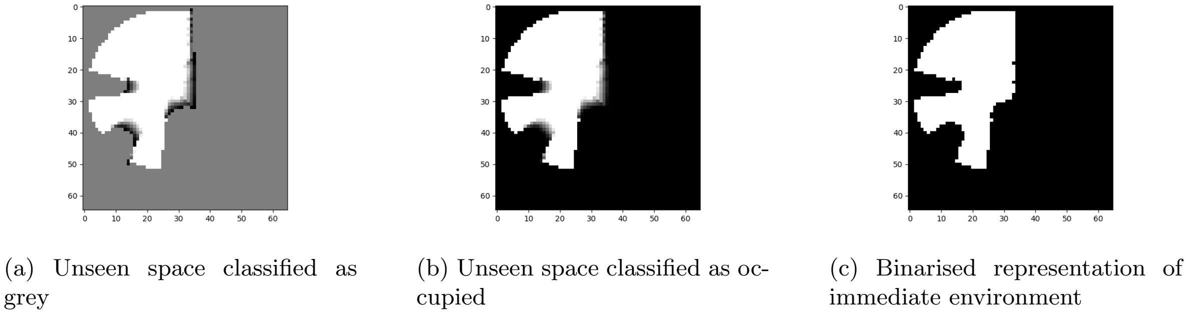
It is worth noting that the chosen loss function for the autoencoder, *Weighted Binary Crossentropy*, as seen in table 3.1, plays an important role. For an agent traversing the environment, the most critical factor is to ensure collision avoidance. If we look at this from a statistics standpoint, Type-II faults are much worse than Type-I faults. This is because a model that cannot reproduce an object, may result in a collision if the agent thinks the space is empty. However, a false positive will make the agent avoid a region of empty space, and may in the worst case make the agent unsure as to where to go because of clutter in its reconstructed map. The latter consequence can mean a deterioration in performance, while the former consequence is potentially disastrous. During the training of the autoencoder, this is compensated for by the fact that a false positive (a positive in this sense meaning an obstacle) is penalised less than a false negative.

1. **Insert point cloud from LiDAR into local TSDF representation using Voxblox**

   The first step in the dimensionality reduction process already starts here. For scanning the environment an Ouster OS1 LiDAR system with 64 beams is used [61]. As we can see in the technical specifications in table A.1, we receive 1.3 million points per second. This is a data stream of very large dimensions. To reduce this dimensionality while preserving necessary features, these point clouds are fed to the TSDF server of the Voxblox algorithm [31]. The TSDF representation is constructed using the method explained in section 3.4.4, and as each new point cloud is received from the LiDAR, it is integrated into the existing TSDF point cloud. It is important to note that because this process integrates point clouds into an already existing representation, there is a possibility of erroneous detections being propagated through several iterations of the TSDF representations before it is corrected. This is especially true if an erroneous detection disappears from the direct view of the LiDAR, as the TSDF server will assume object permanence and therefore preserve the detection.

   The most interesting property of the local TSDF representation is the fact that it has a fixed size, in our case 64x64x16. This is hugely important as the input of our convolutional neural networks expects a constant input dimension size. The local TSDF will thus in practical terms, be a sliding window of the immediate local vicinity of the MAV, with the MAV position being in the middle of the frame. Since we are doing local path planning and collision avoidance, knowing the immediate vicinity should be sufficient for generating a local collision-free path.

2. **Convert the TSDF point cloud into a binarised 4D-tensor**



(a) Unseen space classified as grey

(b) Unseen space classified as occupied

(c) Binarised representation of immediate environment

Figure 3.14: Different configurations of the input TSDF point cloud tensor shown in 2D

Now that we have a rich TSDF point cloud representation around the MAV, we will binarise it. We are only interested in knowing if a pixel is empty or occupied, and thus binarising the TSDF representation will enable us to generate a 3D occupancy map like the one illustrated in section 3.4.3. However, we are considering unexplored space as occupied. This is because a TSDF detection of an object inherently does not know what is inside an object, and the result is that some detections are only represented as thin edges as seen in figure 3.14a. This is a problem when training the autoencoder, as it will have a hard time learning thin objects as they constitute a small fraction of the overall pixels in the input tensor. Ensuring that unexplored space is considered occupied solves this problem and makes the autoencoder much better at reconstructing a useful 3D map of the environment. It is worth noting that this means that simple geometric objects initially are not represented by their geometry until the agent has had the possibility of seeing this object from different angles.

We then binarise the tensor with a binarisation threshold of 0.1. This means that all space that is occupied, and all space closer to 10cm to a wall is considered occupied, while the rest is considered free space. Including 10cm of the area protruding from the surface of obstacles adds a *padding* which does two things. Firstly it makes sure that the volume of the detected objects is increased slightly to allow for easier learning for the autoencoder. Secondly, it adds a safety margin for the agent making it less likely to collide during training.

Binarising the TSDF representation also has a very important practical advantage, as we are able to encode each pixel with an 8-bit integer value instead of the 64-bit floating-point value returned by Voxblox. The resulting reduction in the size of each set of TSDF point clouds is by a factor of 8, which in turn significantly increases the size of the dataset available for training later on.

3. **Encode this 4D-tensor into a latent space**

   Now we pass the binarised tensor into the autoencoder. When doing inference, it is important to note that we do not add noise as described in section 3.4.6; we only pass the mean value through to the decoder. If we had not done this, we would have an output that would be varying for a constant TSDF point cloud input. We both extract the reconstructed tensor and the latent space from this process. The reconstructed tensor gives us a clue about how well the input image features have been encoded. However, this is *not a direct proof* of what is encoded into the

latent space. Suppose a TSDF point cloud is fed into the autoencoder which differs significantly from anything it has seen before. In that case, we cannot be absolutely certain about what gets encoded into the latent space. The decoder will try to reconstruct the features it sees in the latent space, but this assumes that the decoder will be able to recognise the features in the latent space.Looking at the reconstruction can serve as a good *indication* about what gets encoded into the latent space.

4. **Feed this latent space representation to the agent**

   Once the encoder has generated this latent space representation, it is then fed to the learning-based agent, which will learn to recognise features from this latent space to generate collision-free paths.

3.5.2 Computational Graph

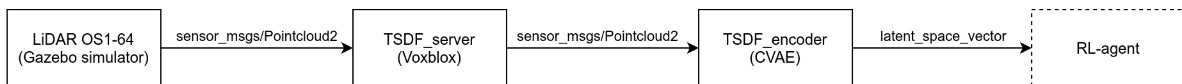


Figure 3.15: The computational graph for creating the latent space vector

The computational graph of the CVAE pipeline is illustrated in figure 3.15.

3.6 Results

3.6.1 Training Methodology

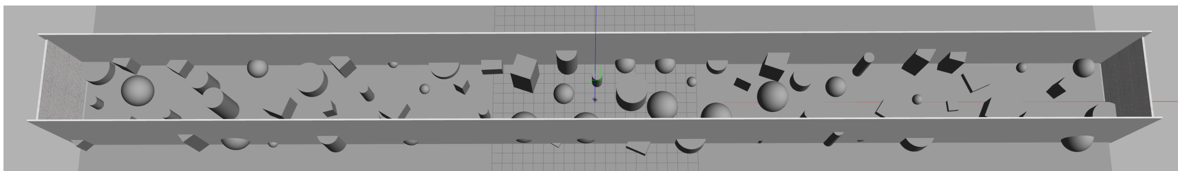
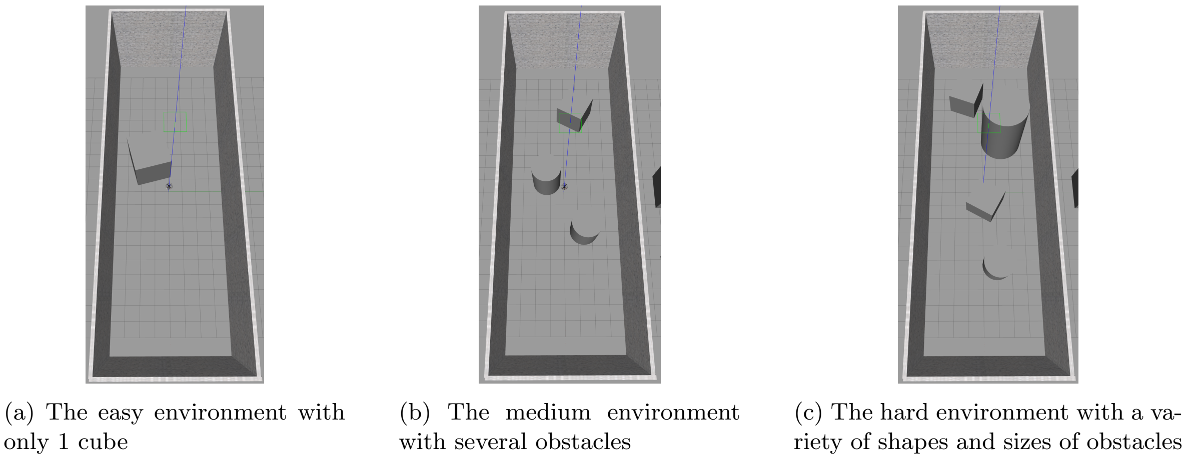


Figure 3.16: The generalised training environment



(a) The easy environment with only 1 cube

(b) The medium environment with several obstacles

(c) The hard environment with a variety of shapes and sizes of obstacles

Figure 3.17: The training environments which get progressively more difficult

The training procedure in chapter 4 uses the environments shown in figure 3.17. The first environment 3.17a is a room in which there exists only one large cube as an obstacle to the MAV. In the second environment 3.17b we introduce more types of objects with different geometries. In the third environment 3.17c, the number and complexity of obstacles are increased even further. The obstacles in these environments are shuffled at a certain interval to ensure generalisability. This serves as an excellent way of testing the performance of a generalised autoencoder which is trained on a varied data set from the environment in figure 3.16. We can test how much deterioration in reconstruction we get as the environment we are testing in gets increasingly more difficult. This will give us an important clue as to how well the autoencoder can generalise between different complexities in the environments in which it is operating.

The training data for the generalised autoencoder is collected in a randomised environment, exemplified in figure 3.16. This is a 100m long corridor where obstacles are randomly placed. The MAV is traversing the environment running the Dagger algorithm, as explained in chapter 4. For every 10 runs, the obstacles in the environment are shuffled randomly inside the corridor. The walls of the corridor always stay the same. The goal is then to have an encoder that generalises to unseen configurations of this obstacle corridor. The shuffling should help prevent overfitting as the model will receive an input dataset that is well distributed. We collect 100,000 TSDF point clouds from the training environment for training, and 20,000 TSDF point clouds for validation. Other types of environments like caves and urban tunnels were also investigated. However, to be able to assess the performance of the CVAE precisely, an environment like the one exemplified in figure 3.16 gives a better indication as to what types of objects the CVAE is able to understand or not.

We want to compress the input dimensions as much as possible. Therefore it is necessary to test the limits to how accurate the CVAE can reconstruct point clouds with different dimensions of the latent space, number and width of dense layers, and the number of filters and number of convolutional layers. To know approximately where to start, we look to the paper [50], where they use a similar approach of constructing a 40x40x40 TSDF which is then passed to a 3D CNN network. We note that they construct a TSDF with a size that is in the same order of magnitude as ours at 64x64x16, and that they use three convolutional layers in their encoder with the number of filters $[16, 32, 64]$. We use this as a suggestion as to where to start in the parameter search.

We are trying to capture the correct granularity of details from the dataset with the complexity of our autoencoder model. This means that it is not interesting for us to capture a detailed representation of the surfaces of objects, but only the coarse general geometric structure of the immediate environment. This should be sufficient for doing collision avoidance. However, this needs to be tested to find the right balance. We will therefore test the efficacy of a chosen autoencoder model in the chapter 4.

3.6.2 Metric

To directly compare the different network configurations, we will look at the loss of the training. As explained earlier, this includes both the KL-divergence from a standard normal distribution and the reconstruction loss. This means that we will get an idea about how well the latent space is regularised and how good the network is at reconstructing the input data. It is important to have a validation dataset with which the autoencoder may be continually validated during training. This will indicate which point the model is starting to overfit to the input data, as the training loss and the validation loss will start to diverge.

Each model will be evaluated at the point at which the validation loss and the training loss start to diverge.

We will use the F1-metric evaluate each architecture's performance. This is the geometric mean between *precision*, as seen in equation 3.21, and *recall*, as seen in equation 3.22.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{3.21}$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{3.22}$$

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.23}$$

3.6.3 Performance of Different Network Configurations

Model architecture nr.	1	2	3	4	5	6	7	8	9	10	11	12
Total number of parameters	1180037	1601701	1902757	867333	952055	1218537	1640201	1979657	3030345	4447208	1646697	990825
Latent space dimension	50	50	50	50	50	100	100	100	100	100	100	100
Dense layer structure	[512,256]	[512,256]	[512,512]	[256,128]	[256,256]	[512,256]	[512,256]	[512,512]	[512,512]	[512,256]	[512,256]	[512,256]
Convolutional layer structure	[8,16,32]	[16,32,64]	[16,32,64]	[8,16,32,64]	[8,16,32,64]	[8,16,32]	[16,32,64]	[16,32,64]	[32,64,128]	[16,32,64]	[8,16,32,64]	[8,16,32,64]
Maxpooling	Y	Y	Y	N	N	Y	Y	Y	Y	N	N	N
Validation loss	236.6	283.6	259.6	221.4	235.5	237.2	272.8	247.4	294.5	217	240	240.6
Validation KL-loss	25.81	26.34	26.17	25.21	23.14	27.76	25.03	26.05	23.71	30.67	21.55	23.54
Validation reconstruction loss	210.8	257.3	233.4	196.2	212.4	209.4	247.2	221.4	276.1	186.3	218.5	219.7
Validation precision	0.95	0.923	0.9379	0.9512	0.9421	0.9453	0.9331	0.9458	0.9129	0.952	0.9414	0.9466
Validation recall	0.9605	0.9665	0.9634	0.9647	0.9684	0.969	0.9583	0.9626	0.9571	0.9703	0.9685	0.9623
Validation F1-score	0.9552	0.9443	0.9504	0.9579	0.9551	0.957	0.9455	0.9541	0.9384	0.9611	0.9547	0.9544

Table 3.2: CVAE parameter tuning

3.6.4 Effect of Compression on the Data Stream

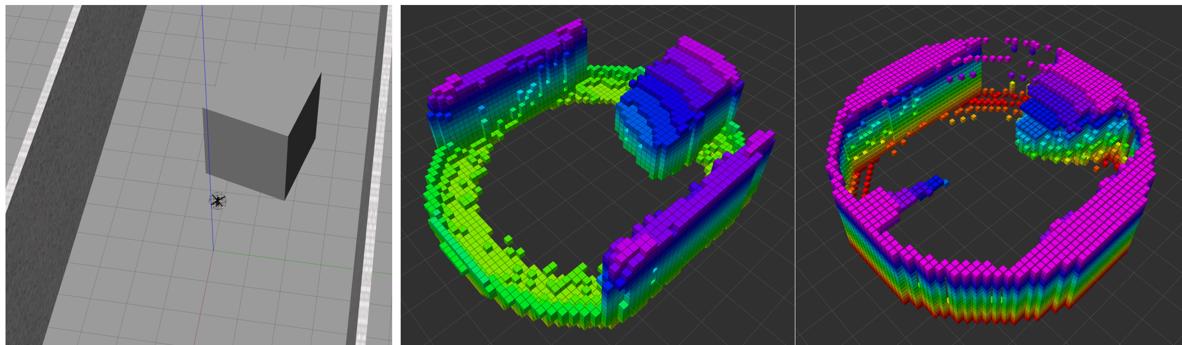
We present the bandwidth of the different components of the CVAE pipeline.

Data stream	Bandwith @ 10 Hz
Raw LiDAR point clouds	$\sim$ 4.09 MB/s
TSDF point clouds	$\sim$ 5.54 MB/s
Latent vector	$\sim$ 2.07 KB/s

Table 3.3: Bandwidth of the different data streams in the CVAE pipeline

3.6.5 Generalisability of Networks

The following illustrations show the reconstruction performance of the autoencoder in increasingly more complex environments.

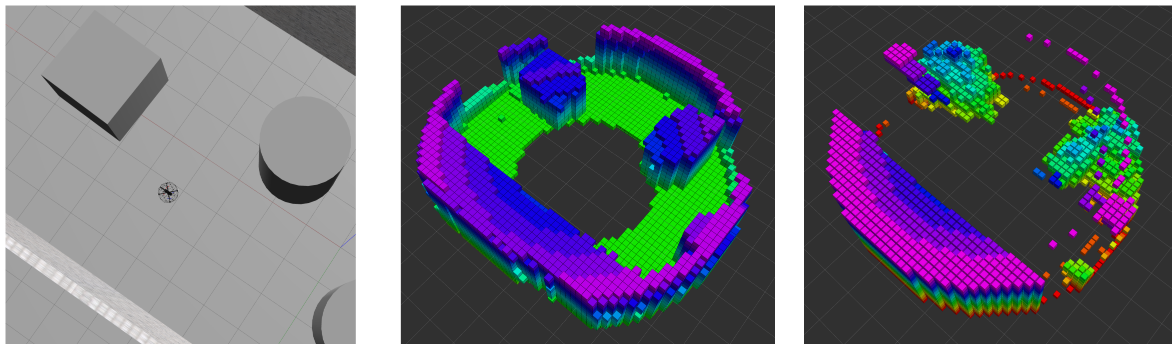


(a) The MAV in Gazebo simulation

(b) The occupancy map generated by the TSDF server

(c) The reconstructed output TSDF point cloud from the CVAE

Figure 3.18: Illustration of the performance of the CVAE in the easy environment 3.17a

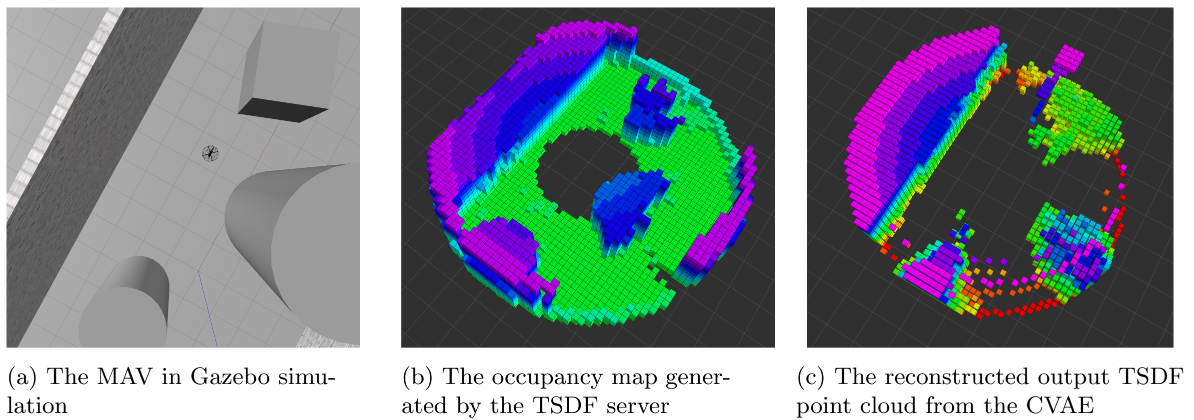


(a) The MAV in Gazebo simulation

(b) The occupancy map generated by the TSDF server

(c) The reconstructed output TSDF point cloud from the CVAE

Figure 3.19: Illustration of the performance of the CVAE in the medium environment 3.17b

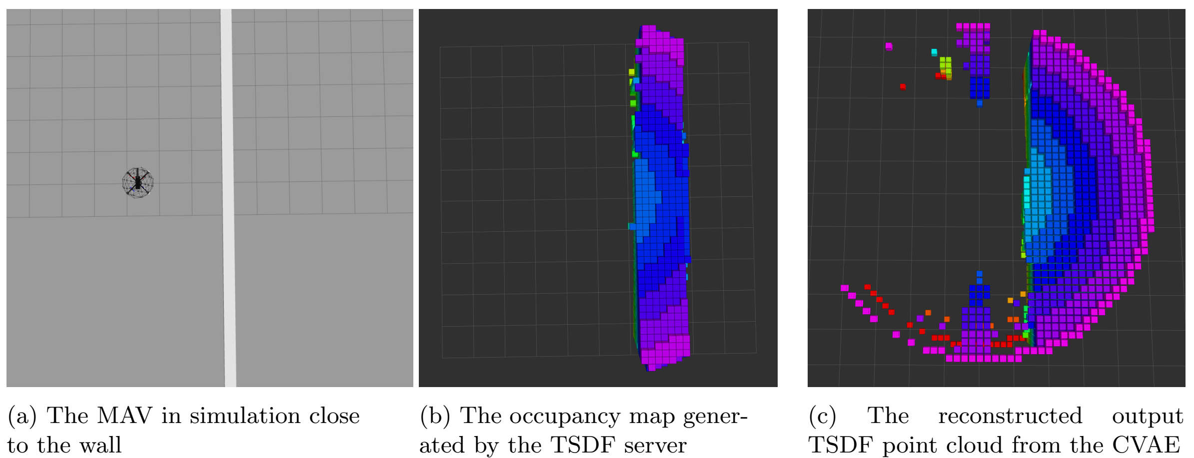


(a) The MAV in Gazebo simulation

(b) The occupancy map generated by the TSDF server

(c) The reconstructed output TSDF point cloud from the CVAE

Figure 3.20: Illustration of the performance of the CVAE in the hard environment 3.17c

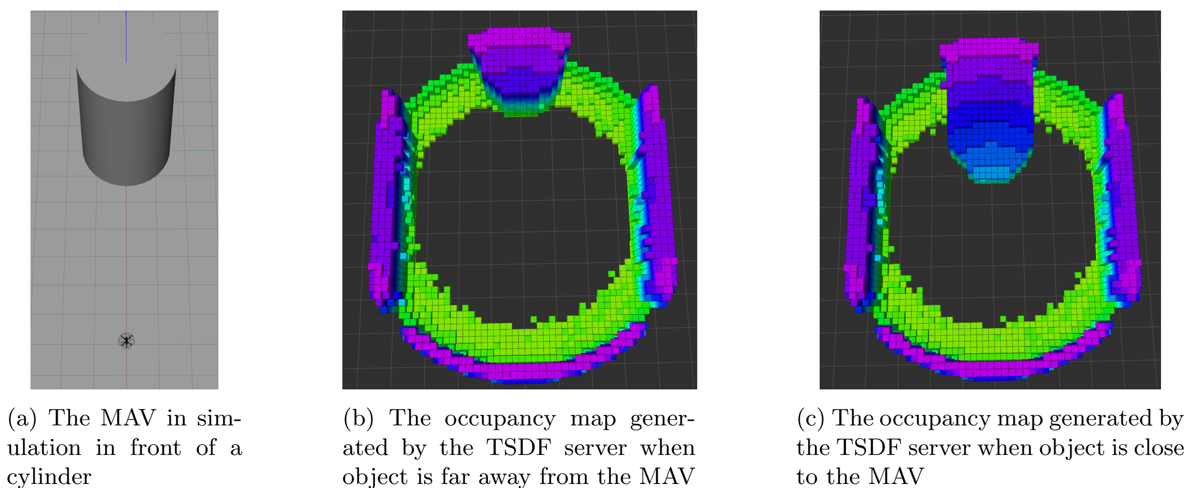


(a) The MAV in simulation close to the wall

(b) The occupancy map generated by the TSDF server

(c) The reconstructed output TSDF point cloud from the CVAE

Figure 3.21: Generalisation of a wall



(a) The MAV in simulation in front of a cylinder

(b) The occupancy map generated by the TSDF server when object is far away from the MAV

(c) The occupancy map generated by the TSDF server when object is close to the MAV

Figure 3.22: The occupancy map created when the MAV is approaching a cylinder head-on

3.7 Discussion

3.7.1 Model Architecture

We begin by looking at table 3.2 where the results from exhaustive experimentation with CVAE architecture parameters are presented. They centre around 50 and 100 as latent space dimensions, as these have been found empirically through early trials to result in the best reconstructions for the given problem. We remind the reader that this chapter aims to compress the input TSDF point cloud as much as possible while retaining useful features. A smaller latent space means that a learning-based agent will have an easier job of learning the patterns present, as they necessarily will be of lower dimensions and be more general. Therefore we do not only look for the overall *best* reconstruction but also a prudent balance between latent space dimensionality, network complexity and reconstruction performance.

It is evident that the model performance is a balancing act between the three main components of the CVAE architecture; the structure and depth in the convolutional layers, the structure and depth in the dense layers, and the latent space dimensions.

The first part of the network consists of the convolutional layers and eventually a max-pooling layer. As mentioned previously, this part of the network extracts features from the input TSDF point cloud. The number of filters at each layer determines the complexity of features extracted at each feature map, and reduces the feature map dimensionality. In table 3.2 we can see that we alternate between having three convolutional layers and one pooling layer, four convolutional layers, and only three convolutional layers. Observe the difference in the total number of parameters between model nr. 10 and 7. The only difference between these two architectures is the inclusion of a max-pooling layer that decreases the feature map's dimensionality before this is passed on to the fully-connected part of the network. This means that the additional 2.8M network parameters can only be attributed to the interface between the convolutional layers and the dense layers in the CVAE. Consequently, we do see a benefit of these extra parameter weights in the F1-score between these two architectures at the expense of the massive increase in computational cost. Instead of a max-pooling layer, model nr. 11 uses an additional convolutional layer to reduce the dimensionality of the last feature map before the dense layers. Comparing nr. 7 and 10, we see a great advantage of using an extra convolutional layer instead of a max-pooling layer, as the F1-score of the model nr 11 is now much closer to the F1-score of model nr 10. This is also because the model nr. 11 has roughly the same number parameter weights as model nr. 7, and a great number fewer than nr. 10.

An interesting comparison can be made between the model nr. 8 and 9. Here we see the effect of *overparametrisation* in the convolutional layers, as the simpler convolutional architecture of model nr. 8 results in a substantially better F1-score with everything else being equal.

As explained in section 2.1.2, we know that a more complex fully connected network architecture results in a more fine-grained learning of feature in the training data. We also remember that complex networks are prone to *overparametrise* and get stuck at local minima. Therefore, it is very interesting to observe the difference between Model nr. 4 and 5 in table 3.2, as we can see that they are identical except for the depth in their dense layers. The model nr. 4 as a slightly simpler dense layer architecture than model nr. 5, but despite this its F1-score is actually better. This may suggest that the granularity of features presented by the Convolutional layers fits better with the simpler architecture of model nr. 4, and that model nr. 5 may overparametrise slightly. The opposite is the case between the model nr.

2 and 3. Here we see an improved F1-score by increasing the depth and complexity of the dense layers, which suggests that the features collected by the convolutional layers fit better with this network.

To reach a good balance between the complexities in these different model architectures, we look at the tradeoff between computational complexity, latent space dimensions and model performance. Model nr. 4 seems to be the best tradeoff between these parameters. At only 867k total number of parameters and a latent space dimension of 50 it has an impressive F1-score, and is only outperformed by the model nr. 10, which has a latent space of 100 and 4.4M number of network parameters. Therefore, this is the model architecture we will use for the remainder of this thesis.

However, it is important to mention that these results are only applicable to the types of environments and objects presented in figure 3.16 and figures 3.17a-3.17c which are limited to straight walls, cubes, cylinders and spheres of different dimensions. We have found the network parameters suitable for extracting features of this granularity from the environment. It is to be expected that should the MAV explore much more complex environments; the CVAE will try to represent these environments using the types of figures it has seen in this dataset which may not be sufficient at all.

Lastly, we need to discuss the fact that the F1-score is an aggregate measure that tells us about the reconstruction performance of the CVAE across the whole input-output point cloud tuple. No information about the performance with respect to the locality of the objects, or their size, is included. The first problem with this is that we do not know whether all sectors of the input TSDF point cloud are treated equally during training. If there exists a bias in the training data for the locality of objects, this will not be accounted for. The result may be that the latent space is unevenly distributed with respect to how an object moves around in the local TSDF point cloud. The second problem is the problem of object size. In the current CVAE setup, we are treating each input-output voxel tuple independently in the cost function. This means that objects that are larger in size are statistically more likely to be reconstructed than smaller objects solely due to their increased number of voxels. The current F1-score treats each input-output voxel tuple independently as well and is thus not able to capture the imbalance with respect to the coupling between the likelihood of reconstruction and the size of objects. Ideally, we should have incorporated into the loss function a penalty for not reconstructing an object regardless of its size. The F1-score should also have been adjusted measure the geometric average between precision and recall not from independent input-output voxel tuples but rather from the precision and recall of entire objects in the input-output point cloud tuple. However, this would require extensive labelling of the training data.

3.7.2 Visual Evaluation of Model Performance

Before proceeding with the visual inspection of the autoencoder performance, it must be mentioned how these visualisations are created and what exactly they represent. As all unknown space is considered occupied, and the input to the autoencoder itself has the dimensions of a rectangular cuboid, if we were to look directly at the input and output TSDF point clouds, we would see nothing but a rectangular cuboid. To overcome this, a great deal of points have been filtered out. All points outside the visibility radius of the local representation have been filtered away, and so have the points corresponding to the unobservable cone-shaped regions directly above and below the agent. These points will always be considered occupied by both input and output and thus have no significance for our analysis.

Let us begin by looking at the figure 3.18. Here, the MAV is placed directly in front of a cube with walls on either side. We can see how this gets encoded into the occupancy map from the TSDF server

in figure 3.18b, with only some geometry preserved. The first thing to notice is how the top of the box seems to be increasing towards the edge of the TSDF point cloud. This is due to the geometry of how the LiDAR scanning operates, as shown in figure 3.4. The second important thing to notice is that the sides of the cube are not at 90-degree angles to the side facing the MAV. This is to be expected, as the left side of the cube has not been observed as the MAV has been approaching the cube straight-on. For the right side of the cube to be observed, the MAV needs to be a certain distance away from the cube, and at this distance, the cube is outside the visible region of the TSDF occupancy map. If we look at the reconstructed TSDF point cloud in figure 3.18c, we see that the cube is now a more pointy object. Some loss of geometry is to be expected, as the CVAE is performing a lossy compression. Also, notice how there exists two artefacts at either end of the reconstructed TSDF point cloud that is present in all figures 3.18c-3.21c. Somehow the model has learned during training that having these two artefacts present will improve either the reconstruction loss or the KL-divergence loss.

We continue by looking at the figure 3.19. The wall is perfectly represented in the reconstruction in figure 3.19c, while the two obstacles have not been decoded in such a way that preserves their geometry. We can see that they are represented as two blobs with no discernible straight sides as is the case in the input data. However, their locations in the body frame seem to line up nicely with the input TSDF point cloud in figure 3.19b.

Moving on to the more challenging environment in figure 3.20, we have three obstacles placed around the MAV. In this case, we can see that the autoencoder struggles a lot more, as shown in figure 3.20c. We can clearly see that it understand that there exist three separate objects, however, it is not able to accurately reconstruct either their geometry or their position. The cylinder to the bottom right of the TSDF point cloud is placed way too far off-centre, which could result in the agent misunderstanding where there is a safe path.

An interesting property of this autoencoder is its ability to generalise across many input samples. Let us look at the figure 3.21. The MAV is placed close to the wall, but due to how it has been spawned, the wall is not represented as a solid mass all out to the outer rim of the TSDF point cloud radius. Even though the input to the autoencoder is only a strip, the autoencoder can recognise this as a wall and generalises this to the other examples of walls it has seen before, as can be seen in figure 3.21c.

In the case of approaching an object head-on, the object will not have been seen from different angles. Because we have defined all unseen space as occupied space, we generate a TSDF point cloud as shown in figure 3.22. In figure 3.22b we see that the general shape of the cylinder is preserved, but as we get closer to the cylinder, we see that the shape stretches out, as seen in figure 3.22c. Since this cylinder has not been seen from several different angles, the TSDF point cloud assumes that the space behind it is occupied, which results in the shape of the cylinder not being preserved. It is important to keep this behaviour in mind as we proceed to the navigation problem in the following chapter.

In this section we have tried to make a generalised autoencoder using many different types of objects with different dimensions. The results in this section show that the autoencoder is not able to accurately reconstruct this wide array of different types of objects and geometries that are present in the training environment 3.16. Therefore, it is evident that more work needs to be done to this autoencoder to generalise it, before it can perform in a wide range of different environments.

3.7.3 Implementation Evaluation

In addition to evaluating the autoencoders performance, we also have to consider the whole pipeline in which it is implemented. We know that the best performance a learning-based model can deliver is strictly bounded by the quality of the data it is fed. This is very much the case for this autoencoder as well, and we have to discuss the performance of the algorithms and physical considerations of the perception system, which comes before the autoencoder in the computational pipeline.

The first important aspect to consider is the physical considerations of the perception system itself. As previously mentioned, the geometry of the LiDAR ray casting means that we have both observable and unobservable regions in the space around the robot, as seen in figure 3.4. This means that the perception system can give a rich representation of an environment on the plane in which it sits, but it is very limited in its ability to judge obstacles' heights. Another consequence of this fact is that the path that the MAV can take while retaining observability in the direction it is going is necessarily also limited. If the MAV decides to ascend directly, it will move into unobservable space. Increasing the angle of the LiDAR raycasting will significantly improve this.

To mitigate the shortcomings of a LiDAR system such as the one we are using here, it is normal to integrate all incoming TSDF point clouds into either a local map or a global map. In our implementation, we are only working with a local map of fixed size, in which the MAV is situated directly in the middle. The radius of this local map representation has been chosen so that it is small enough to have an accurate representation of the immediate surroundings and at the same time being large enough to let the agent plan further ahead than just a couple of seconds. But constructing this local map is not trivial. In this implementation, we are using the local TSDF mapping from the Voxblox [31] software package. While this implementation is fast and has some interesting properties, it is also not perfectly suited for this use case. At its core, this TSDF generation process is designed to be used for integrating local TSDF point clouds into a global ESDF map of the environment. This means that some of the TSDF point cloud integration, which would have been very useful for our use case, has been left for the generation of the global ESDF map. One example of this is the way the TSDF map handles object permanence. Since we assume a static map, we can integrate the TSDF point clouds taken at different angles of an object to ascertain its geometry more precisely. This works well for objects that move inside the visible region of the LiDAR. In figure 3.19b we see that the TSDF mapping has seen both the box and the cylinder from many angles, and they are thus represented as free-standing objects. However, this is not the case for objects moving from inside the visible region to outside.

Let us examine figure 3.18b. As the MAV has been approaching this cube, it has clearly noticed its true height. However, all parts of the cube that exits the visible region are erased from memory. We observe this in the figure with an object increasing in height towards the edges of the local map. The same is true for objects that move from the visible region to underneath it. The TSDF saves no information about its position and geometry. This means that even though we are using an integrating method for generating the TSDF point cloud, the area directly above and below the MAV is unobservable. Any velocity vector of the MAV that points outside the visible region will be executed without considering the immediate environment. Therefore, it is apparent that the input to the autoencoder should use an integrating approach that assumes object permanence, not only in the visible region, but also in the non-visible region. This way, the shortcomings of the LiDAR perception systems could be mitigated. Another possible solution to this problem is to use a sliding window of past observations,

and pass those on to the agent to have a temporal perspective over the objects positions.

It is also worth mentioning that the TSDF generation process assumes a low yaw rotational velocity. The local TSDF map reacts slowly to changes in yaw because of the way it integrates the incoming TSDF point clouds. If a sudden and quick rotation is induced to the MAV, i.e. because of a collision, the resulting TSDF mapping is very noisy. It then takes a considerable amount of time to clear erroneous voxels. Practically, this has two implications. The first implication is that we should fly the MAV yawless. Since the only way of maintaining a constant yaw angle inside a confined environment is through the perception of the surrounding environment, a sensor fusion framework that can detect rotations in the yaw axis of the MAV accurately is needed. The second is that the local TSDF representation is unstable for collision-prone MAVs. If a MAV is to frequently collide with walls, it would need some time to stabilise its local TSDF map before proceeding with confidence. This adds a cost to the flying time of the MAV.

Let us look at table 3.3 presenting the bandwidth of the data streams between the different components of the CVAE pipeline. It might come as a surprise to the reader that the bandwidth actually increases after the raw LiDAR point clouds have been integrated into the TSDF representation. There are two fundamental reasons for this. The raw point cloud stream coming from the LiDAR is a *sparse* representation of the environment. This means that only the points corresponding to an object are included. However, in the TSDF point cloud all points in the 3D space within a given radius are included. This makes this representation a *dense* representation. The second reason for the increase in bandwidth is that the TSDF encodes a truncated signed distance into each point in 3D space, whereas in the raw point cloud from the LiDAR all points have the same underlying value corresponding to a surface. Furthermore, we can observe that the bandwidth of the data stream significantly reduces as the TSDF point clouds are compressed into a latent vector. The resulting latent vector data stream reduces the bandwidth by three orders of magnitude. This result is immensely useful for applications outside of the scope of this thesis as well, as any situation in which bandwidth is severely limited could benefit from such an intelligent compression.

3.8 Conclusion

This chapter has looked at a novel and efficient way of compressing a 3D voxel-based Truncated Signed Distance map into a latent space representation using a Convolutional Variational Autoencoder. An exhaustive search of model architectures has illustrated how to balance the different subcomponents of the CVAE to optimise reconstruction performance while keeping the computational cost low. Both the training and evaluation of this compression algorithm was conducted based on single corresponding input-output voxel tuples and is thus not treating the 3D environment in a topological manner where similar voxels are clustered together. This makes the algorithm more sensitive to objects which contains a large number of voxels given the statistical nature of the reconstruction on a voxel-to-voxel basis. To overcome this, labelling of the input dataset into discrete objects should be implemented in the training process to ensure maximum recall of objects regardless of their mass.

We showed that the algorithm is capable of compressing the bandwidth of the point cloud data stream by three orders of magnitude while being able to reconstruct the input point cloud coarsely. The main limitation of this compression method is its limited ability to generalise to unseen environment topologies.

To make this approach more practically feasible, the LiDAR should have a higher field of view to ensure that a higher amount of the immediate environment is directly observable to the MAV. Since the field-of-view may never be a 100% because of the presence of the MAV hardware, a more intelligent way of integrating new point cloud measurements into the local Truncated Signed Distance Field representation should be employed. This could ensure that objects that leaves the directly observable region are remembered. This would increase the size of the action space of the MAV significantly.

Chapter 4

Reinforcement Learning for collision avoidance

4.1 Motivation

The problem of control of Micro Aerial Vehicles (MAVs) have been solved in a multitude of ways by traditional control theory methods like the Linear-Quadratic Regulator (LQR), and Model Predictive Control (MPCs) [62] [63]. These systems have proven to be accurate, robust and well-performing for many different tasks and environments. Their performance is also adequate on the topic of navigation in collision exposed environments, [64]. Traditionally, there has been a formal divide between control algorithms and path planning algorithms. Path planning algorithms often only consider a path in terms of edges and vertices without considering the system's dynamics. In geometrically confined environments, the traditional methods have to solve a series of heavily constrained optimisation problems to propose reasonable trajectories. A MAV is limited in computational power as any additional weight and space lead to shorter battery life. As a result, the computationally heavy optimisation problems lead to slower, and therefore shorter, trajectories.

Biological systems like insects, birds and bats are naturally able to perform collision-free flight and navigation in geometrically constrained environments with a high level of precision. This indicates that there should exist a learning-based way of performing collision-free navigation and flight control. Such a method is tested and elaborated on in [65]. Here the desired response to the environment was learned through imitating the response of an expert planner. This paper shows trajectory results similar to the expert at a fraction of the computational cost and time.

Building on this knowledge, we will attempt a learning-based approach to solve collision avoidance and high-level control of a MAV in geometrically constrained environments. Noting again the traditional diviosion between path planning and flight control, a path planner will typically feed a waypoint reference to a flight controller without considering the dynamics of the system. Thus, our approach will represent an integrated End-to-end approach which will do both path planning and dynamic control of the MAV using the same model.

4.2 Related Work

In [66], the authors present a decentralised algorithm for doing multiagent collision avoidance. They show that this Reinforcement Learning based navigation algorithm scales to multiagent scenarios with more than 2 agents. Using a LiDAR sensor, a ground-based robot localises itself with respect to the map and also detects dynamic obstacles that it uses this to generate a collision-free trajectory.

In [56], an Imitation Learning approach is taken to teach a ground robot to navigate in a cluttered 2D environment using an expert operator. The immediate surroundings is sampled using a laser range finder and fed to a Convolutional Neural network which extracts the relevant features from the environment. Together with the information about the goal position, these features are then fed to a fully connected neural network representing the agent's policy.

In [67], the authors build upon the system in [56] by including a Reinforcement Learning step after the initial Imitation Learning. They still use the laser scanner, but instead of a CNN extracting features, they use a Minimum pooling strategy which is then fed directly to the agent. The Reinforcement Learning agent is trained using an extension of the TRPO algorithm, namely the Constrained Policy Optimization (CPO) algorithm.

In [68], the authors develop an autonomous car that navigates using a fusion of monocular vision and a LiDAR 2D depth map. They feed each of the corresponding data streams into its own CNN, which are merged and flattened after a couple of layers to produce an action for the car. This method uses Deep Q- networks and a discretisation of the action space into five possible actions.

In [69], the authors train a cross-modal Variational Autoencoder whose inputs are an RGB image and the relative position of the next goal. The latent space of this autoencoder is then fed directly to a control policy neural network, which only has this latent space as input. This agent then controls a small MAV with velocity commands, which tries to find and fly through gates on a MAV racing track.

In [70], the authors present a fixed-time path generation algorithm that produces optimal motion plans for static environments using a stepping neural network approach. The core of the proposed *OracleNet* algorithm consists of Recurrent Neural Networks to determine end-to-end trajectories.

In [71], the authors conduct an exhaustive study of both end-to-end methods and unsupervised-learning-based architectures for ground robot navigation in dynamic environments. The state representation is a 2D laser scanner angular map with which an autoencoder is trained, and a long short-term memory network (LSTM) is used to predict future sequences. These two are then fed to a fully connected network which generates a control action for the robot.

In [72], an algorithm for ground robot navigation in crowded environments using Deep Reinforcement Learning is developed. The input to the model is a cross-modal representation consisting of the goal and robot state, the known states of pedestrians and other robots, the latent space from an autoencoder trained on an occupancy grid, and an angular map. These are all concatenated and fed into the Reinforcement Learning agent, which generates trajectories.

In [73], the authors present an algorithm for collision avoidance of a fixed-wing MAV that uses a ray-casting method with rays protruding from the tip of the MAV to measure the distance to a potential object in its path. The agent controlling the MAV is trained using an experience pool consisting of human expert experiences. Based on the DDPG and MPC algorithms, a novel algorithm MEP-DDPG is designed to train an agent using both experience pools and the output of an MPC algorithm. In [74], the Rapidly Random-exploring Tree (RRT) algorithm is improved using a CNN model to generate a

nonuniform sampling distribution. This way, costly and unnecessary computation is avoided as the CNN model can predict the probability distribution of the optimal path on the map.

In [75], a control network is fed the latent space from a trained Variational Autoencoder and the output of a recurrent neural network to drive a racing car in a 2D simulation. The variational autoencoder was trained on images from the simulation environment to be able to make a compact representation of the input data. In addition, they showed that a memory capacity greatly improved the agents performance.

In [76], the authors propose a Reinforcement Learning-based algorithm for the manipulation of objects using robotic arms. The first step of their algorithm is to train a deep spatial autoencoder to acquire a set of features, with which the Reinforcement Learning agent is later trained on.

In [77], the authors present a learning-based pipeline to do local navigation of a quadrupedal robot in environments with both static and dynamic obstacles. The robot is given high-level navigation commands, and is able to safely navigate around obstacles using the data from a depth camera. Their approach combines the output from a Variational Autoencoder with an RL agent and a LSTM layer.

Transitioning between training a policy on demonstrations and training using a deep RL method can be tricky since they can have vastly different gradients. In this paper [78] from 2019, the authors propose an actor-critic framework merging a behavioural cloning loss from an expert with a Q-learning loss. This method allows for a smoother transition between Imitation Learning methods and RL methods.

4.3 Problem Formulation

The purpose of this chapter is to discuss and propose a data-driven method of utilising the compressed latent space discussed in chapter 3 for executing collision-free navigation and flight control. The navigation and control will be performed by a neural network, referred to as the learning-based agent, trained to reach a goal point in a collision-free manner given a geometrically constrained environment.

This chapter aims to design an agent with a policy capable of mapping a latent space representation of the surrounding environment and a state vector into an acceleration vector. Details of the network architectures for the agent will be discussed further in section 4.5.1. The output acceleration vector will be handled by the inner control loop system of the MAV. This is illustrated in figure 4.1. The conditions of the output will be further discussed in the section about wanted behaviour of the MAV 4.4.7.

As observed in figure 4.1, we can keep the decoder and use it to indirectly evaluate how well the features in the latent space describe the immediate surroundings of the MAV. For example, if the reconstruction completely misses an object, it is a strong indication that the latent space does not contain the necessary information for the agent to be able to generate optimal actions.

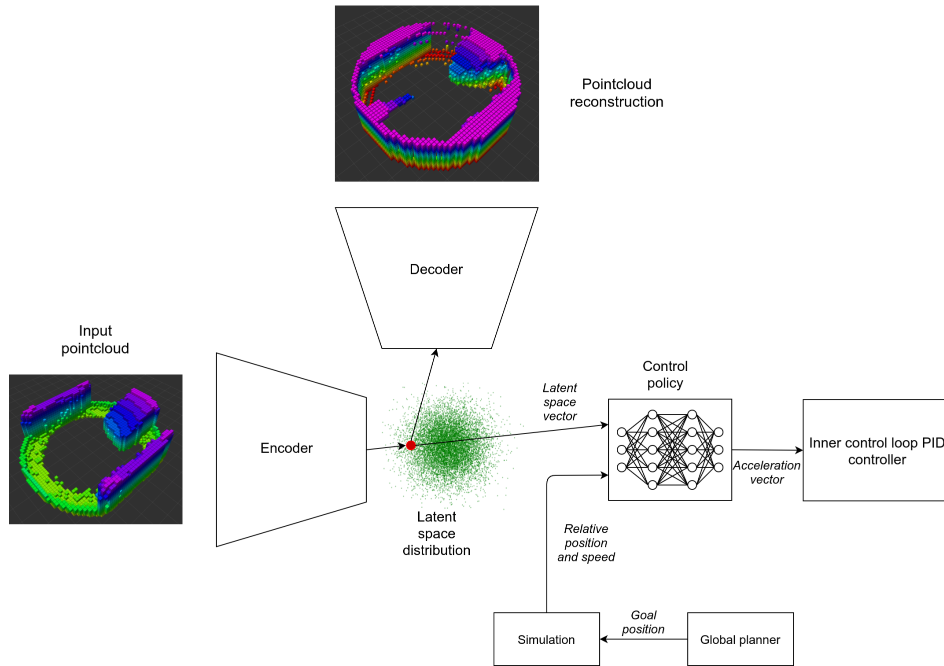


Figure 4.1: The agent has a latent space representation of the environment and the position relative to the goal, and the speed as input. The output of the agent is an acceleration vector fed into the inner control loop of the UAV.

4.3.1 End-to-end Versus Modular Training

It is worth discussing the implementation in this thesis in the context of other approaches like the method presented in [68] which uses an end-to-end approach in terms of neural network architecture. We have split the perception and control sections into two separate networks, and two separate training processes. First, we train the autoencoder using an unsupervised approach, and then we train the agent on the latent space from the autoencoder. However, consider the case of training just one network end-to-end. This would be an architecture that would be some combination of the encoder part of the autoencoder and the agent.

The first immediate advantage of making such a network is that we could know that the features collected in the first layers of the networks were optimised to be strictly the ones needed for collision avoidance. With our implementation, we have a feature extractor that is regarded as a black box. The autoencoder may just as well extract information irrelevant to the collision avoidance problem.

The drawbacks of such an integrated architecture is that we would have no way of observing how well the network extracted features from the environment. With the current implementation, we can continuously monitor the reconstructed TSDF point cloud from the latent space and watch for inconsistencies. This means that debugging is much easier. For example, if an end-to-end network architecture agent collides, one does not know whether it did so because of a bad feature extraction or confusion from the features extracted.

According to the paper [77], dividing the training into two separate stages yields a more sample efficient Reinforcement Learning stage as well as a superior policy. We use this property, and the fact that the encoding process can be monitored, as motivation for the modular approach we have chosen

in this thesis.

4.3.2 Objective of the Chapter

The overall objective of this chapter is to present, discuss and evaluate two data-driven methods for training a learning-based agent to navigate and high-level control a MAV through confined environments using a latent space representation. This chapter will therefore rely heavily on the ability of the encoder to represent the local environment. First, we will train the learning-based agent using an Imitation Learning (IL) framework using an *expert* to make optimal suggestions with which the agent may be trained. Second, we will train the learning-based agent in a Reinforcement Learning (RL) framework using pre-trained weights from a short IL process. We will discuss the chosen approaches with respect to its performance in executing collision-free flight, and suggest how these methods could be improved. The goal is to have a learning-based agent which can solve both the problem of path planning and high-level control in an End-to-end fashion, with the encoder taking as input a local TSDF point cloud and the learning-based agent calculating an acceleration vector directly.

4.4 Theoretical background

4.4.1 Quadrotor Dynamics

In this section, we will discuss the dynamics of the quadrotor system, as illustrated in the figure 4.2. To describe the attitude and position of the quadrotor, we will be using two coordinate systems; the inertial reference frame $\{\vec{e}_{11}, \vec{e}_{21}, \vec{e}_{31}\}$ and the body-fixed frame $\{\vec{e}_{1B}, \vec{e}_{2B}, \vec{e}_{3B}\}$. The body-fixed frame has its origin placed at the centre of mass for the quadrotor, which means that the first and second axes ($\{\vec{e}_{1B}, \vec{e}_{2B}\}$) of the body frame points span out the plane on which all the rotors lie. The axis $\vec{e}_{3B}$ is placed normal to this plane, such that it points in the opposite direction to the thrust vector of the quadrotor propellers.

Each rotor generates a thrust F_i, and all rotors are placed with equal distances to each other and the centre of mass. In order to balance the torques generated by the spinning rotors, their spinning direction is opposite to that of their neighbors.

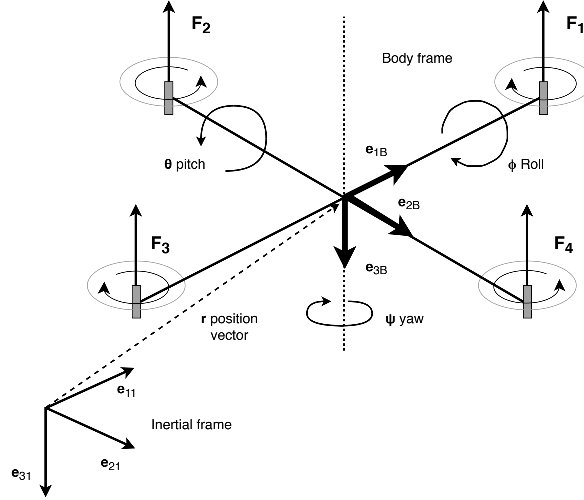


Figure 4.2: Model of the Quadrotor

$m \in \mathbb{R}$	the total mass of the quadrotor
$\boldsymbol{J} \in \mathbb{R}^{3\times3}$	the inertia matrix with respect to the body-fixed frame
$\boldsymbol{R} \in SO(3)$	the rotation matrix from the body-fixed frame to the inertial frame
$SO(3)$	The group of all rotations about the origin of three-dimensional Euclidean space $\mathbb{R}^3$
$\boldsymbol{\Omega} \in \mathbb{R}^3$	the angular velocity in the body-fixed frame
$\boldsymbol{x} \in \mathbb{R}^3$	the location of the center of mass in the inertial frame
$\boldsymbol{v} \in \mathbb{R}^3$	the velocity of the center of mass in the inertial frame
$d \in \mathbb{R}$	the distance from the center of mass to the center of each rotor in the $\vec{e}_{1B}, \vec{e}_{2B}$ plane
$f_i \in \mathbb{R}$	the thrust generated by the i-th propeller along the -$\vec{e}_{3B}$ axis
$\tau_i \in \mathbb{R}$	the torque generated by the i-th propeller about the $\vec{e}_{3B}$ axis
$F \in \mathbb{R}$	the total thrust, i.e. $F = \sum_{i=1}^{4} F_i$
$M \in \mathbb{R}$	the total moment in the body-fixed frame

Table 4.1: Parameters of the quadrotor helicopter

The location of the centre of mass and its attitude with respect to the inertial frame thus defines the configuration of the quadrotor system. This configuration thus lies on the manifold of the special Euclidean group SE(3), which is all rotations and translations in three-dimensional Euclidean space $\mathbb{R}^3$ that preserves Euclidean distances.

Assuming that the thrust of each propeller is directly controlled and that the torque generated by each propeller is directly proportional to its thurst, we can derive the equations of motion for the quadrotor system. This simplification means that we are leaving out the nonlinear dynamics of the aerodynamic thrust capabilities of the propellers with respect to its torque. The thrust vector is defined to be normal to the plane spanned out by the vectors $\{\vec{e}_{1B}$ and $\vec{e}_{2B}\}$, which means that the total thrust f acts in the direction of -$\vec{e}_{3B}$. To transform this thrust vector into the inertial frame, we only need to multiply the vector e_{31} with the rotation matrix $\boldsymbol{R}$: $-f\boldsymbol{R}\vec{e}_{31}$

To balance the torque generated by the propellers, the first and the third propellers are rotating

clockwise, while the second and fourth propellers are rotating counterclockwise when generating positive thrust. We write the torque of the i-th propeller as $\tau_i = (-1)^i c_{\tau f} F_i$ for a fixed constant $c_{\tau f}$. The relationship between the total thrust $\boldsymbol{F}$ and the total moment $\boldsymbol{M}$ can thus be formulated as:

$$\begin{bmatrix} F \\ M_1 \\ M_2 \\ M_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -d & 0 & d \\ d & 0 & -d & 0 \\ -c_{\tau f} & c_{\tau f} & -c_{\tau f} & c_{\tau f} \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \tag{4.1}$$

This matrix is invertible when $d \neq 0$ and $c_{\tau f} \neq 0$, as its determinant is $8c_{\tau f} d^2$. This means that for a given $\boldsymbol{F}$ and $\boldsymbol{M}$, we can find the corresponding thrust F_i for each rotor.

The equations of motion for the quadrotor are thus:

$$\dot{\boldsymbol{x}} = \boldsymbol{v} \tag{4.2a}$$

$$m\dot{\boldsymbol{v}} = mg\vec{e}_{31} - \boldsymbol{F}\boldsymbol{R}\vec{e}_{31} \tag{4.2b}$$

$$\dot{\boldsymbol{R}} = \boldsymbol{R}\hat{\boldsymbol{\Omega}} \tag{4.2c}$$

$$\boldsymbol{J}\dot{\boldsymbol{\Omega}} + \boldsymbol{\Omega} \times \boldsymbol{J}\boldsymbol{\Omega} = \boldsymbol{M} \tag{4.2d}$$

where the *hat map* $\hat{\cdot} : \mathbb{R}^3 \to SO(3)$ is defined as a cross-product operation such that $\hat{x}y = x \times y$ for all $x, y \in \mathbb{R}^3$

4.4.2 Reinforcement Learning

Reinforcement Learning (RL) is a class of Machine Learning (ML) methods concerned with the interaction between a given environment and an agent[36, p.1]. We distinguish between the agent and the environment by defining every state the agent can control directly, as the agent itself. The rest of the system is referred to as the environment. The interaction is based on a series of discreet time steps. First, the agent alters the environment by applying input. The agent then receives a scalar reward based on the input provided and the state of the environment it was applied on. The goal of the agent is to learn the behaviour that maximises the sum of rewards for a series of consecutive time steps. To be able to discuss the RL methods used in this thesis, we first have to go through some fundamental concepts in RL. The background of RL is rich in details and methods with strong links to one another. It is helpful to study the early methods of RL to understand the concepts of the method used in this chapter. Therefore, the topic was rigorously detailed in both project theses [2][3] leading to this master thesis. Some knowledge of basic RL is advantageous for the coming theory section. However, we will elaborate the theory needed to understand the PPO algorithm used in this chapter.

Terminology in Reinforcement Learning and Control Engineering

RL and control engineering concerns themselves with much of the same concepts regarding optimal control. However, in the intersection between RL and control engineering, the terminology is often different. We explain the discrepancies here.

The word *agent* in RL is defined as the acting unit which interacts in the environment. In control engineering, this would be termed the *controller* of the system.

The *environment* in which the agent operates, is defined as the *controlled system* in the context of control engineering.

The *action* made by the agent in an RL problem the input the agent has on the environment, and this would be termed *control signal* by control engineers.

Markov Decision Processes

Markov Decision Processes (MDPs) are stochastic processes in which the *Markov property* holds. This property can be defined as follows:

$$P\{X(t_{n+1}) \in B | X(t_n) \dots X(t_1)\} = P\{X(t_{n+1}) \in B | X(t_n)\} \tag{4.3}$$

The Markov property states that the probability of the next state $X(t_{n+1})$, in a given state space Borel set B, given all the history of the previous states of X, is equal to the probability of the following state $X(t_{n+1})$ given only the current state [36, p.49].

This property of *markovian dynamics* is essential in the field of RL, as *an optimal decision about the next state of the system can be decided solely on the last state.* For this property to hold, one must make sure that the state space encodes the necessary information for this to be the case.

The *Markov chain* is a graph representation of the dynamics of a system where each node of the chain corresponds to a state in which the Markov property holds. The probabilities of transitioning between the nodes are formalised through a *Transition matrix.*

The set S of states in the environment is the finite set $s^1, s^2, \dots, s^n$ which defines the *state space.* Each state s encodes all the information that matters to an agent about its history in a given problem such that the Markov property holds. We define *legal states* as the states that the agent is able to explore, like empty space, and *illegal states* in which the agent is unable to explore, like inside obstacles. The set A of actions in the environment is the finite set $a^1, a^2, \dots, a^k$ which defines the *action space* of the agent. Given a state $s \in S$, the set of actions that can be applied is denoted $A(s)$. The agent uses these actions to control the state of the system. Not all actions can be applied in every state, with a common example being the actions in a state where a collision occurs.

The *transition function* determines the *transition probabilities* of moving from a state $s \in S$ given an action $a \in A$ to a new state s'. This matrix contains the probabilities of transitioning between all reachable system states, and is defined as follows: $T : S \times A \times S \rightarrow [0, 1]$. The probability of going from a state s to a new state s' given an action a, must be in the interval $[0, 1]$.

The *reward function* is defined as: $R : S \times A \times S \rightarrow \mathcal{R}$, where a reward R is generated when going from a state s to a new state s' given an action a. This reward signal is used to label the behaviour of the agent. By maximising this reward, the agent will find the optimal behaviour for interacting with the environment.

The Markov Decision Process (MDP) is formally defined as the tuple $\langle S, A, T, R \rangle$, where S is the set of possible states, A is the set of possible actions, T is the transition matrix, and R is the set of possible rewards. The transition matrix T and the reward function R constitutes what we call the *model* of the MDP.

The *policy* π is a function that determines what action a to apply in a given state s: $\pi : S \rightarrow A$. It can be referred to as a mapping between the state space and the action space. This policy is the set of all the actions an agent will take in every state of a given environment. The agent uses the policy

to interact with the MDP as follows:

1. An initial state s_0 is generated from a given *initial state distribution*
2. The agent chooses an action $a_0 = \pi(s_0)$ based on the policy π
3. Using the transition matrix T and the reward function R, the agent makes a transition to the next state s_1 with a probability given by the transition matrix $T(s_0, a_0, s_1)$ which results in an award given by the reward function $r_0 = R(s_0, a_0, s_1)$
4. The process repeats for the following states in the chain which produces the tuples $\langle s_0, a_0, r_0 \rangle, \ldots, \langle s_n, a_n, r_n \rangle$
5. The process ends when the agent reaches some termination criteria, often a goal state s_{goal}

The interaction is illustrated in figure 4.3 for a general time step t.

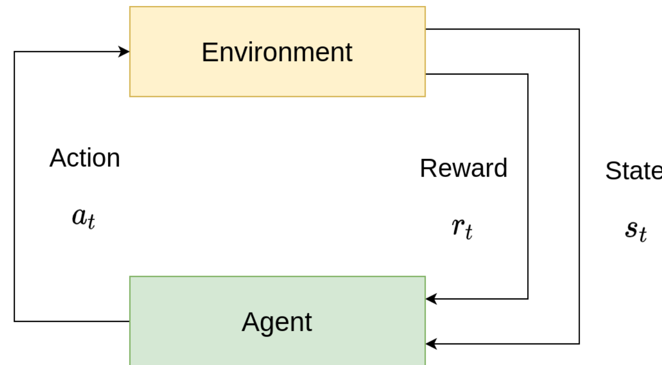


Figure 4.3: Illustration of the environment-agent interaction for a time step t. A state s_t is fed into the agent. The agent applies an action a_t. The environment returns a reward r_t for the state-action pair.

Policy Gradient Methods

Within the field of RL, there are several types of algorithms that work in different types of *state spaces* and *action spaces*. The dimensionality and the nature of these spaces can be very different, and thus the algorithms that try to solve the problems in which these spaces are defined are also equally diverse.

In RL the main distinction between methods are concerned with *discreet* an *continuous* state and action spaces. Consider a problem described with a discreet state space and a discreet action space. In this situation, it is possible to list every action that can be applied for every state. Here, the agent can traverse the state space, attempt different combinations of state-action pairs, and be rewarded. The statistical reward for each state-action combination can then be calculated and stored in a large matrix with states along one dimension and actions along the second dimension. The optimal policy will then be to, for each state, pick the action that has the highest found statistical value in the matrix. However, if the problem is continuous in either the state space, action space or both spaces, the number of state-action combinations will be infinite. This makes storing state-action values in a matrix infeasible, and complicates the RL problem significantly.

Often in the field of robotics, as is also the case in this thesis, the problem is using *continuous* state and action spaces. Practically, this means that the robot we are controlling is moving in a *smooth and*

continuous environment, which then translates to the state variables being smooth and continuous. The same is the case for the action space. The action space presented in this thesis is the output of the RL algorithm, which sends an acceleration vector to the low-level PID controllers of the quadrotor helicopter. This acceleration vector is also smooth and continuous in that both its magnitude and orientation are smooth and continuous values.

We introduced the concept of *Markov decision processes* in the previous section 4.4.2. Here, the *state-* and *action space* is defined as a finite set of actions A and states S. However, we can generalise all the concepts introduced in that section to the case of continuous and differentiable action and state spaces.

When working with continuous action and state spaces we turn to a class of RL algorithms called *policy gradient methods*. These methods consist of *parametrised policies*, meaning that we have a function with a policy *parameter vector* θ which deterministically maps every state s in our continuous state space into an action a in the continuous action space. Many different types of models may be used to parametrise the policy, however in this thesis we will focus on parametrisation using Artificial Neural Networks (ANNs). We define our parametrised policy as:

$$\pi(a|s,\theta) = Pr(A_t = a|S_t = s, \theta_t = \theta) \tag{4.4}$$

Equation 4.4 describes the policy as the probability of the agent taking an action a at time t, given the state s and the parameter vector θ. The parameter vector θ consists of all the weights and biases present in the neural network.

Policy gradient methods try to increase the performance of the parametrised policy by using a function $J(\theta)$ which measures the policy's performance. This function will return a scalar value based on the parameters in the parameter vector θ. We can thus write the general working principle behind policy gradient methods as:

$$\theta_{t+1} = \theta_t + \alpha \Delta \hat{J(\theta_t)} \tag{4.5}$$

This *update rule* approximates a gradient of the performance function $\hat{J}(\theta_t)$ that determines in which way all the parameters in the network must be changed to increase the overall function with respect to the policy's parameter vector θ. Since we are learning from data samples, we are working with stochastic estimates of the gradient of the performance measure function $J(\theta)$. This process is the same as the process described in section 2.1, under "Gradient-based training".

This process can also be expressed as a maximisation problem:

$$\max_{\theta} E\left[\sum_{t=0}^{H} R(s_t) \middle| \pi_\theta\right] \tag{4.6}$$

This maximisation problem tries to find the optimal parameter vector θ for maximising the *expected sum of future rewards* given the policy network π_θ.

Now that we know that the parametrised policy $\pi_\theta(a|s)$ is stochastic, we can interpret the policy as a probability of taking an action a in state s. This brings with it a very useful property, namely that the parametrised policy is a smooth and differentiable function with respect to its parameter vector θ. Moreover, since we know that the function is smooth and differentiable, we can also change its parameter vector in a gradual fashion. This makes it possible to avoid sudden and potentially

disastrous changes in the policy, bringing with it better properties for convergence.

The next challenge we need to overcome is the fact that measuring the performance of the policy is both dependent on the action selection *and* the distribution of states in which those actions are made. Furthermore, both of these distributions are affected by changes in the policy's parameter vector θ. We can imagine the case in which we isolate just one state and observe how the action changes as we do changes in the parameter vector. However, knowing how the distribution of future states is changed as we change the actions the agent chooses is a function of an environment that will be unknown to us.

The *policy gradient theorem* is the answer to this challenge:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{4.7}$$

$\mu(s)$ describes the weight of the error in the given state s, which practically relates to its frequency in the dataset. Normalising by this value ensures that states that are visited often do not get larger gradients than those visited infrequently.

$q_\pi(s,a)$ denotes the *value* (Q-value) of being in a specific state s, taking the action a, and following the policy π from thereon. This term is used to provide a magnitude to the gradient; if this term is large, and thus the action gives a large value, the gradients get scaled proportionally, and if it is small, the opposite is the case.

$\nabla \pi(a|s,\theta)$ denotes the gradient to the probability that an action a is taken in state s, given a parameter vector *theta*. This, combined with the Q-value, tells us that if a state-action pair generates a lot of value, the probability of that state-action pair will be increased by a lot. In contrast, a proportionally small value will scale state-action pairs that generate a low value.

Monte Carlo Policy Gradient

The policy gradient theorem in equation 4.7 is theoretically useful but practically not very helpful for our algorithm. We cannot sum over all actions in every single state as we have a continuous action and state space. Therefore this formula needs to be further developed.

First, we remove the summation over all states and the term $\mu(s)$, which again is a measure of how often different states occur under the current policy. Under the assumption that the agent following the policy π will visit each state in the state space with the same probability, this can be removed.

We replace the s with S_t, which is the expectation of being in a state in a given timestep under the policy π. This expectation can then be sampled. Lastly, we know that the policy gradient only needs to be proportional to the real gradient since any proportional term will be absorbed into the learning rate parameter α. This now gives us:

$$\nabla J(\theta) = E_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \tag{4.8}$$

To remove the summation over all actions, we will only consider the action a in the current state s. This means that we replace the summation over all possible actions with an expectation under the policy π, which can then be sampled:

$$\nabla J(\theta) = E_\pi \Big[q_\pi(S_t,A_t) \nabla \pi(A_t|S_t,\theta) \Big] \tag{4.9}$$

We need to normalise this gradient with respect to the different probabilities of actions. Doing this we will normalise the gradient magnitudes so they are not biased towards those actions that are performed frequently. Therefore our expression needs to be divided by $\pi(A_t|S_t, \theta)$. To simplify further, we also use the following identity: $E_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t)$. G_t then denotes the total *discounted sum of future rewards*. Our equation now looks like this:

$$\nabla J(\theta) = E\left[G_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}\right] \tag{4.10}$$

This is in fact the update rule for the plain-vanilla policy gradient algorithm REINFORCE [79]. We use the identity $\nabla ln(x) = \frac{\nabla x}{x}$ to simplify, and insert the gradient estimate into our stochastic gradient ascent algorithm:

$$\theta_{t+1} = \theta_t + \alpha G_t ln(\pi(A_t|S_t, \theta)) \tag{4.11}$$

The intuition behind this algorithm is very simple; every increment along the policy gradient is proportional to the return G_t. A positive step along the gradient will increase the probability of the given action, and when this gets scaled by the expected future sum of rewards we see that actions with higher returns increase their probabilities proportionally. This algorithm is a *Monte carlo*-based algorithm since it samples all the rewards through the whole episode and later uses that sequence of rewards to calculate the discounted sum of future rewards at every time step.

Baseline

To reduce the variance from the G_t-term during training, we may introduce the concept of a *baseline* $b(s)$ [36, p.329]. This baseline may be any function which does not depend on the action a at a given state s. We thus have a generalisation of the Policy gradient theorem 4.7:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \big(q_\pi(s, a) - b(s)\big) \nabla \pi(a|s, \theta) \tag{4.12}$$

If we insert this concept into our stochastic gradient ascent algorithm, we get the following:

$$\theta_{t+1} = \theta_t + \alpha\big(G_t - b(S_t)\big) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{4.13}$$

When designing the baseline function, it is important that it is in the same order of magnitude as the values of an action. We will then end up scaling the magnitude of the gradient corresponding to some *advantage* of a specific action compared to the baseline. This process can be generalised into a general *advantage function*:

$$A(s, a) = Q(s, a) - V(s) \tag{4.14}$$

This function represents an essential concept in RL. Suppose the value of an action, or the discounted sum of future rewards in the Monte Carlo setting, is larger than a baseline represented by a value function. In that case, this means that this action will improve the overall policy of an agent. Conversely, if the advantage is negative, it will worsen the performance of the policy. This means that,

in the context of equation 4.13, the probability of a bad action will be lowered as the gradient becomes negative, and vice versa with a positive action.

Actor-critic Methods

If we approximate the value function used in the previous advantage function calculation with a parametrised function approximator, we have what is called an *Actor-critic method.* This means that we now have two neural networks; the *actor* network $\pi(A_t|S_t, \theta_t)$ which for a given state S_t and parameter vector θ_t generates an action A_t, and the *critic* network $\hat{v}(S_t, w)$ which for a given state S_t and parameter vector w generates a value estimate. The training of the critic network is a classic supervised-learning problem, where the neural network is trying to predict the discounted sum of future rewards given the current actor policy and a state, and is trained using the samples from the trajectories generated by the actor and their actual discounted sum of future rewards.

With the parametrisation of the value function we introduce *bootstrapping.* This refers to the fact that a future estimate will be calculated using an existing estimate. Thus, we introduce bias and an asymptotic dependence of the function approximation. This is offset by the fact that bootstrapping reduces the variance in the training and thereby accelerates learning [36, p.124]. We integrate this into our stochastic policy gradient ascent framework:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha\Big(G_t - \hat{v}(S_t, w)\Big)\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)\Big)\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \end{aligned} \tag{4.15}$$

To be able to learn continuously, we replace the G_t which corresponds to the actual discounted sum of future rewards in a complete simulation episode with $R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$ where γ denotes the discounting factor. Now we can apply the gradient ascent at every timestep.

Trust Region Policy Optimization (TRPO)

To understand the Proximal Policy Optimization (PPO) algorithm which is used in this thesis, it is very useful to begin by explaining the Trust Region Policy Optimization (TRPO) algorithm [80]. We know that calculating a proper step length α for gradient descent is crucial for convergence from standard optimisation problems. In algorithms that use a constant step length, like the Deep Deterministic Policy Gradient algorithm, we could end up in the situation where optimal solutions could be missed entirely because of the step length striding completely past it. It could even happen that an optimal solution would be lost after the algorithm nudged the optimisation process away. In general, the step direction that should be taken is usually well calculated in that it finds the steepest direction with which to descend. However, the step length α must be chosen to ensure a monotonic improvement to the overall function value. Trust-region methods like TRPO do precisely that.

It is worth noting that in the context of RL, straying away from an optimal point because of bad step size is much worse than in normal optimisation problems. This is because the data the agent is training on is not static, and any change in the policy network which results in a worse policy will give the agent worse observation data with which to learn.

If we replace G_t in equation 4.10 with the advantage function $A(s,a)$ from equation 4.14, we have the following gradient ascent algorithm:

$$\nabla_\theta J(\theta) = \hat{E}_t\Big[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t\Big] \tag{4.16}$$

Note the use of hats $\hat{ }$ which is used to indicate that we are here working with estimates based on sampling. We rewrite the policy gradient as a loss function L^{PG} which we aim to maximise:

$$L^{PG}(\theta) = \hat{E}_t\Big[log\pi_\theta(a_t|s_t)\hat{A}_t\Big] \tag{4.17}$$

if we instead use *importance sampling* we may use a state-action sampling which is based on the old parameter vector θ_{old}:

$$L^{IS}_{\theta_{old}}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \tag{4.18}$$

Now we have the basic optimisation problem which the TRPO algorithm tries to maximise. To constrain the optimisation process from moving too far from the previous policy, we will also include a constraint to the step size. Using the KL-divergence explained in equation 2.9 we can measure the difference between two random distributions. If we ensure that the new policy does not violate some threshold δ from the old policy using this KL-divergence, we can constrain the step size accordingly. The TRPO algorithm is formulated as follows:

$$\begin{aligned} \max_\theta \quad & \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \\ s.t. \quad & \hat{E}_t\big[KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(., s_t)]\big] \leq \delta \end{aligned} \tag{4.19}$$

PPO

In this thesis, we will use the Proximal Policy Optimization algorithm (PPO), an iteration of the TRPO algorithm. The goal behind PPO is to simplify the implementation and thereby decrease the computational complexity in the training phase. It is shown that it manages this while at the same time having comparable performance to other state-of-the-art algorithms [81].

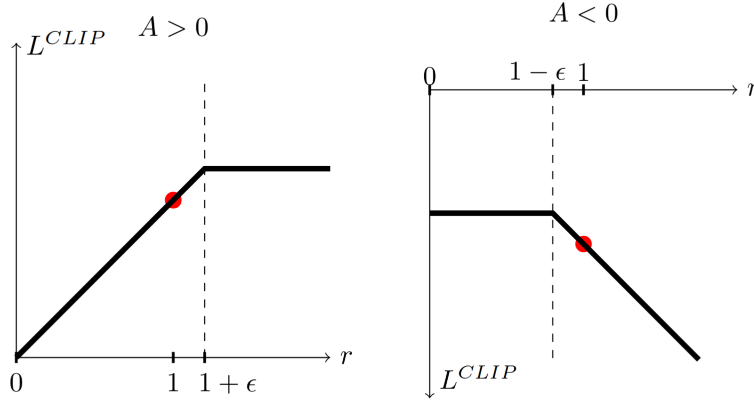
The most computationally expensive operation during the TRPO training is calculating the KL-divergence and doing a backtracking line search. See algorithm 2 for reference. PPO cleverly avoids this by including the constraint in step length directly into the optimisation objective itself.

Before defining this optimisation objective, we have to define the probability ratio $r_t(\theta)$ which is familiar from the TRPO optimisation objective in equation 4.19:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{4.20}$$

Practically, we can see that $r_t(\theta_{old}) = 1$. We rewrite the objective function from TRPO using $r_t(\theta)$:

$$L^{CPI}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \hat{E}_t\big[r_t(\theta)\hat{A}_t\big] \tag{4.21}$$



(a) The case when the advantage estimate is positive (b) The case when the advantage estimate is negative

Figure 4.4: Clipped objective function visualised with respect to the clipped probability ratio

Just like in the case of the TRPO algorithm, we need to make sure that the policy update is constrained in such a way that the policy network is not pushed into a region in parameter space where the future collected data will be of such a quality that the algorithm never recovers again. Therefore, instead of solving this problem using a KL-divergence constraint, we will use a clipped objective function.

$$L^{CLIP}(\theta) = \hat{E}_t\big[min\big(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\big] \tag{4.22}$$

To understand this clipped objective function better, let us look at figure 4.4. We have two different cases; a case for when the advantage estimate $\hat{A}_t$ is positive 4.4a and another for when it is negative 4.4b. Let us begin by looking at the case in which the advantage function is positive. We can see then that whenever the probability ratio $r_t(\theta)$ is also positive, new probability ratio $r_t(\theta_{new})$ is clipped so that it can only change the values in the parameter vector with a factor of $1+\epsilon$ with respect to the old probability ratio $r_t(\theta_{old})$. Thus, the parameter ϵ limits the magnitude of the positive gradient, which means that it limits how much the increase of probability for a given action is. Conversely, suppose the advantage function is negative and the probability ratio $r_t(\theta)$ is also negative. In that case, we also limit how big the magnitude of the step length along the negative gradient will be. If an action is bad, and is less likely, we limit the step length to $1-\epsilon$.

If the action taken gives both a negative advantage estimate and is more probable, the $min()$ operator in equation 4.22 returns $r_t(\theta)\hat{A}_t$. This means that the algorithm will undo its decision to make the action more probable in the first place, and the step length will be proportional to the quality of the action.

4.4.3 Imitation Learning

Imitation Learning (IL) is a set of methods for replicating a desired dynamic behaviour based on demonstrations of some expert agent [82]. It is closely related to **Supervised Learning** as they both attempt to reconstruct a behaviour given by an underlying data set. However, when doing IL, one has

to account for the fact that the underlying data distribution varies according to the actions that the agent itself makes. This is in contrast with normal Supervised learning, where an independent and identically distributed data set may be hand-crafted for the best performance of the model. While RL is cursed by its necessity of experiencing its own data, IL can learn a policy π directly from a dataset $\mathcal{D}$. This implies two things. Firstly, IL is constrained to the performance of the data generated by the expert. However, it can outperform the expert by, for example, manipulating the data to remove unwanted behaviour. Secondly, IL is immensely more sample efficient than RL, with some works exhibiting an exponential difference between RL and IL [83]. This means that having IL as a first step in the RL process will save a lot of computation, as the weights of the neural network of the agent are already tuned in the right direction. Practically this means that our MAV can learn basic skills like hovering and moving at a constant speed from the expert, thus saving a significant amount of time by not having to learn this through RL.

Defining the Imitation Learning Problem

The expert behaviour is implicitly demonstrated through example trajectories. Each trajectory $\boldsymbol{\tau}$ is defined as a time sequence of extracted features $\boldsymbol{\phi}$ denoted $\boldsymbol{\tau} = [\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_T]$. These features can for instance be the state of the system. Additionally, some context vector containing some information about the task is denoted $\boldsymbol{s}$.

The data set of example behaviour with size N can then be written as the set $\mathcal{D} = \{(\boldsymbol{\tau}_i, \boldsymbol{s}_i)\}_{i=1}^{N}$.

Different Methods

We can distinguish between two different types of IL methods, *Behaviour Cloning* and *Inverse RL*. It is also interesting to distinguish between model-based and model-free methods. In a model-based method the system can be simulated, and the generated policy be deemed feasible. In a model-free method, the system will train faster, but the policy can, in theory, be unfeasible. We will further only discuss the use of the model-based solution.

Behaviour Cloning

In BC we aquire the dataset $\{(\boldsymbol{x}_t, \boldsymbol{s}_t, \boldsymbol{u}_t)\}$. In that case we can derive the policy by solving for the mapping between the state $\boldsymbol{x}$ and context $\boldsymbol{s}$ tuple, and the control input $\boldsymbol{u}$ as seen in 4.23.

$$\boldsymbol{u_t} = \pi(\boldsymbol{x_t}, \boldsymbol{s_t}) \tag{4.23}$$

Inverse Reinforcement Learning

Hand-engineering reward functions refer to when developers tune the reward function by inspecting the MAV behaviour and altering the weights they believe might are the reason for unwanted behaviour. For small scale problems, this method is entirely viable. For larger-scale problems, finding an optimal reward function by hand-tuning can be infeasible for projects constrained by time and resources.

In Inverse Reinforcement Learning (IRL), the reward function is derived from a set of data samples generated by an expert [84]. The optimal behaviour of the expert implicitly details the optimal reward function. Thus, IRL seeks to calculate the reward function explaining the behaviour of the agent.

This project will not use the methods of IRL, so we will not detail it further.

4.4.4 Expert Planner

To generate data for training our learning-based agent using the IL framework, we implemented an *expert* which we used to generate data sets of optimal *actions*. As illustrated in figure 4.5, the expert consists of two modules. The Rapidly Exploring Random Graph algorithm (RRG) [85] is used for generating a path between the current position of the MAV and a goal point given a global map of the whole environment. The first node of the path returned by the RRG algorithm is fed into a PD controller, which converts this into an acceleration vector. This makes the output of the expert directly comparable to the output of our learning-based agent, and we are thus able to train directly on the actions from the expert. Let us look closer at the RRG algorithm.

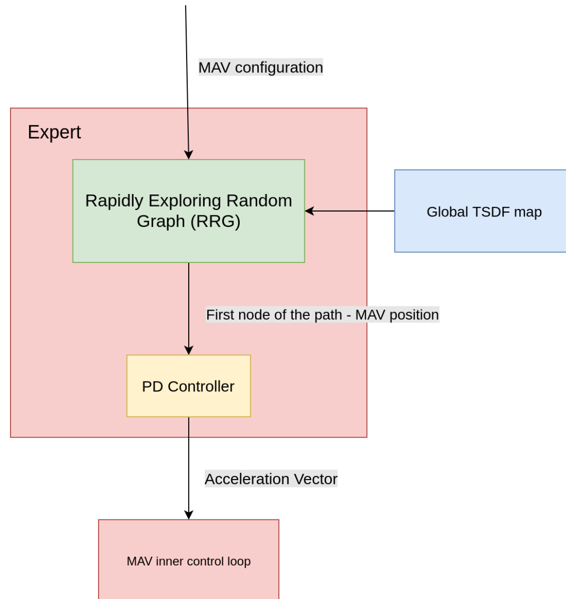


Figure 4.5: The structure of the expert used for Imitation Learning

Rapidly Exploring Random Graphs

The Rapidly exploring Random Graph algorithm is an optimal sampling-based motion planning method [85]. It is both *Probabilistically complete* and *Asympotically Optimal*.

Probabilistic completeness means that the algorithm is guaranteed to converge to a solution given that a valid collision-free path exists in free space as the number of iterations tends to infinity. However, this guarantee is not worth much to us if the path returned by the algorithm is non-optimal. This why the notion of Asymptotic Optimality is useful. An algorithm is asymptotically optimal if it is guaranteed to converge on an optimal path with respect to a cost function as the number of iterations tend to infinity.

RRG is an incremental algorithm to build a connected roadmap. It will first try to connect the nearest node to a given new sample. If this attempt was successful, the new node is added to the set of vertexes. Every time a new sample x_{new} is added to the set of vertexes V, the algorithm tries to make connections from all other vertices in V that are located within a ball of radius r as defined in equation 4.24. The variable γ_{RRG} is defined in equation 4.25.

$$r(card(V)) = min(\gamma_{RRG}(log(card(V))/card(V))^{1/d}, \eta) \tag{4.24}$$

$$\gamma_{RRG} > \gamma^{\star}_{RRG} = 2(1 + 1/d)^{1/d}(\mu(X_{free})/\zeta_d)^{1/d} \tag{4.25}$$

where :

η	the constant of the local steering function
d	dimensions of the space
$card(.)$	Cardinality
$\mu(X_{free})$	Lebesque measure of the obstacle-free space
ζ_d	volume of the unit ball in d-dimensional Euclidean space

The local steering function of the system defines its behaviour from one time step to the next and is thus a useful limiting factor when constructing paths. The Lebesque measure of obstacle-free space is the equivalent to a volume in a 3-dimensional euclidean space, but generalised to a d-dimensional space.

At each discrete time step of the expert, a bounded area around the MAV is uniformly sampled. This set of sampled points are then passed to the RRG algorithm, which calculates an optimal path.

The complete algorithm can be found in algorithm 4.

Generation of the TSDF Map for the Expert

The expert needs an apriori TSDF map of the whole environment to do the sampling operation. We have used two different methods of creating this TSDF map. The first is to use the Graph-Based exploration algorithm proposed by [7] to iteratively build the TSDF map as the MAV explored the environment and then save it. This was only used off-line.

The other method is to use the Ground Truth module from the Voxblox software package [31]. Voxblox Ground Truth is a module for automatically generating the TSDF maps directly from the Gazebo world file. We wanted the agent to have as general a policy as possible; therefore randomly scrambling environments was used. The module creates a map TSDF map in real-time, making it very useful for generating a new global TSDF map after every random scrambling of the environment.

4.4.5 The Data Aggregation Algorithm (DAgger)

A challenge when doing IL is the continuously changing underlying data distribution. Unlike plain-vanilla supervised learning, one cannot assume an independent and identically distributed training data set. This is because the training distribution itself is generated using the models' own actions. The naive approach to solving this problem is by ignoring it completely and train an agent directly on the action of an expert $\pi^{\star}$ through a set of states $d_{\pi^{\star}}$ which are then visited by the expert. We can formulate this as the following mathematical process:

$$\hat{\pi}_{sup} = \underset{\pi \in \Pi}{\arg\min}\, E_{sd_{\pi^{\star}}}[l(s, \pi)] \tag{4.26}$$

where $l(s, \pi)$ is a given loss function. The fundamental flaw with this approach is that the agent

only sees a distribution of states s which are already optimal given the expert policy $\pi^\star$. Practically, this means that as soon as the agent finds itself outside the set of states $d_{\pi^\star}$ chosen by the optimal policy, it has had no similar training data and therefore will not perform well.

For teaching our agent the actions of the expert planner, we will use the Data Aggregation (DAgger) algorithm from the paper [86], and seen in algorithm 5. This algorithm solves this problem by efficiently exploring the state space. At the first iteration of the algorithm, a dataset of trajectories D are gathered directly from the expert's policy. It then trains a policy $\hat{\pi}_2$ on this dataset. This serves as a way to have some initial training of the parameter weights of the policy π, which ensures that the policy is changed in the right direction. For the following iterations, the current policy $\hat{\pi}_n$ is used for generating new trajectories, and these are added to the dataset D together with the expert actions $\pi^\star(s)$ at each state in that trajectory. After each new trajectory has been added to the training dataset, the next policy $\hat{\pi}_{n+1}$ is then trained on this dataset. Thus, by training each new policy on the aggregate of all previously collected trajectories, the algorithm efficiently explores the state space while learning what the expert would do in each of those situations.

The algorithm also includes a tunable parameter β_i, which lets us modify the policy that the agent uses to explore the state space. Instead of having the policy π_i at iteration i be strictly the result of the estimated policy $\hat{\pi}_i$ trained on the aggregate dataset up to iteration i, we can, with a probability β_i, choose the action of the expert policy $\pi^\star$. This can serve as a way to minimise the visitation of states which will be irrelevant as the agent policy improves. At the beginning of the training, we may have a large β_i, and decay this probability as the policy improves. This results in a policy π_i at iteration i formulated as follows:

$$\pi_i = \beta_i \pi^\star + (1 - \beta_i)\hat{\pi}_i \tag{4.27}$$

4.4.6 Differences Between the Imitation Learning Problem and the Reinforcement Learning Problem

IL and RL are groups of ML methods that are very versatile and used for various applications in the robotics community. The advantage of IL-based methods is a high rate of convergence. It requires less resources in the form of computational time and training data. It has shown remarkable performance in replicating the behaviour of experts. At the same time, the agent is unable to explore the policy space beyond what is provided by the expert. This means it is bounded in performance by the expert. The IL problem is also unable to differentiate between "good" behaviour from the expert and "poor" behaviour. This requires the expert to perform reliably in every situation or to remove data points from the set that can lead to poor performance for the agent.

In general, RL methods are found to create policies that are more able to generalise than policies from IL methods. They are not bounded by the expert but rather by the state-action space the agent traverses and the loss function it uses for optimising. However, continuous RL problems have an infinite state action space to traverse, and therefore requires an immense amount of data to converge to a valid policy. This complication is apparent in robotics for even small continuous state-action spaces. For example, the problem of collision-free navigation with an encoded 3D environment has a very complex state space, making it very prone to convergence towards non-optimal local minima.

As previously stated, the procedure in this chapter is first to train the agent using the IL method.

The resulting agent will be compared to another agent which has been trained by IL only for a short period of time, and later trained to convergence using a RL method. This succession of methods is used to exploit the advantages of each method while minimising their weaknesses. In this way, we can prune the state-action tree of bad behaviour before passing it on to the RL method. This is found to ensure faster convergence while still able to achieve the generalisable policies RL methods are celebrated for [67]. The agent's performance will not be bounded by the expert but might be prone to converge on some local minima, based on how the agent has been optimised before being passed to the RL training framework. The method is based on using an IL agent far from convergence but is able to perform some basic navigation. The specific choice of starter network for the RL method will be based on inspection of the convergence rate as well as inspection of performance.

4.4.7 Desired Behaviour of the MAV

The purpose of the system is to navigate through geometrically constrained environments without colliding. The learning-based agent should therefore ensure that the MAV navigates at a safe distance from objects. The MAV should be evaluated in its ability to choose safe paths. This means keeping relatively straight paths whenever possible to mitigate any risks that an oscillating behaviour might inflict on the system. Furthermore, the target should be reached in a controlled manner with little overshooting. The efficiency of the MAV will also be part of the evaluation, but only to ensure the MAV reaching the target in a reasonable amount of time. The ability of the agent to produce environment cognisant controlled movements will be the main area of focus, with efficiency being less important to this thesis.

4.5 Proposed Method

In this section, we will elaborate on how to use data driven methods to train an agent for collision-free flight using the encoded latent space representation of the environment detailed in chapter 3. We will also detail the decisions and reasoning behind the technical choices made for the system. The data-driven methods proposed will be the IL method Data Aggregation (DAgger) [86], and the RL method Proximal Policy Optimization (PPO)[81]. The method constitutes training a model using the DAgger algorithm to train an IL agent until convergence. We will then evaluate this agent. We save the model for every iteration of the DAgger algorithm, which makes us able to use a version of the agent, which is far from convergence and switch to the PPO method and train until convergence with this method. We will then discuss and compare the two different methods. The first being performance of a learning based agent which has been trained solely on the IL method. The second is the learning based agent, initiated with weights of an unconverged IL trained agent, and trained until convergence with the RL method.

4.5.1 The Agent

Here we will discuss and detail the processes behind choosing the two agent networks. The agent is only able to observe the difference between the goal and the MAV, the speed of the MAV and a latent space representation of the environment. The purpose of the agent is to create a policy mapping the observed input to an acceleration vector. The acceleration vector should be calculated in such a

way that a consecutive sequence of acceleration vectors fulfils the wanted behaviour of the MAV, as discussed in section 4.4.7.

The agent is parametrised as a deep neural network. Consequently, the size and complexity of the chosen network greatly affect the behaviour during training and execution, as explained in section 2.1. This affects the agent's behaviour, among others, its sensitivity to overfitting and the number of data samples needed for convergence. Therefore, we will aim for a network that has a complexity that matches the granularity of the large non-linear input space. At the same time, the agent complexity should not be too high to overfit and lose the ability to generalise. We based the discussion around the network as observed in the paper [72] discussed earlier. Here the authors present an agent that has a similar complexity in the input space, consisting of an encoded occupancy map, states of objects and goal and an angular map. In this paper, they used a ground-based robot, which restricts the output of the network to two dimensions, which is significantly less complex than 3D flight. They used three fully connected hidden layers to connect their different input streams into the output layers. Given their similar problem, we adopted a network with similar complexity.

It was also discussed that the complexity of the agent network should, to some degree, be similar to the complexity of the encoder. This is reasonable to expect as the granularity with which the encoder compresses the environment will be reflected in the complexity in the latent space. Thus, the agent needs to be sufficiently complex to map and utilise the information in the latent space. If the complexity of the latent space far exceeded that of the agent, the agent would not be able to make a consistent and optimal mapping from the latent space to a control action. Since both the encoding process in the autoencoder and the mapping from input states to control actions in the agent are considered black boxes, we can only infer their compatibility through exhaustive testing. It may be the case that decoding the latent space just for optimal collision avoidance is a task that requires a much simpler network than fully reconstructing the input TSDF point cloud.

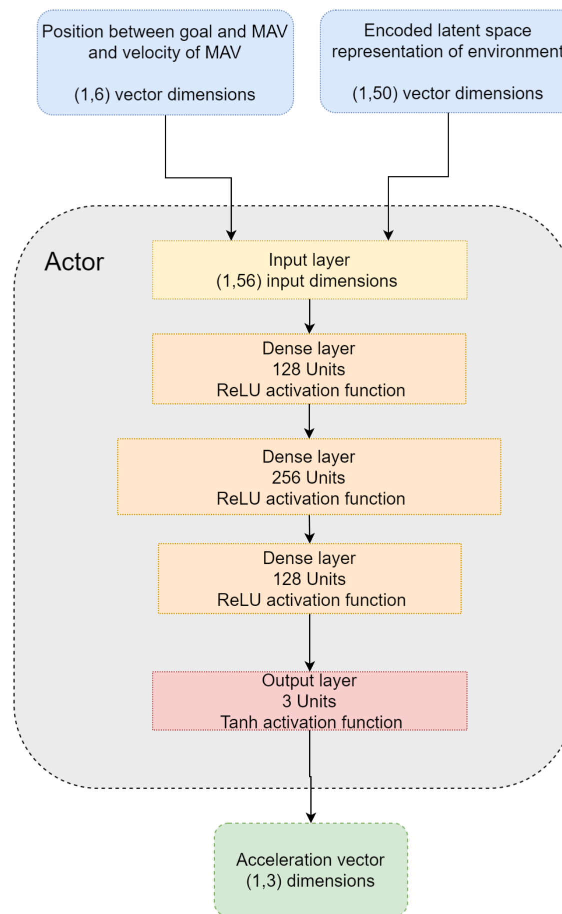


Figure 4.6: The large network architecture.

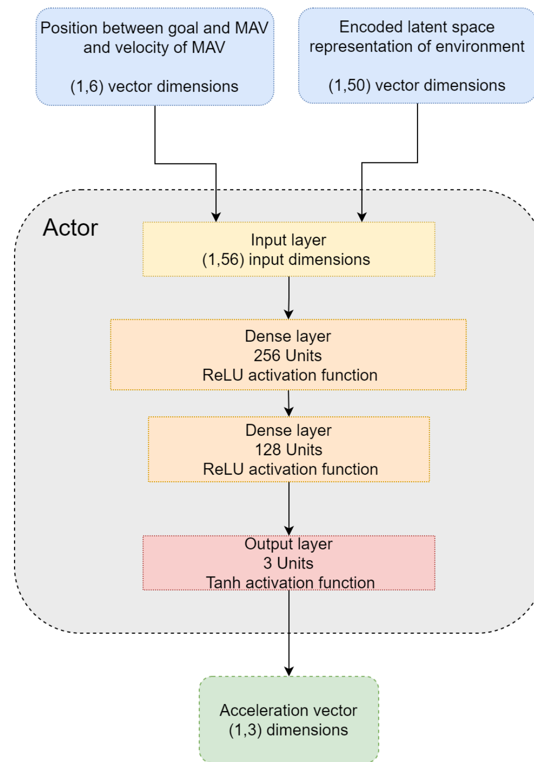


Figure 4.7: The small network architecture.

The process of finding the right architecture also had an element of trial and error, where we trained different networks with different encoders. The resulting architectures were found and tuned through this process. They differ only in the number of hidden layers.

The first chosen network architecture is visualised in 4.6. This architecture consists of three fully connected hidden layers. The number of units in the first and third layers was 128. The second hidden layer had 256 units. The activation function for the hidden layers was the Rectified Linear Unit (ReLU) function defined as $f(x) = max(x, 0)$, as explained in section 2.1.

The second network architecture is visualised in figure 4.7 and had two hidden layers. The first layer with 256 units, and the second with 128 units. The activation function for these layers was the ReLU function, as seen in equation 2.2.

The output of the agent is an acceleration vector for the MAV. This is a vector in 3 dimensions protruding from the origin of the *navigation frame* of the MAV. The navigation frame of the MAV shares the same origin as the *body frame*, but it is always parallel with the *world frame*. The consequence is that this acceleration vector stays the same independently of the MAV's attitude. This avoids an unnecessary coupling between the attitude of the MAV and the acceleration vector which the agent produces. The acceleration vector is translated to roll, pitch, yawrate, and thrust commands using the equations in equation 4.2.

The output layer is a dense layer of three units with the *tanh* activation function. This function is useful as it maps inputs to the space of $[-1, 1]$, which is a useful constraint to have on the magnitude of the acceleration vector. Using the *tanh* activation function also ensures smooth derivatives.

Memoryless Aspect of the Agent

Assuming the Markov property for this system, the agent will only evaluate the information given in the current time step when making decisions. This *memoryless* property of the agent consequently will not produce control actions that are optimal with respect to the previous actions and is a source of oscillatory behaviour over a sliding window of control actions. This may result in *greedy* policies, which do not consider the optimal *sequence* of control actions, but only the optimal with respect to the current state of the system.

4.5.2 Environments

We have created several environments to gather sample data and for evaluating the agent. When discussing and creating these environments, we have looked for some specific properties. Firstly, the intention of having a simulated environment is to facilitate the data-driven methods for collision avoidance. Therefore, the environment should contain a number of obstacles such that most trajectories within a 6-meter radius have some obstacle blocking a straight path, as the goal points generated during training were always at a 6-meter distance or more. The goal point was also spawned at least 1 meter from each wall and 0.5 meters from each obstacle. Further, the environment should not be too cluttered or too complex making small perturbations from the optimal path result in a collision.

To allow for a generalised policy, we wanted the environment to change and increase the number of possible states a MAV could experience. We created 6 environments for training and 1 for testing. The environment has a base of dimensions $8m \times 22m$ with 10 meter high walls. Four obstacles were used. The largest obstacle was a cylinder with a radius of 1.5 meters and a height of 8 meters. We also used a smaller cylinder with a radius of 1 meter and a height of 7 meters. The two remaining obstacles were identical cuboids with a base of 1×1 and a height of 6 meters. One of the training environments can be observed in figure 4.8.

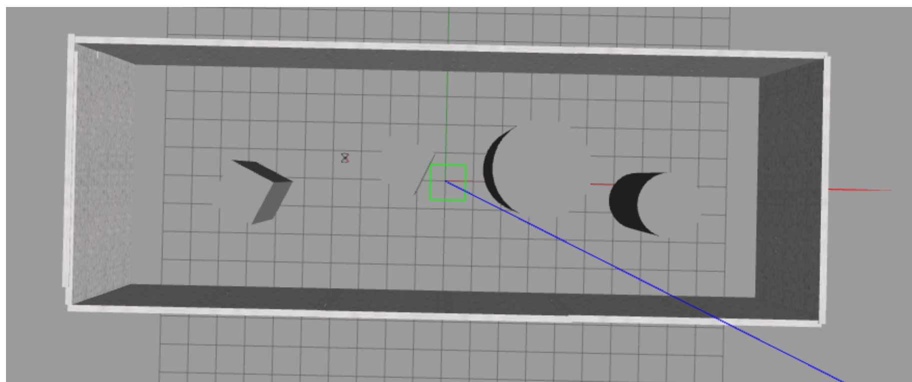


Figure 4.8: One of the six environments used for training the agent.

4.5.3 Autoencoder

In the chapter 3 we investigated the limits to how well an autoencoder is able to generalise, given a random well-distributed training data set. We pushed the architecture to see how complex environments the autoencoder was able to reconstruct. However, in this section, we are much more interested

in having an autoencoder with which we know we will get better results. An inevitable consequence of the data pipeline is that the agent's performance in this chapter will rely heavily on the performance of the encoder. Therefore, we trained the autoencoder on $100,000$ TSDF point cloud samples, which were collected in the 6 training environment environments described in section 4.5.2. To ensure that we did not overfit the encoder, we implemented a validation loss using a separate, similar environment to generate the validation dataset. The network architecture chosen for analysis in chapter 3 was used here as well because of its impressive performance in comparison to its number of trainable parameters. To show some degree of generalisation, the autoencoder has *not* been trained on point clouds collected in the environment in which the agent will be tested later.

4.5.4 Evaluation Metric

In order to quantify the quality of the different models and methods, we will run 100 iterations consisting in spawning new goal points and attempting to reach them. The different runs can either be classified as “goal reached”, “timeout”, “encoder-caused collision” or “agent-caused collision”. The collision class distinguish between two classes of collisions. Encoder-caused collisions will be caused by the failure of the encoder to represent the obstacle in the latent space. Agent-caused collisions will be marked as caused by a sub-optimal or poor performing, policy.

The trajectory is labelled with “timeout” if it has not reached a goal or collided within a specific time after spawning. The time it takes for a timeout to occur in a given run is proportional to the distance between the MAV spawn position and goal spawn position with the function $max_time(distance) = 3 * distance$. As the smallest distance is 6 and the largest is 7, a timeout occurs after between 18 and 21 seconds of flight.

To differentiate between the two classes of collision, we need to determine how well the encoder represented the environment for a given amount of time steps before the collisions. Here we will use the 5 last time steps as this results in the minimal amount of time the agent needs to do an evasive manoeuvre. We define a voxel neighbourhood around the MAV of dimensions $10 \times 10 \times 3$ as the immediate neighbourhood of the agent. As none of our environments can collide upwards, and the LiDAR can only detect sideways, it is less important if this dimension is represented correctly. This corresponds to 2 meter square in the x-y plane and 30 centimeter above and below the drone. We then take the binarised TSDF input to the encoder and the decoded latent vector of the same binarised TSDF and determine how similar they are. To calculate their similarities, we will take the XNOR of the two images. The XNOR truth table can be seen in table 4.5.4. The mean is then calculated as a percentage. Determining if it is a collision caused by the agent or the encoder is done by thresholding. Suppose the incoming neighbourhood TSDF is similar to the decoded neighbourhood TSDF to some given percentage. In that case, it is reasonable to assume the fault is a poor policy from the agent. By observing many collisions, representations of above 90 per cent accuracy captured the object in the neighbourhood quite well. If the representation quality was under 80 per cent it was often missed. Therefore the threshold deciding if the agent or encoder caused the collision was set to 85 per cent.

Input TSDF	Decoded TSDF	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

This is an attempt to create some understanding of the cause of a collision. However, it does not give any guarantees as to what exactly caused the collision. For example, one can imagine a situation in which the decoded TSDF point cloud is correct, but its placement in the latent space is not consistent with similar environments. An agent could thus not be expected to have understood the latent space properly. This represents a complex coupling between the encoder and the agent, and it is apparent that ambiguous training examples from the latent space could lead to a suboptimal behaviour from the agent. Nevertheless, this metric may serve as a strong *indication* as to what happened prior to the collision.

After observing several trained agents, it was decided that the goal radius should be increased when evaluating the agent. This is because the precision needed to reach small goals was very difficult to achieve with such a large state space. It should be noted that the data-driven methods are good enough to reach this level of precision when training and evaluating the agent in a collision-free environment. A possible explanation for the lack of precision in the close proximity of the goal point is that the complexity of the input space itself creates a much more complex mapping between state and action, which itself is much more optimised for collision avoidance than precisely hitting the goal point. Therefore, the ability to reach small goal points is not as emphasised in this thesis compared to the collision avoidance properties of the control actions, even though such precise movements are essential when navigating in collision prone environments in general. The goal radius during evaluation was thus set to 1 meter.

4.5.5 Reward Function Design and Implementation

The structure and parameter values of the reward function implicitly encode how the optimisation process is conducted, and therefore it has a big impact on the agent behaviour. Therefore, it is important that the reward function is closely related to the desired behaviour of the system, discussed in 4.4.7. In these terms, it should punish reckless behaviour leading to collisions and reward controlled movement towards the goal point.

The Reward Function

We will discuss the reward function implemented in this method, term for term.

The first term in the reward function, r_{goal}, is the reward for reaching the goal point. This can be seen in equation 4.28. This equation will be activated when the agent has reached the goal, and be zero otherwise. Thus, this reward is only active for a single time step.

$$r_{goal} = \begin{cases} 100 & \text{if agent is within the goal sphere} \\ 0 & \text{if agent is not within the goal sphere} \end{cases} \tag{4.28}$$

The $r_{collision}$ term is the penalty for collision and is similar to the goal reward. It can be seen in equation 4.29. Here the reward is negative, meaning it will penalise trajectories that result in a collision.

$$r_{collision} = \begin{cases} -100 & \text{if agent has collided} \\ 0 & \text{if agent has not collided} \end{cases} \tag{4.29}$$

The next part of the reward function is the quadratic penalty on state, denoted as $\boldsymbol{x^T Q x}$. This is the reward that is supposed to ensure policy optimisation in the direction of controlled flight in the general direction of the goal point. The first three elements of the state vector are the euclidean distance between the MAV and the goal point. This is symbolised by the variable $\epsilon_i = mav_i - goal_i$, where i is the axis. The other three variables are the velocity of the MAV in each axis. The weights are represented in the diagonal matrix Q. The weights tunable but restricted to negative or zero to ensure consistent gradients.

$$\boldsymbol{Q} = \begin{bmatrix} q_{x_{pos}} & 0 & 0 & 0 & 0 & 0 \\ 0 & q_{y_{pos}} & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{z_{pos}} & 0 & 0 & 0 \\ 0 & 0 & 0 & q_{x_{vel}} & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{y_{vel}} & 0 \\ 0 & 0 & 0 & 0 & 0 & q_{z_{vel}} \end{bmatrix} \quad , \quad \boldsymbol{x} = \begin{bmatrix} \text{difference in position} \\ \text{velocity} \end{bmatrix} = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ v_x \\ v_y \\ v_z \end{bmatrix} \tag{4.30}$$

The next part of the reward function is the quadratic penalty on input observed in the term $u^T Ru$. The matrix R is a negative semi-definite diagonal weight matrix. The matrix R and input vector u can be observed in 4.31.

$$\boldsymbol{R} = \begin{bmatrix} r_x & 0 & 0 \\ 0 & r_y & 0 \\ 0 & 0 & r_z \end{bmatrix} \quad , \quad \boldsymbol{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \tag{4.31}$$

The last term of the reward function penalises the agent for being close to obstacles. This term is observed in equation 4.32, where $\delta_{obstacle}$ is the distance to the closest object and α is a negative tunable parameter. The MAV searches a neighbourhood around the MAV for voxels that are occupied. This neighbourhood is set to 1 meter in the x-y plane and 0.6 meters in the z-axis. This is considered a particularly risky parts of the environments in terms of collisions. The purpose of this term is, therefore, to encourage the agent to stay away from risky areas. The values will be discrete as the voxels are discretisations of the 3D space.

$$\alpha \delta_{obstacle} \tag{4.32}$$

The complete reward function is shown in equation 4.33.

$$r = r_{goal} + r_{collision} + \boldsymbol{x^T Q x} + \boldsymbol{u^T R u} + \alpha \delta_{obstacle} \tag{4.33}$$

Motivation for Using Dense Rewards

It is useful in RL to distinguish between sparse and dense reward functions. A sparse reward function is typically only concerned with the outcome of the trajectory, for example, rewarding goal reached or penalising collisions. In this way, the agent is unbiased toward finding potentially suboptimal policies. However, the absence of intermediary rewards can lead to inefficient learning and difficulty in convergence[87]. On the other hand, labelling each applied action of the agent with a score, as in a dense function, will ultimately lead to higher learning efficiency. This also enables the tuning of specific behaviours. In this implementation, we have chosen to use a dense reward function.

The motivation behind the choice of a dense reward function was to be able to specify a cautious control sequence. In addition, we wanted efficient learning because of the time constraint. The performance of the IL agent indicated that finding a very optimal policy was not too realistic. A more global optimal policy found by a sparse reward system would therefore not likely perform much better. This will be discussed further in section 4.6.2.

Tuning for the RL Reward Function

As discussed in the opening section 1.1.2, this work is a continuation of the projects conducted in the fall of 2020. In both projects, extensive tuning of the RL problem was performed for achieving optimal flight trajectories. The components of the reward function used in those projects are similar to the one in this thesis. Navigation in collision-free environments and navigation in geometrically constrained environments is very different in terms of complexity and behaviours. The tuned values found in the earlier projects were used as initial values and proportions for the reward function with the intention of tuning them to more ideal values for the new problem. Fully training and tuning RL agents by hand engineering reward functions requires extensive resources in the form of computational power and time. Given the nature of the problem, as will be discussed further in section 4.6.2, and the time constraint, the reward function was only tuned to a sufficient degree. For instance, the state and input penalisation was increased to ensure a more stable flight.

4.6 Limitations in the Implementation Process

Here we will discuss the limitations with respect to the scope of the thesis and in regards to the methods used.

4.6.1 Topics of Discussion Regarding the Expert Planner

The expert, as detailed in section 4.4.4, is able to perform very accurate path planning with no collisions. However, different aspects of this path planner need to be discussed when using it as an expert in our IL framework.

First of all, the IL process is not able to distinguish between wanted and unwanted behaviour generated by the expert. This naturally limits the agent's learning to the examples that the expert can provide, and the agent will inherit the behaviour generated by the expert. From this we are faced with a couple of challenges and trade-offs.

Ability to Find Path

There are two main parts of the DAgger algorithm. In the first part of the DAgger algorithm, the expert will control the agent directly. In the second part, it is only used to label the states off-line with corresponding optimal acceleration vectors. This leads to two different discussions of how to tune the variables *size* and *size_extension*.

In the first part of the algorithm, the expert output is examined as a sequence of control inputs. The system is reset if it has not found a path after a number of attempts. This number is set to 7 to allow for the planner to find a path in difficult areas. A failure to find a path here will lead to the expert producing a control action at the current time step with the same values as in the previous time step. We have observed that keeping the same control input across time steps can lead to unwanted behaviour with stochastic properties. If the previous input was very small, then the behaviour will be fine, but if the previous input was large, then the result can be very unstable and unfit for a geometrically constrained environment. Therefore, the challenge is that if the MAV accelerates to a high speed, then the next recorded input needs to aggressively slow the MAV down. This may result in a much more aggressive behaviour of the MAV than what is wanted. The states and actions where the expert planner does not find a path are not logged and therefore not used for training the agent. This part of the algorithm is the shortest, corresponding to around $5\% - 10\%$ of the total data gathered for the IL problem. At the same time, this is the first training iteration, which corresponds to the largest fall in the loss function. Therefore, this iteration is important for how the agent searches through the state space in later iterations.

There are several ways to circumvent this issue, for example, setting the x and y accelerations to zero, or decaying them by some function $x_{t+1} = e^{\gamma} x_t$, where γ is the amount of failed paths. It is important to keep some value for the acceleration in the z-direction to ensure a stable altitude.

In the second part of the algorithm, the expert is called off-line to label the states with acceleration vectors. A failure to find a path here will lead to the data point being dropped from the training set. These data points will thus often correspond to colliding, or high risk trajectories. They are important samples to show the model for it to understand what collision avoidance means. It is important to minimise the amount of times it fails to find a path in this part of the algorithm. At the same time, we want a clear safe zone between the MAV and the objects in the environment.

Both of the issues in the first and second part of the algorithm were resolved by tuning the two parameters *size* and *size_extension*, which are used to determine if a sampled node is free or occupied.

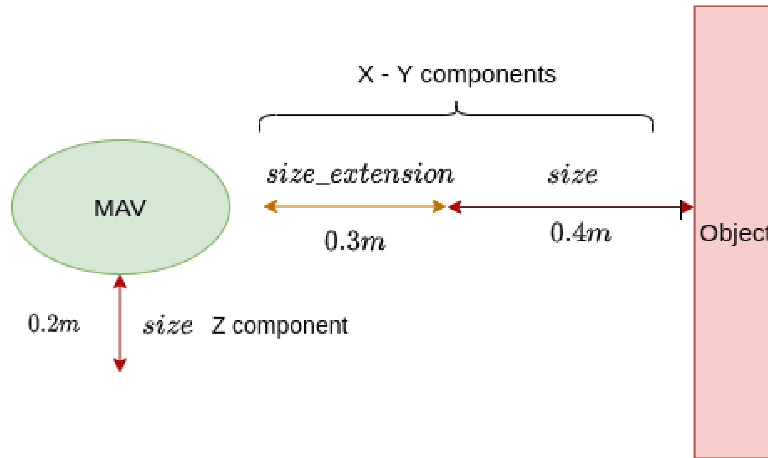


Figure 4.9: Illustration of the distance parameters *size* and *size_extension*.

As noted previously in the expert planner section 4.4.4, the RRG path planner samples points in 3D space within some bounded volume in the close proximity of the MAV to create the path graph. The considered environments have a small enough size that this bounded volume is set large enough to cover the map. Requirements for the dimensions of the path can also be set here. These requirements determine whether the nodes in the sampled space are to be labelled as occupied or free. The variables we tuned were *size* and *size_extension*. The first variable *size* is the minimum required distance to objects for a trajectory to be considered safe. The *size_extension* serves as an additional distance extending the initial minimum distance. The path planning algorithm will first sample the immediate space around the MAV and attempt to find a path using the total distance $size + size_extension$. It will label any node with a distance less than $size + size_extension$ to an object as occupied. If the planner fails to find a path it will try to make a path but only use the distance *size*, increasing the amount of space considered "free". If the planner again fails to find the path, an error is raised.

The *size* variable was set to $[x = 0.4, y = 0.4, z = 0.2]$ and the *size_extension* was set to $[x = 0.3, y = 0.3, z = 0.0]$. These values were found by iteratively tuning, trying to keep the values as high as possible while ensuring that the planner could find viable paths in most situations. For reference, the actual size of the MAV is approximately $[x = 0.4, y = 0.4, z = 0.11]$. The nodes in the sampled 3D space correspond to where the centre of the MAV would be, leaving a minimum margin of 0.2 meters in the $x - y$ plane. With this configuration, we obtained a low rate of failures to find paths while also observing a high rate of viable paths with a safe distance to objects. It was found that even small increases in the *size* parameter would have a large impact on the ability of the expert to find a collision-free path.

Difference in Observability between the Expert and the Agent

One of the principles in IL is that the expert and agent should have the same *observability* of the system. In this training process, the expert has a pre-built global map and can sample every point of the map and do collision checks. On the other hand, the agent is fed a latent space representation that has no guarantees of representing the environment in an easily understood way. Using the decoder during the training, we can get an indication as to how the latent space represents the environment.

However, there are no guarantees that the agent will interpret this information in a similar manner. This makes the IL problem difficult, and a certain level of performance can therefore not be guaranteed.

Consider the edge case where the decoded latent space shows an object between the source and destination when in reality the path is obstacle-free. In this case, the expert, whose planner has full observability of the environment, will produce acceleration vectors that point directly into the object from the point of the agent's view.

This is an example of an inconsistency inherited from the trained decoder. Therefore, it should be fair to assume that the agent would inherit more of these inconsistencies, especially during training. These types of mistakes from the encoder in the training data will most certainly lead to ambiguity regarding the constraints of walls and obstacles. The geometrical constraints of the environment are learned implicitly by showing trajectories avoiding obstacles. However, if some data samples corresponding to the acceleration vector pointing into an object as interpreted by the latent space, the implicitly hard constraints for hitting obstacles are lost. The trained policy based on a dataset containing such edge cases will have softer constraints to collisions and therefore be more collision prone.

Multi Modal Imitation Learning

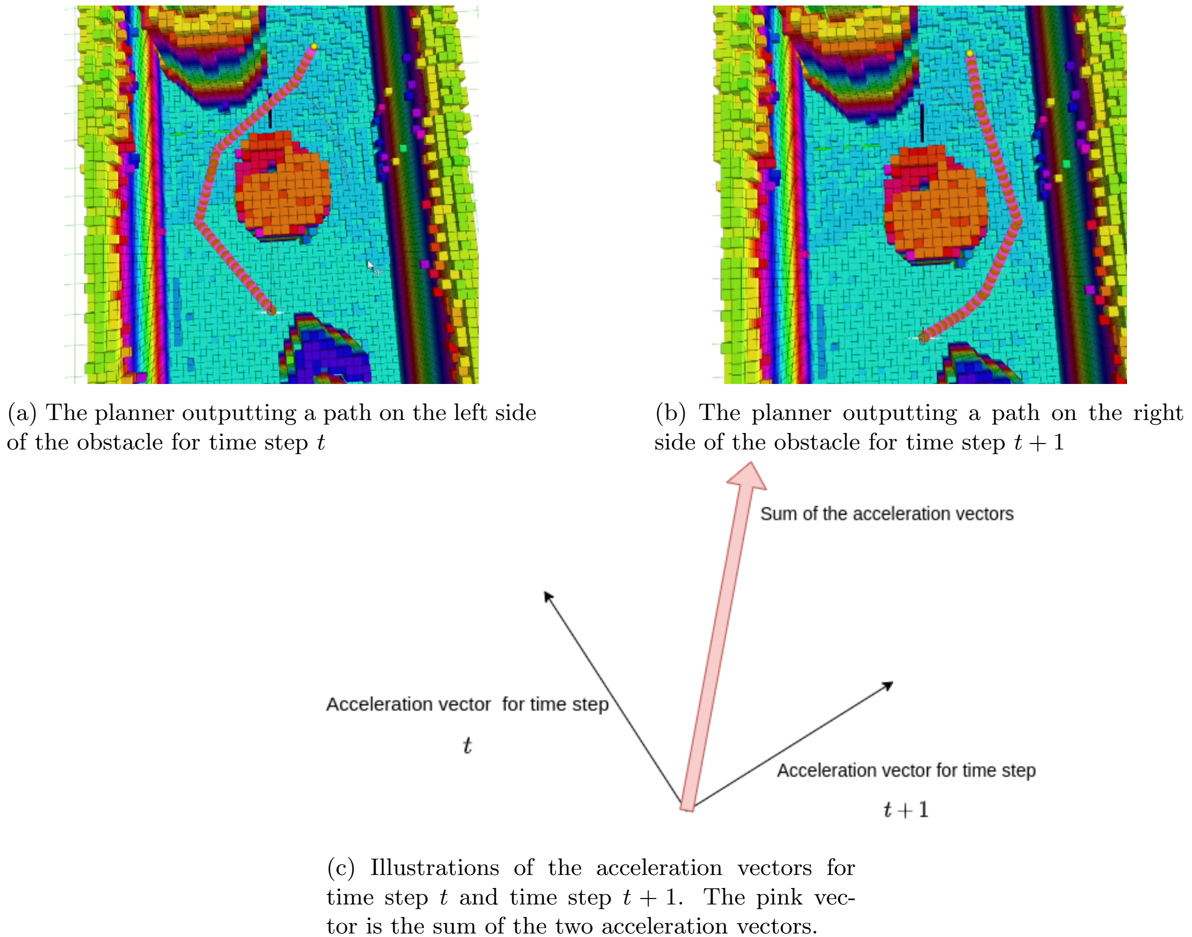
As discussed further in 4.4.4, the planner samples the 3D space uniformly to determine if a part of the space is free or occupied. The planner is initiated anew for every single time step. This means it will create new samples and a new shortest path every iteration. The process of finding the shortest path is, therefore, a stochastic process which can return ambiguous outputs.

Consider the case illustrated in figure 4.10. Here we observe two consecutive time steps, time step t corresponds to image 4.10a, while time step $t+1$ corresponds to image 4.10b. As the environment in these figures is observed from directly above, it should be noted that the orange and red obstacle in the path is a 7-meter tall cylinder. From the two figures 4.10a and 4.10b it is observed that the MAV and the goal points are spawned at almost directly opposite sides of the object. Due to this symmetry, the direction of travel is essentially decided by the stochastic sampling process, which may thus return different acceleration vectors for many consecutive time steps. This will lead to oscillatory behaviour in the data set. This behaviour is problematic for two main reasons.

1. The ambiguous nature of the expert planner can cause oscillatory behaviour in the agent by mapping very different acceleration vector to the same states.
2. The gradient used for optimising the policy will use the mean of the acceleration vector for the same states. This will result in a policy where the MAV will, at times, aim directly for the obstacle.

Let us now consider figure 4.10c illustrating the two calculated acceleration vectors for each respective time step and the sum of the two acceleration vectors illustrated as a thicker arrow. The thicker arrow points almost directly straight forward. This is due to most of the acceleration in the left-right direction being cancelled out. When we optimise the agent based on the training data, we calculate the residuals between the predicted acceleration vectors and the acceleration vector given by the expert for each state. The loss function is a mean square error of these residuals and represents the gradient of our network. Consider now a hypothetical example of a single update batch of 32 data points where

16 of the points have the acceleration vector v_{left} corresponding to image 4.10a. The remaining 16 points have the acceleration vector v_{right} corresponding to image 4.10b. Also imply that the states are very similar to each other $x_t \approx x_{t+1}, t \in [1, 32]$. In this case, the update step of this batch will optimise the policy to generate acceleration vectors that are parallel to the sum of the acceleration vectors as depicted in image 4.10c. Practically this may result in the agent essentially accelerating directly toward the object if there exists any ambiguity as to which side of an object it should pass. This behaviour is unwanted as it undermines the hard constraints on collisions that we want for our system.



(a) The planner outputting a path on the left side of the obstacle for time step t

(b) The planner outputting a path on the right side of the obstacle for time step $t+1$

(c) Illustrations of the acceleration vectors for time step t and time step $t+1$. The pink vector is the sum of the two acceleration vectors.

Figure 4.10: An example of the stochastic properties of the path finding algorithm of the planner. This is visualised in Rviz. The different colored cubes observed in the image are the global TSDF representation of the environment that is used by the planner. The dark pink line is the shortest path, as found by the planner for that time step. This path connect the MAV on the lower part of the picture with the goal point on the upper part of the image. The goal point is illustrated by a yellow sphere.

Alternative Output for the agent

To counteract the multi-modal IL problem, a different structure of the output of the agent was discussed and partially implemented. Unfortunately, it has been left out of the scope of this thesis due to time

constraints. The proposed alternative output head of the neural network can be observed in figure 4.11. Here the predicted output vector would be split into two different acceleration vectors, based on which direction it was pointing in the y-direction. Left acceleration vectors are those that act in the plane in which $y < 0$, and the right acceleration vectors in the other plane where $y \geq 0$. In addition to these two acceleration vectors, we would implement a predictor which would determine which of the acceleration vectors to use for each time step. This predictor would use a Sigmoid activation function to force the predicted output to be either 0 or 1. A prediction below 0.5 would be characterised as using the left acceleration and above 0.5 a right acceleration. The loss function would be a binary cross-entropy loss between the predicted left or right acceleration vector and the sign of the y component, interpreted as binary, in the expert acceleration vector.

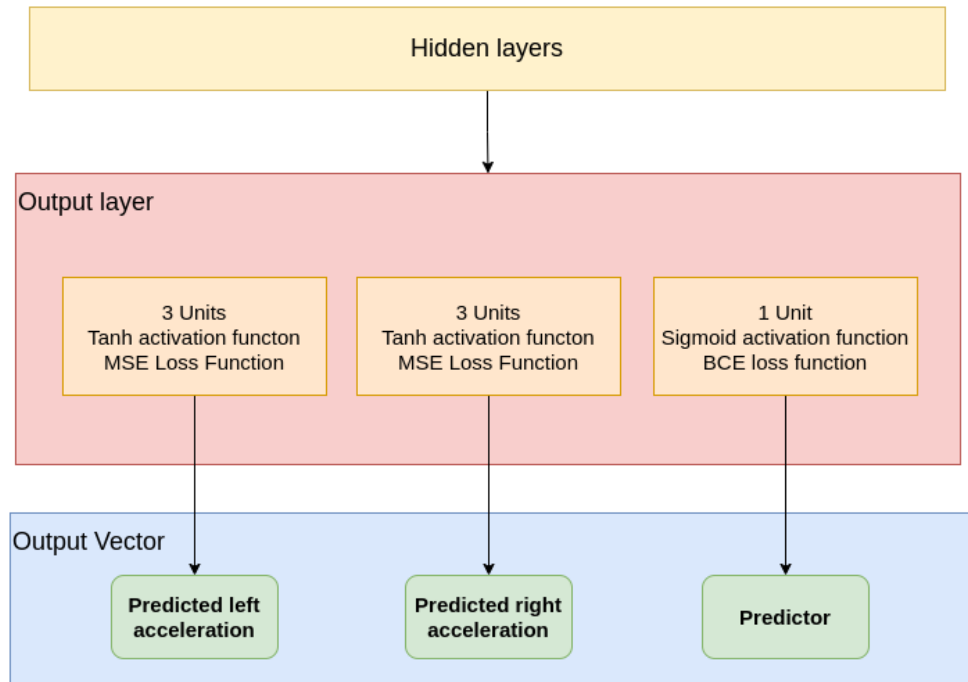


Figure 4.11: Visualisation of the multiple output agent for fixing the multi modal imitation problem.

This type of agent would be spared from the issues originating from the multi-modal IL problem. Here the acceleration vector corresponding to image 4.10a would only optimise the left acceleration output. In this way the sum of data points could not correspond to the mean of the figure as visualised in figure 4.10c. However, this method would only work if the MAV and the goal point were spawned on approximately the same y-coordinates or else it could lead to bias in the policy. In addition, it would not solve the issue of the policy being induced with oscillatory behaviour as the predictor could be trained to oscillate.

To clarify, this network architecture was only partially implemented but had to be abandoned due to time constraints.

Incompatibleness with the Voxblox Ground Truth Module

As discussed earlier, it was important to achieve generalisability of the system. This led us to pursue a direction of randomly scrambling the objects in a given environment. As discussed in the expert section 4.4.4, the expert requires a pre-built map to find paths. For the randomly scrambled environment, the Voxblox Ground Truth module was used for automatic online map generation. Significant amounts of time and resources were used to create this truly randomised environment. However, there was a compatibility issue between the expert planner and the maps generated by the Voxblox Ground Truth module. When using these maps the ability of the expert to find a path was greatly reduced. It was able to find paths in which no collision occurred for a straight path. In these situations, the expert only uses ray-casting to determine if the path is collision-free, and then returns the destination point as the solution. The ability to find non-straight paths was greatly reduced, being only able when the path is very close to a straight line. The issue remains unresolved as it was discovered too late in the process. To circumvent this issue we randomly scrambled 7 environments and mapped them with the Graph-Based planner [7] in exploration mode. These maps were then compatible with the path planner.

4.6.2 The Imitation learning Problem and the Reinforcement Learning Problem

During the work on this thesis, a considerable amount of time has been spent trying to overcome the challenges from the IL process at the expense of the RL process.

As discussed in section 4.4.6, the approach of this chapter is to train an agent using a IL method and then train it further using a RL method. During the development of this thesis, it was found that training a policy that could show properties of understanding the environment was a very difficult task. It was also very dependent on the quality of the encoder in terms of ability to represent the environment and generate consistent latent space representations. The IL is a much simpler problem than the RL problem as the agent receives the suggestion of an expert agent for each time step. This makes the process much more straightforward than in the RL-setting where the optimisation is done through random perturbations of the agent policy in each time step.

ML methods are constituted on the notion that there exists some underlying function $f(x)$ expressing the relationship between the input and the output. The methods revolve around using a parametrised function approximation $g(x)$ such that it approximates the underlying function, $g(x) \approx f(x)$. The approximating function $g(x)$ is the policy network of the agent. We need to make a function approximation $g(x)$, which makes a consistent mapping from the input data of the agent to an action. Since the input of the agent includes the latent space, the function approximation $g(x)$ will only be consistent if there exists a consistent way of mapping from the latent space. Regarding this, there are two points of interest.

1. The problem of collision-free navigation has been solved earlier with classical control methods as described in the introduction.
2. In the mathematical theory regarding neural networks the class of theorems named universal approximation theorems states that a neural network can approximate any function, provided an arbitrary number of units [88].

Considering point 1, we are strongly motivated to believe that there exists a function, $f(x)$, however complex, that maps some representation of the environment into an acceleration vector. The representation should contain sufficient information about the environment. This function should map in such a way as to allow for collision-free behaviour. Considering point 2, if this function exists, then there exists some network architecture capable of approximating it.

This means the problem should be possible to solve, given a learning-based method. The question is if the input latent space vector is consistent enough and if the network architecture is complex enough. As developers, it is difficult to ascertain the complexity or structure of the latent space. Given an inconsistent input latent space, it is not given that there exists an underlying function $f(x)$ that allows for approximating a control policy able to satisfy the wanted behaviour discussed in 4.4.7.

As stated previously, the IL problem is simpler and grants more guarantees in terms of performance. If the system is constrained by the underlying function being difficult to approximate, then it is more efficient to use the time and resources on the IL problem than the RL problem.

4.7 Results

Here we will present how well the agent performed with respect to the criteria introduced in the evaluation metric section 4.5.4. The agent will be evaluated in the test environment as described in section 4.5.2 about environments. We will first present how well the agent performs when trained only using IL. This agent will be referred to as the *IL trained agent*. Then we will evaluate the agent which receives pre-trained weights from the IL process and is further trained by the RL method PPO. This agent will be referred to as the *learning-based agent* The weights that the agent is initialised with in the RL training process have been extracted from the IL process long before the process would reach convergence. This is to avoid that the agent should be initialised to some local minimum that the RL training process is not able to escape from. The presented percentages are rounded to the nearest integer to allow for a comprehensible discussion.

4.7.1 Computational Complexity

We present the inference time for both the encoder and the agent, as well as the computational time for the graph-based local planner. The neural networks are executed multi-threaded, and the algorithms are evaluated on an Intel(R) Core(TM) i7-8700 CPU @ 3.20 GHz. The training environments were used for timing the controllers.

Process	**Time**
TSDF creation	32.06 [ms]
Encoder	48.85 [ms]
Agent	2.43 [ms]

Table 4.2: Inference time for the learning-based pipeline

Process	Time	
	mean [ms]	Std. Dev. [ms]
TSDF creation	32.06	
Path planning and control calculation	44.32	145.18

Table 4.3: Statistics for the computational time of the local planner

4.7.2 Imitation Learning

The two IL agents were trained until convergence and evaluated using the metric discussed in section 4.5.4. The results for the test environment can be seen in table 4.4. Here the MAV was able to reach the goal sphere for 148 runs, it had a total of 160 collisions and 92 timeouts. 128 of the 160 collisions had a neighbourhood around the MAV deemed to be represented in the latent space to a sufficient degree. On the other hand, 32 of the collisions had neighbourhoods that were poorly represented. The percentage of each of the categories are represented in the second row.

Agent 1			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
148	128	32	92
37 %	32 %	8 %	23 %

Table 4.4: Large network agent performance on the test environment.

In table 4.5 we can observe the performance of the large network agent trained by IL on the test environment. The total amount of runs are 400. Here, the number of goals reached was 169, the total number of collisions was 134, and the number of timeouts was 97. For the collisions, 102 of the 134 were labelled as having a sufficiently well represented neighborhood around the MAV, while 32 were labelled as insufficiently represented.

Agent 1			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
169	102	32	97
42 %	26 %	8 %	24 %

Table 4.5: Large network agent performance on the training environment.

Now we will present the results gathered from the agent with the smaller network. Table 4.6 shows the performance of the agent in the test environment. Here, 127 runs resulted in a reached goal, 184 resulted in collision and 89 of the runs timed out.

Agent 2			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
127	163	21	89
32 %	41 %	5 %	22 %

Table 4.6: Smaller network agent performance on the test environment.

Table 4.7 shows the performance of the small network in the training environment. Here 204 trajectories ended in a reached goal state, 148 ended in a collision and 49 timed out.

Agent 2			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
203	120	28	49
51 %	30 %	7 %	12 %

Table 4.7: Smaller network agent performance on the training environment.

4.7.3 Reinforcement Learning

Here we will present the results for the training with the RL method PPO. In table 4.8 we observe the performance of the learning-based agent initialised by the large network and further trained with the PPO algorithm. Here, 82 trajectories ended in “reach goal” and 153 trajectories collided with a sufficient representation of the neighbourhood of the MAV. Further, 38 runs led to collision with an insufficient representation of the neighbourhood while 127 timed out.

Agent 1 After PPO Training			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
82	153	38	127
20 %	38 %	10 %	32 %

Table 4.8: Large network learning-based agent performance on the test environment.

Table 4.9 shows the performance of the learning-based agent with a large network architecture trained with PPO on the training environment. This evaluation led to 118 trajectories ending in reached goal and 116 trajectories colliding with a sufficient representation of the neighbourhood of the MAV. 33 of the runs led to collision with an insufficient representation of the neighbourhood while 133 of the runs timed out.

Agent 1 After PPO Training			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
118	116	33	133
30 %	29 %	8 %	33 %

Table 4.9: Large network learning-based agent performance on the training environment.

Now we will present the results from the learning-based agent with a smaller architecture. In table 4.10 we observe the agent performance in the test environment. Here, 144 of the trajectories reached a goal while 152 of collided with a sufficient representation in the latent space, while 47 collided with an insufficient representation in the latent space. 57 of the evaluation runs timed out.

In table 4.11 the agent performance in the training environment is presented. 184 of the trajectories reached goal, 108 collided with sufficient representation in the latent space while 31 collided with insufficient representation. 77 of the trajectories timed out.

Agent 2 After PPO Test Environment			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
144	152	47	57
36 %	38 %	12 %	14 %

Table 4.10: Small network learning-based agent performance on the test environment.

Agent 2 After PPO Training Environment			
Reached Goals	Agent-Caused Collisions	Encoder-Caused Collisions	Timeouts
184	108	31	77
46 %	27 %	8 %	19 %

Table 4.11: Small network learning-based agent performance on the training environment.

4.8 Discussion

In this part we will discuss the performance of the agents as presented in chapter 4.7. The information in the presented tables are the statistics of the outcomes from the trajectories. These are useful to discern the general performance of different agents. To further contribute to the discussion, we will distinguish different traits of the MAV behaviour by illustrating trajectories where the trait is clearly present.

We begin by discussing the computational complexity of the whole pipeline.

4.8.1 Computational Complexity

The networks are run on a CPU on a desktop computer. Needless to say, this is a very computationally different environment to the mobile System-on-Chip (SoC) computer found on common MAVs. Since all systems on a MAV are heavily constrained by power consumption, the computer systems may often be run on CPU systems that are more lightweight and offer better performance per watt of energy. However, many SoCs that are suited for MAVs also integrate GPUs. If an algorithm can be run partially or fully on a GPU, it will free up computational resources for the CPU and thus run much more efficiently.

The Tensorflow software library [11] is highly optimised for running on GPUs. Since we have based all our neural network models on this framework, it is reasonable to expect that the inference times in table 4.2 can be further reduced if an SoC with an integrated Nvidia GPU is used.

The largest part of the inference time is used on the encoder, and this is the major bottleneck for the total inference time. However, even running this entire pipeline, from LiDAR point cloud inputs to control outputs, on a CPU can result in a high-level control frequency of 12 Hz. This is sufficient for a MAV navigating at slow to medium speeds through a cluttered environment. It is important to note that the inference time for the learning-based pipeline is constant. The computational complexity does not increase or decrease over time, or with the complexity of the scene.

Let us now look at table 4.3 which shows the computational time for the graph-based planner we use as expert and baseline. The time it takes to create the TSDF representation is the same, as it uses the same code base as in the learning-based pipeline. The most significant difference is the time it takes to calculate a path and a control action varies greatly. We see a standard deviation of 145.18 ms from the

mean of 44.32. This clearly shows how the computational complexity of this path planning algorithm varies greatly with time and the complexity of the immediate environment. The best-case scenario happens when the randomly generated goal point happens to be situated in a straight and collision-free path from where the MAV happens to be. In this case, the time the algorithm uses for creating a path is only about 2 ms as it is only checking for collisions along a straight line. However, whenever the path planning algorithm needs to sample a complex immediate environment to do graph-based path optimisation, the computational time often jumps to about 350 ms. Therefore, it is apparent that in the worst-case scenario, the graph-based path planner is only able to reach a frequency of about 2 Hz when including the time spent constructing the map of the immediate environment.

4.8.2 Imitation Learning

We will now discuss the general performance of the two agents by their performance as presented in table 4.4, 4.5, 4.6 and 4.6. This discussion will revolve around the ability of the agent to generalise and its performance with respect to collision avoidance.

The most crucial topic of interest is the substantial amount of collisions present in every evaluation. Considering the main topic of evaluation being collision avoidance, we can clearly deduct from this metric that none of these agents consistently provide collision avoidance. A 40 percent chance of colliding in the test environment is a clear indication of this fact. At the same time, there seem to be times where it shows signs of being environment-aware and thus demonstrates collision-free trajectories. These trajectories occur when the MAV loops around objects to reach a goal or attempt to suddenly change its trajectory when it is about to hit an object. These examples are motivating examples as they can indicate where potential future work should be focused.

From here, we will discuss the degree to which collision avoidance is present in the policy. It should be noted that both agents show some tendencies of being overfitted. This is observed in the amounts of trajectories labelled as *reached goal* in the training environment, table 4.5 and 4.7, being higher than the number of reached goals in the test environment, table 4.4 and 4.6. The same conclusion can be deducted from the significant reduction in collisions. Using this argument, it can be stated that agent 2 is more overfitted to the environment than agent 1. An important aspect of collision-free navigation is to ensure the generalisability of the agent. Overfitting is unwanted as it inhibits the ability of the agent to generalise.

A subset of the trajectories that reached the goal can be caused by the goal and MAV being spawned with a direct and collision-free path between them. These should not account for a large portion of the trajectories, but since the goals are generated randomly, one does not avoid the possibility of a straight path occurring.

The large network IL agent is most able to generalise between the two models. From the evaluation in the test environment, we can observe that it is about as likely to reach within 1 meter of the goal point as to collide. Reaching within 1 meter of the goal means travelling between 5 and 6 meters of collision-free trajectory. Such a long flight in environments as cluttered as presented here deduces that the agent is able to show some property of environment awareness, although not consistently.

It is interesting to note the proportional difference of collisions caused by the encoder when comparing performance in the training and test environments. The training set has a notably higher proportion of encoder caused collisions than the testing set. The encoder-caused collisions in the test environment for the first agent accounted for 20% of the total collisions. In the training environments,

it accounted for 23.9%. The difference is more apparent in agent 2. In the test environment, 11% of the collisions are due to encoder errors, while in the training environment, the same number is 18.9%. It is difficult to ascertain the reason for this. It might be that the overfitted agent has a more reckless behaviour to the point of reaching states that are harder to decode, but this is unlikely considering the decrease in the total amount of collisions and increase in reached goals. The most probable cause is that the agent is overfitted to the encoder, which does not necessarily return consistent outputs or any guarantee for performance. Any irregularity in the encoder output might have drastic implications in regard to the input sequence. This would mean instances of irregular outputs from the encoder can have more fatal outcomes for an overfitted agent.

The encoder was trained in the same training environments as evaluated here. This might be an indication that the encoder is able to generalise better than the agent. The autoencoder is trained with a *validation loss*, and this is probably as important to integrate in the agent training to monitor the generalisability of the policy. This might also be viewed as a generalisation bottle-neck for the pipeline as a whole. If the encoder is only able to generalise to a certain degree, then the ability of the agent to generalise will be constrained accordingly.

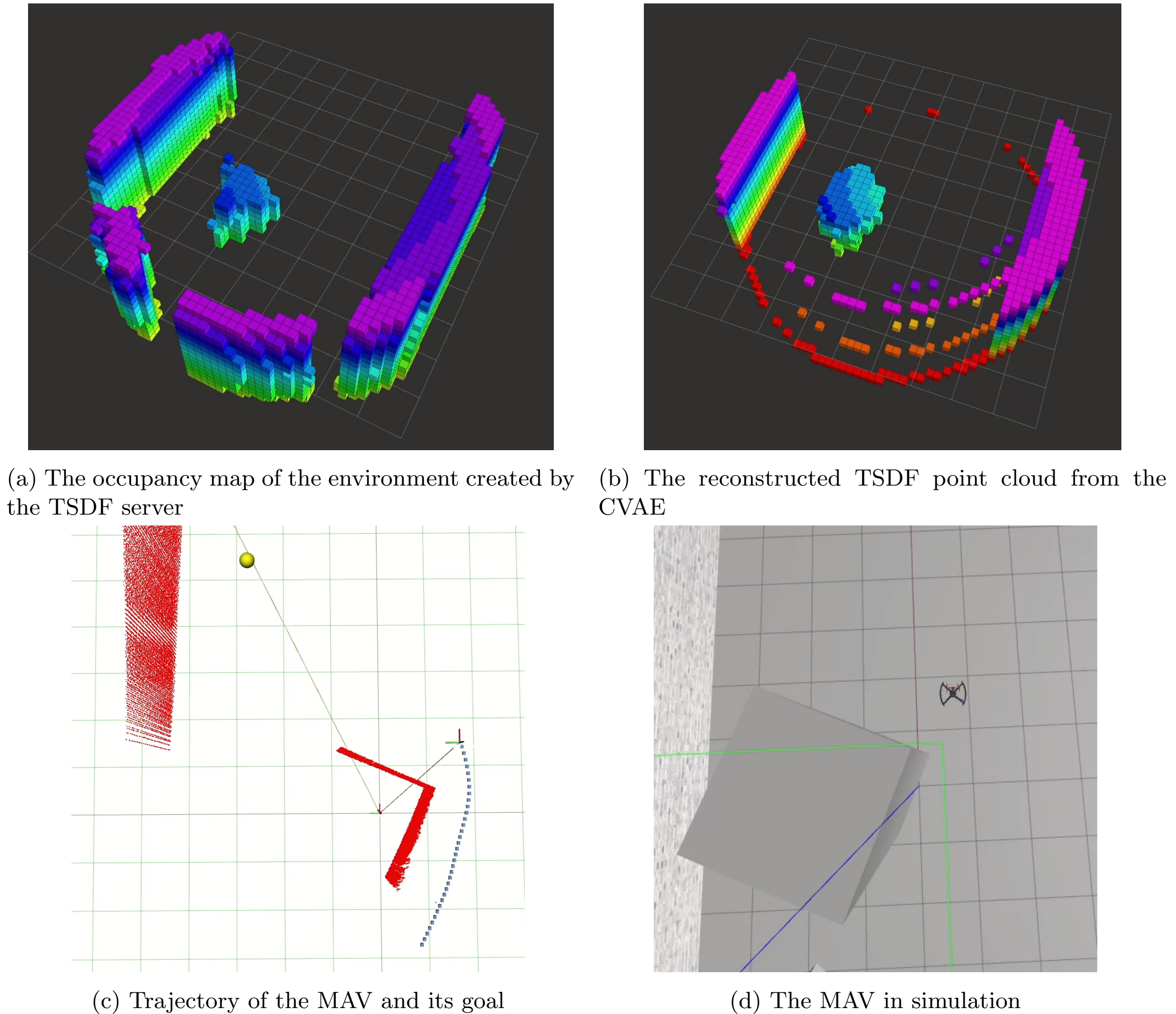
Exemplified Behaviour of the Imitation Learning Agent

Here we will discuss different types of performance that can be observed from the trained agent. To visualise these performances, we will use four different images as shown in figure 4.12. The first image, figure 4.12a, shows the output TSDF at a particular time step t, with the MAV in the centre. While the second image, figure 4.12b, shows the output TSDF decoded from the latent vector of the same time step t. These images have the same properties as the ones discussed in chapter 2. The third image, figure 4.12c, shows the trajectory of the MAV in the visualisation tool Rviz, observed from above. Here the MAV is represented as the green, red and blue axes. The yellow sphere is the goal point the robot is to reach, it is here visualised with a radius of 0.1 meters. Additionally, in this image we can observe the path as the grey dotted line from the spawn point to the MAV. The yellow line visualises the MAV position in relation to the origin of the world frame. The red dots are the simulated LiDAR points cast from the MAV onto the objects. These are the points that the local TSDF representation is constructed with, and are therefore important to understand the observability of the agent. It should be noted that the proportion of the axes of the body frame in figure 4.12c does not correspond to the actual sizes of the MAV in the simulation. The fourth image, figure 4.12d, shows the MAV in the simulated environment world in Gazebo as seen from above. In this image, the sizes and proportions are correct. The Gazebo image is therefore useful for understanding the scales of the drone and objects in the environment.

The performances can be categorised into three different classes of trajectories. The first class of trajectory will show the MAV when it manages to perform collision avoidance. The second will exemplify when a collision occurs even though the reconstructed TSDF is able to represent the object with sufficient fidelity. The third shows a collision occurring without the obstacle being represented correctly in the reconstructed local TSDF. It is important to note that these visualisations are hand-picked and discussed here as they are representative of general tendencies in the trajectories. These tendencies often do not occur in isolation. They are usually entwined within in parts of the input sequence which makes establishing a cause of success or failure difficult to ascertain. The examples visualised here are meant as illustrations of situations where these tendencies are easily distinguishable.

Trajectory Indicating Collision Avoidance Properties

Consider the trajectory visualised in figure 4.12. Here the local TSDF map in 4.12a is a correct representation of the environment observed in image 4.12c and 4.12d. The decoded version in 4.12b is able to reconstruct the incoming TSDF with sufficient fidelity. In image 4.12c the trajectory of the MAV is plotted. It is observed that it is successful in keeping a safe distance from the cylinder and the wall. This example illustrates the tendency when the agent expresses clear indications of being aware of the surrounding environment by controlling the drone in the desired manner.



(a) The occupancy map of the environment created by the TSDF server

(b) The reconstructed TSDF point cloud from the CVAE

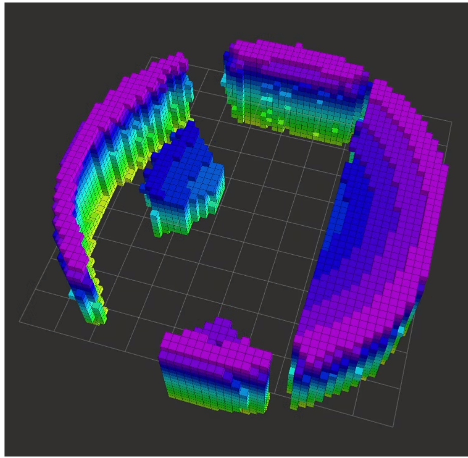
(c) Trajectory of the MAV and its goal

(d) The MAV in simulation

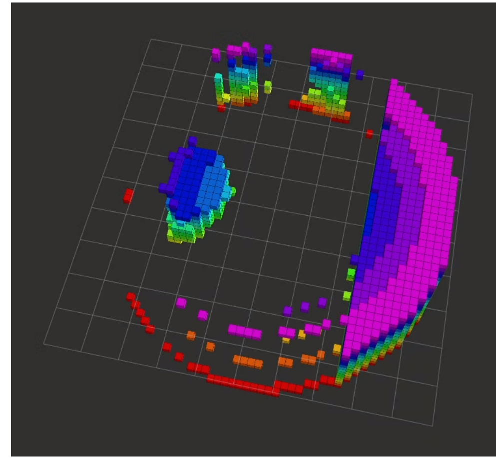
Figure 4.12: An example trajectory of the IL agent performing collision avoidance with the object well represented in the deconstructed latent space.

Another example of the IL agent executing the desired behaviour is observed in figure 4.13. The reconstruction in image 4.13b is deemed as representing the environment well, even though it misrepresents the object on the lower part of the image. For this particular trajectory, the object in the lower part is not critical for collision avoidance regarding the route between the MAV and goal spawn points. The wall on the right side and the cylinder on the left bounds the trajectory of the MAV. Here

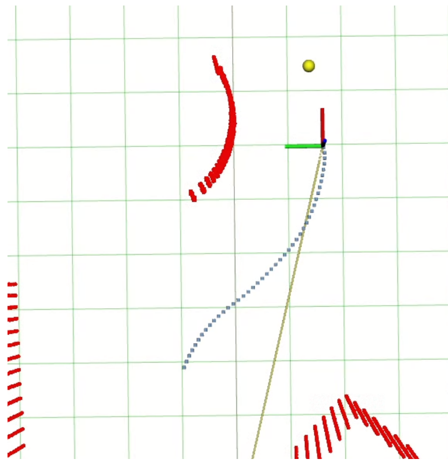
the MAV is able to reach the goal in a collision-free manner, while also having a safe distance to the closest object. This is an example of the desired behaviour of the agent.



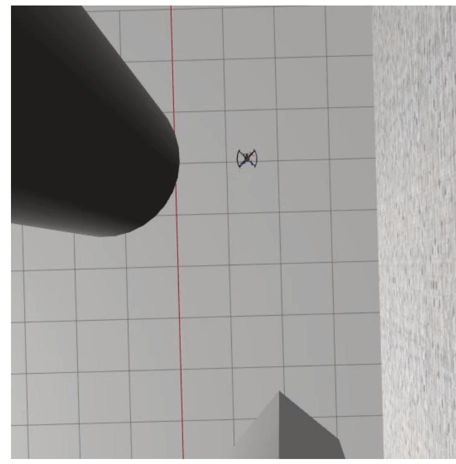
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed TSDF point cloud from the CVAE



(c) Trajectory of the MAV and its goal



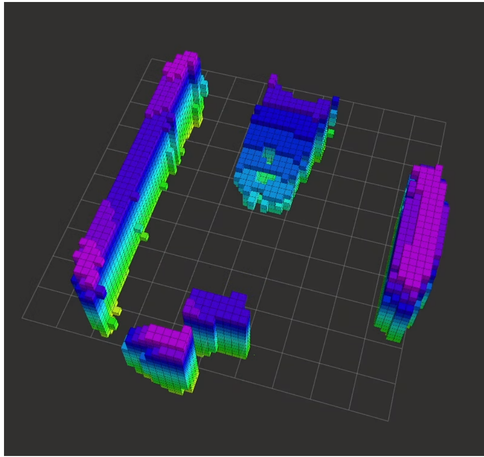
(d) The MAV observed from above in the simulation.

Figure 4.13: A trajectory by the IL agent exhibiting collision avoidance properties with good reconstruction

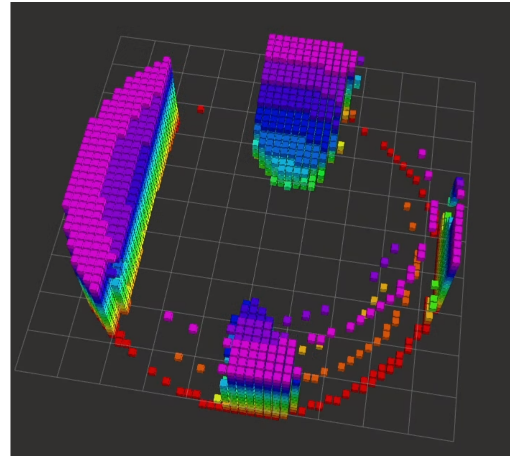
Collision Trajectory with Object Present in Latent Space

Now consider figure 4.14. Here the reconstructed TSDF in 4.14b, reconstructs the input TSDF in 4.14a, to an accurate degree. The reconstructed TSDF is able to represent the object with sufficient fidelity, meaning there is a decent indication that the object is represented in the corresponding latent vector. Nonetheless, the MAV moves directly into the object. From image 4.14c, it is apparent that the MAV is only a few time steps from colliding with the obstacle. It is interesting to note that the trajectory is a very straight line. Suppose we consider that this is a series of control inputs where each calculated input only observes the state of that particular time step. Then the straightness of the path shows a consecutive policy for the traversed state space. If the policy produced very different outputs during this particular sequence, we would observe some perturbations in the path. However,

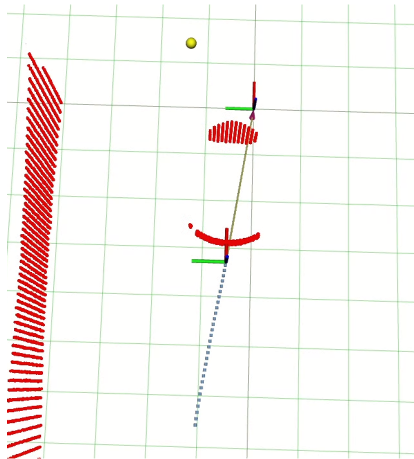
we observe none. Meaning one of two situations must be happening. The first possibility is that the agent does not decode the latent space vector correctly, making the object inobservable for all states. The reconstructed TSDF in image 4.14b is decoded using the decoder of the autoencoder pair. There is no guarantee that this corresponds to the way the agent decodes the latent space. On the other hand, due to the multi-modal issue in optimising the policy, it may be conditioned to aim for obstacles as discussed in section 4.6.1. This may happen if there is close to an equal chance of picking a path that goes left or right to avoid an obstacle. In either case, this MAV behaviour is highly undesired.



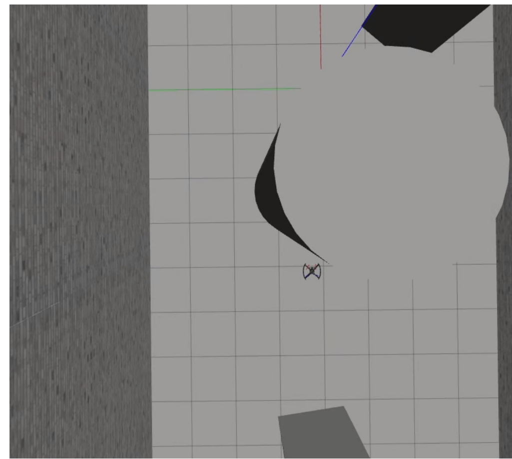
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed output TSDF point cloud from the CVAE



(c) Trajectory of the MAV and its goal. The MAV is the lower of the two axis frames.



(d) The MAV in simulation.

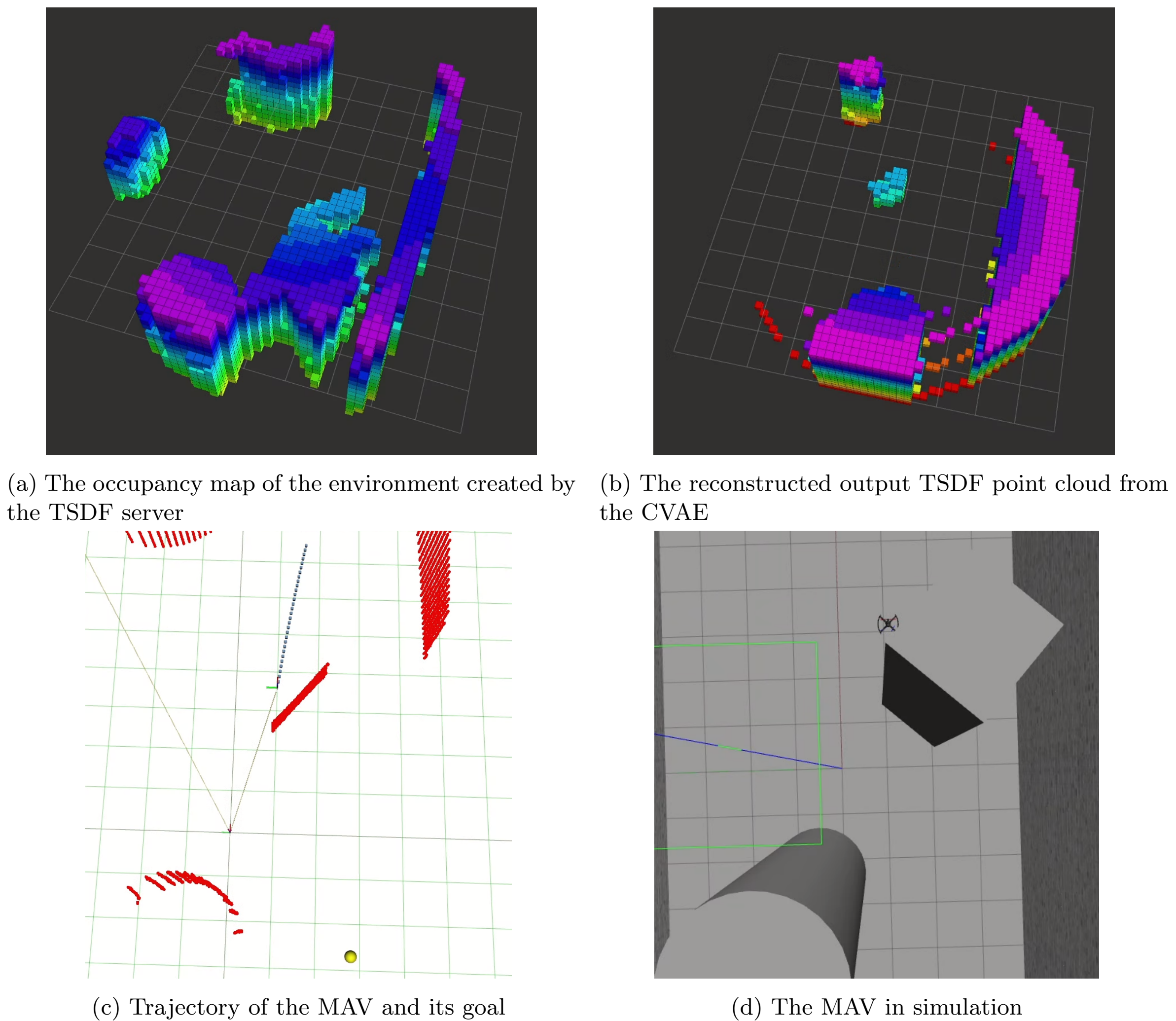
Figure 4.14: A colliding trajectory by the IL agent with good reconstruction

Collision Trajectory Without Object Correctly Represented in the Latent Space

In figure 4.15 the third class of agent performance can be observed. In this collision example there are severe differences in the original local TSDF in 4.15a and the reconstructed TSDF in 4.15b. There is also an error in the original TSDF map where the two obstacles visualised in 4.15d are merged together. This is caused by the ego-motion of the MAV and can be a source of errors for the encoder

and agent. The problem arises when an object has only been seen from one certain angle. Since the TSDF server assumes that unseen space is occupied, as the MAV approaches an object straight on, the object will "grow" in length. This is illustrated in figure 3.22. Practically this means that the corridor of free space between the cylinder and the rectangular box in figure 4.15d will be "filled in" by the TSDF server as seen in figure 4.15a. Only when the MAV can directly observe this free space will the TSDF point cloud clear this part of the TSDF to more closely resemble the real environment. This means that the agent must take control actions to see an object from several angles to better understand where it is going. However, since our implementation is *memoryless* and therefore assumes the Markov property, this behaviour is difficult to teach the agent. It will always make decisions only based on the current input TSDF point cloud, and it cannot distinguish whether this TSDF point cloud fully reflects the true environment.

In this example, there is little indication that the latent space contains sufficient information about the obstacle. Here it would seem that the latent space vector refers to an object being further away, to the bottom of the image. In image 4.15c the agent trajectory is close to a direct line with some indication of the path steering towards the goal point. It is not possible to distil the entire control sequence based on figure 4.15. However, it is important to note that a collision will only be a matter of chance if the obstacle is unobservable for the agent. The agent cannot be expected to perform collision avoidance in a situation where the latent space does not contain sufficient information about the environment. As trajectories similar to the one illustrated in this example occur during training as well, they will lead to data ambiguity and a poorer policy. This problematic behaviour is discussed further in the observability section 4.6.1.



(a) The occupancy map of the environment created by the TSDF server

(b) The reconstructed output TSDF point cloud from the CVAE

(c) Trajectory of the MAV and its goal

(d) The MAV in simulation

Figure 4.15: A colliding trajectory by the IL agent with erroneous reconstruction

4.8.3 Reinforcement Learning

Consider now the performance of the learning-based agent trained with RL as presented in the 4 4.8, 4.9, 4.10 and 4.11. It is evident that the large network performs much worse than any other model in the presented results. This is observed by the amount of reached goals having declined and the number of collisions having risen. The proportion of timeouts are substantially high which is evident of a slower trajectory, as is confirmed by visual inspection. The architecture of this agent has been shown to be able to perform better, as observed in the presentation of the results of the IL agent. We will therefore regard the large network learning-based agent as having converged to a sub-optimal policy. The computational resources learning-based methods demand are substantial, and achieving an optimal, or near-optimal, policy is not guaranteed. This is a large disadvantage of using methods such as IL and RL.

In IL, the agent is trained using the acceleration vector found by the expert. The system does not take into account whether the latent vector represents the TSDF accurately or not. When exploration

is performed in RL, the representation is crucial for predicting the input sequence. Given a faulty input vector, collision-free navigation is just a matter of chance. Let us now consider the exploration perspective of the RL agent. Exploration that results in small changes of the MAV position will be rewarded or not penalised too much, while exploration resulting in large changes have a significant chance of collision, generating a large penalty. This penalty of exploration might increase the possibility of achieving a slow and cautious behaviour of the agent as observed in the performance of the large agent.

The performance of the smaller network learning-based agent trained with RL is similar to the performance of the two learning-based agents trained with IL. It is difficult to decide which agent performs better. There are small differences between the agents' performance, but not enough to distinguish the methods used with certainty.

Combining two very different learning-based methods is a demanding task. The different setup of the optimisation problem as observed in the IL method and the RL method are described vastly different [78]. While each attempts to solve the same problem, they inherently will focus on differences. In RL, this difference in the optimisation problem can, to some extent, be tuned by altering it to resemble the implicit reward function of the expert. However, due to the difficulties of solving the IL problem, as described in section 4.6.2, there was not enough time to focus on ensuring consistency in the behaviour of the gradients when changing between the methods.

In regards to tuning the reward function, it is hard to state if there exists some form of optimal tuning of the reward function to ensure the behaviour discussed in 4.4.7. Increasing the goal reward could lead to straight paths being weighted more and therefore result in less collision avoidance properties. Weighing the penalty on the difference in position between the MAV and goal higher could lead to more reckless behaviour. In general this is a difficult input space without guarantees for the existence of an optimal policy, making learning-based approaches difficult.

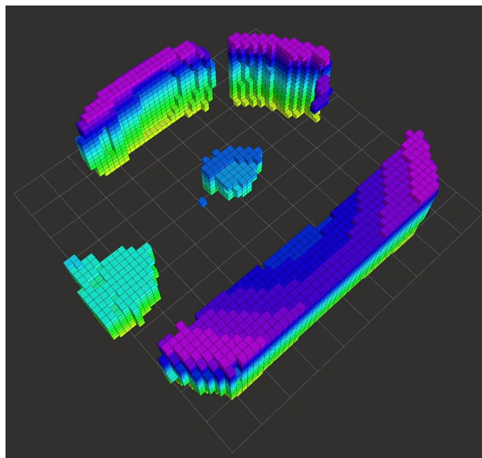
The explained issue here underlines the problem detailed in section 4.6.2. Here the discussion revolves around the existence a true underlying function describing the relationship between the unpredictable input space and the acceleration fulfilling the desired behaviour underlined in section 4.4.7. The parameter space of the agent is very complex, meaning it is not possible to rule out the existence of a combination of weights mapping input to output in an adequate way, but it is very difficult, to approximate this function.

Exemplified Behaviour of the Trained Learning-based Agent

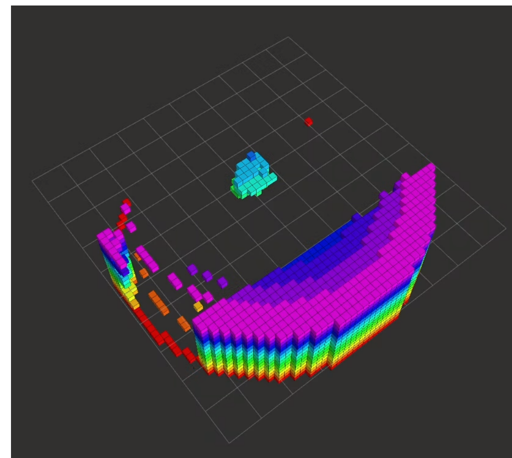
Here we will discuss trajectories that are expressive of the behaviour of the drone. Much of the same discussion from the results from the IL agent performance in section 4.8.2 will be relevant here as well. For this visualisation we have only used the small network.

Trajectory Exemplifying Collision Avoidance Properties

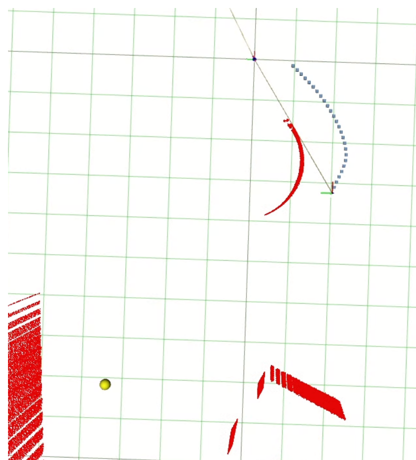
In figure 4.16 the learning-based agent trained with RL executes a trajectory where it manoeuvers efficiently through the environment. Figure 4.17 is also an example of the wanted behaviour of the MAV.



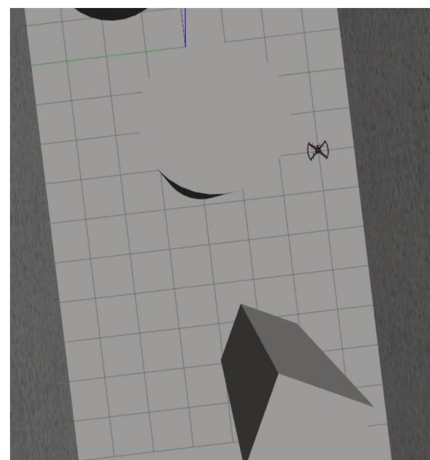
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed TSDF point cloud from the CVAE

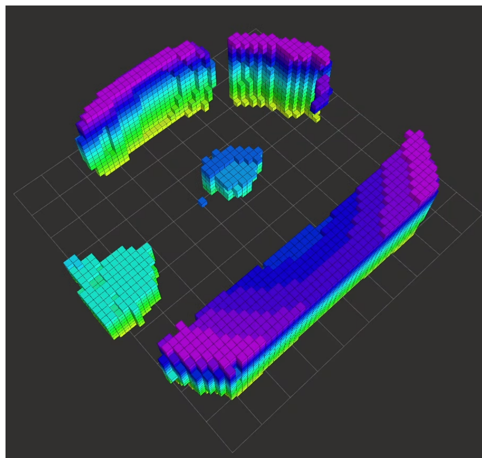


(c) Trajectory of the MAV and its goal

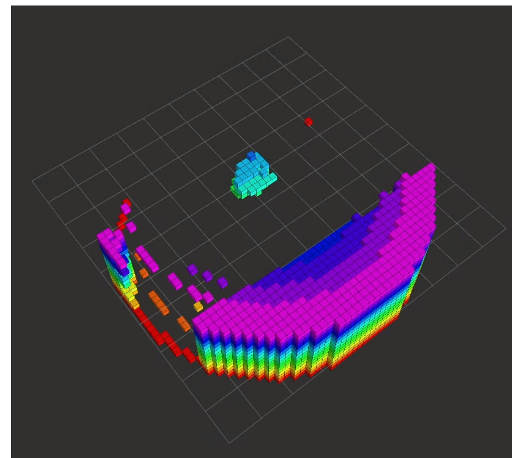


(d) The MAV in simulation

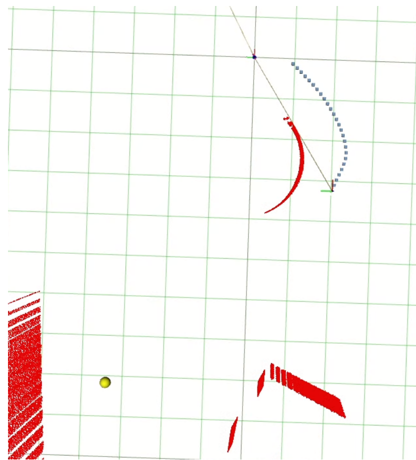
Figure 4.16: An example trajectory of the RL agent performing collision avoidance with the object in the deconstructed latent space.



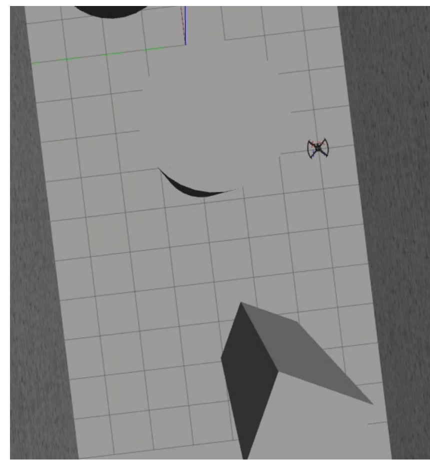
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed TSDF point cloud from the CVAE



(c) Trajectory of the MAV and its goal

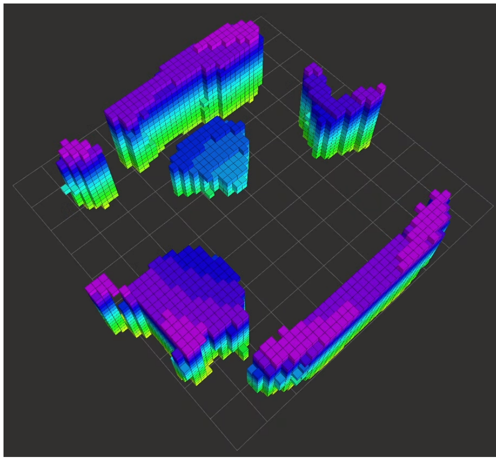


(d) The MAV in simulation

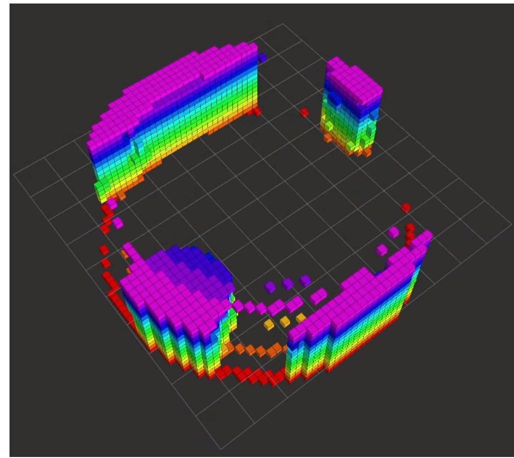
Figure 4.17: A trajectory by the RL agent exhibiting collision avoidance properties with good reconstruction

Trajectory Colliding with Object Missing in the Latent Space

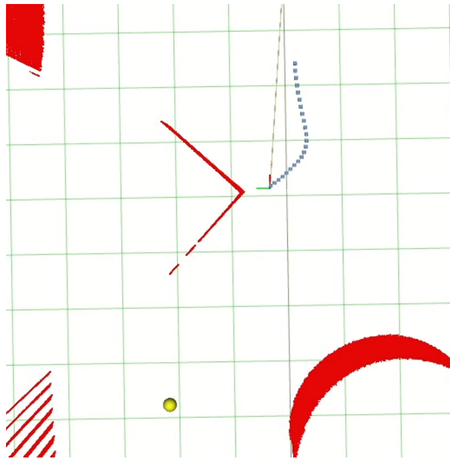
In figure 4.18 the agent collides with an object which is not present in the latent space at all. In this example, the agent cannot be expected to perform collision avoidance.



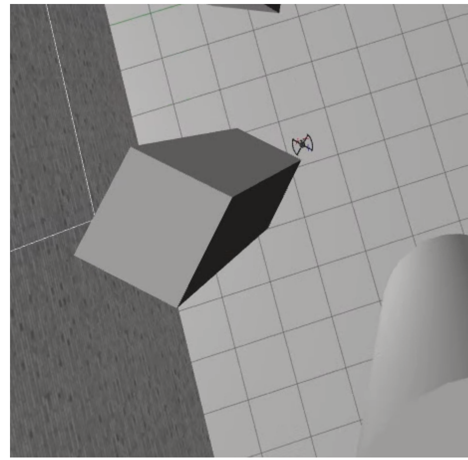
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed TSDF point cloud from the CVAE



(c) Trajectory of the MAV and its goal

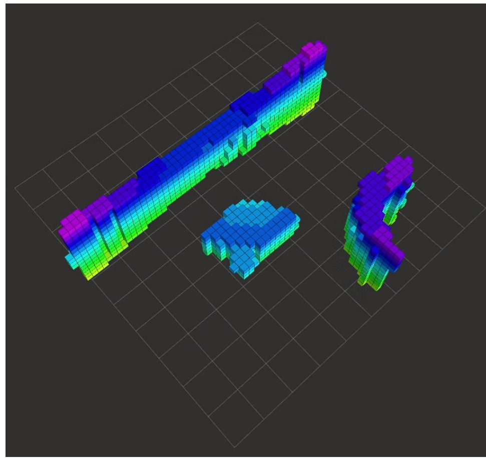


(d) The MAV in simulation

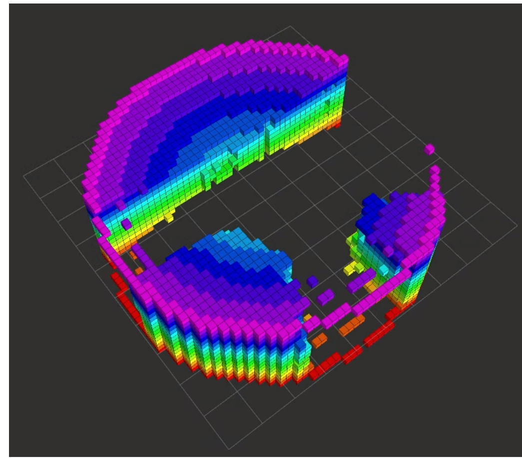
Figure 4.18: An example trajectory of the RL agent colliding with the environment. The MAV collides with an object which is present in the occupancy map from the TSDF server, observed in 4.18a, but is absent from the reconstructed TSDF point cloud from the CVAE, seen in 4.18b

Agent Colliding With Object Well-Represented in the Latent Space

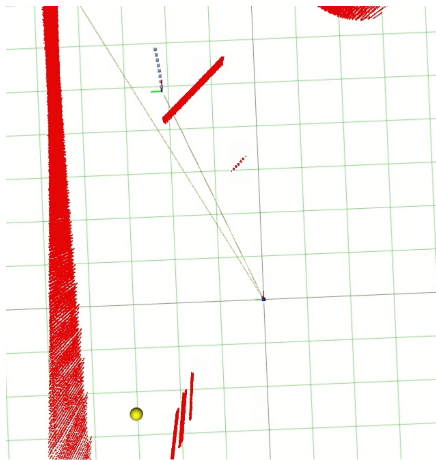
In figure 4.19 the object is present in the latent state representation, and the agent should therefore be expected to perform collision avoidance in this example. The MAV is unable to dodge the obstacle and collides.



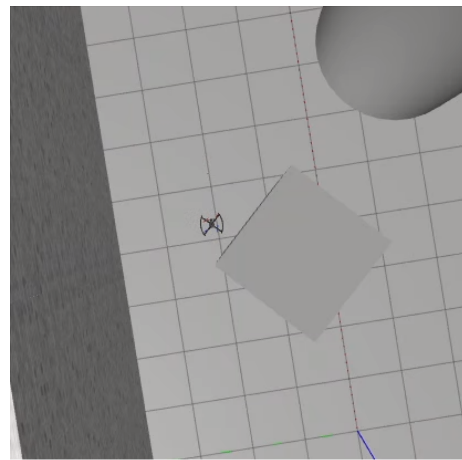
(a) The occupancy map of the environment created by the TSDF server



(b) The reconstructed TSDF point cloud from the CVAE



(c) Trajectory of the MAV and its goal



(d) The MAV in simulation

Figure 4.19: An example of a trajectory by the RL agent leading to a collision with the environment. The collision happens with an object that is clearly represented in the reconstructed TSDF point cloud as seen in 4.19b.

4.9 Conclusion

In this section, we have presented a learning-based agent which, together with the Convolutional Variational encoder, represents an End-to-end method of doing both local path planning and high-level flight control. The learning-based agent was trained both through an Imitation Learning and a Reinforcement Learning framework to teach it to navigate and control a MAV. The inputs to the learning-based agent are the latent space representation of the immediate 3D environment around the MAV and the relative position of the MAV with respect to a given setpoint and the velocity of the MAV. The output of this learning-based agent is an acceleration vector that is fed directly to the low-level control algorithms of the MAV.

A challenge with this approach is a total dependency on the consistency of the latent space representation from the encoder. This means that whenever the latent space fails to encode an object, the

learning-based agent has no way of sensing an impending collision. Even though the latent space is itself regularised through the process of training the CVAE, we have no guarantee as to how this latent space is organised internally or its complexity. The result of this is that making a function that maps from this latent space to a valid collision-free acceleration vector can only be done through exhaustive training of several different network topologies. Our work indicates that a learning-based agent can extract spatial information from the latent space representation of the local environment but naturally struggles when the latent space representation is not consistent or ambiguous.

We highlighted the challenges encountered through the Imitation learning training process when using a traditional path-planning algorithm as an expert in the Data Aggregation Algorithm. Since our learning-based agent is operating according to the Markov property, the system is completely memoryless. This means that the sequence of control outputs from the learning-based agent will not be optimised with respect to each other, only the information available in the current time step. The consequence of this becomes apparent if there exists an ambiguous solution to the optimal path planning problem. One example being if a symmetric object is situated directly between the MAV and the goal point as discussed in section 4.6.1. In this case, the path suggested by the expert path planner may oscillate between subsequent time steps. When these examples are taught to the learning-based agent, the resulting path it will learn may be the average of the paths between which the traditional planner is oscillating. Practically, this can result in the agent accelerating directly into an object if it is unsure if it should go left or right.

When we trained the learning-based agent using the Reinforcement Learning framework, we imported a model which had been previously trained for a short period of time using the Imitation Learning framework. This was to avoid having to teach the agent certain basic skills like stable flight without having to spend time teaching it through exploration using RL. However, an inevitable consequence of this method is that the agent will inherit certain non-optimal behaviours from the IL training process, as previously explained. Our work shows that the learning-based agent trained using the RL framework is difficult and guarantees no optimal, or close to optimal, policy.

Our algorithm runs at a constant 12 Hz regardless of the environment complexity, which contrasts with the traditional sampling-based local path planner used as the baseline. This is significant because it will allow a MAV to navigate at a constant rate without needing to slow down because of an optimisation problem growing in complexity with its surrounding environment.

Chapter 5

Conclusions, Discussion, and Recommendations for Further Work

5.1 Summary

Chapter 2 elaborated and defined the fundamental theory for the following chapters. This theory consisted mostly of machine learning methods. This includes a discussion about the training process of the neural network, which will be the main workhorse throughout this thesis. In addition, the effect the architecture of a neural network has on its optimisation landscape and therefore its tendencies for over- or underfitting on the training data, was also discussed.

Chapter 3 presented a method for heavily compressing point clouds representations of the local environment of the MAV to reduce its dimensionality. This compression was done while retaining geometric information to make it more computationally tractable for the controller in the following chapter. The theory behind the generation of the TSDF point cloud and the CVAE was presented together with an exhaustive search of different network architectures. We showed that it is possible to make a compressed latent space representation of the TSDF point clouds gathered in a given environment and discussed how well the CVAE is able to generalise to unseen environments. The selected network architecture was pushed as far as possible to highlight the challenges posed by compressing point clouds using a CVAE. Then the practicalities regarding the generation of TSDF point clouds, all the way from the LiDAR, up until the passing of the TSDF to the encoder, was discussed with an emphasis on the effects of parameters in the implementation. The CVAE is able to learn environments with a limited subset of types of obstacles well, but the challenge is to have it generalise properly to unseen or more complex environments. There is also no guarantee as to how well the latent space is structured, other than it being regularised to resemble a normal distribution.

Chapter 4 discussed learning-based methods for generating an agent, parametrised as a neural network, capable of executing collision-free flight based on the latent space representation from the CVAE of chapter 3. This represents an End-to-end approach to solving both the local path planning and high-level control problem with the same model, which contrasts with the traditional divide between the two disciplines. The chapter's objective was to train the policy of such an agent using both Imitation Learning (IL) and Reinforcement Learning (RL), and evaluate its effectiveness. Theory behind

the methods used were elaborated and discussed. The two proposed neural network architectures were trained both through the IL framework and the combined IL and RL framework. Each used the best performing encoder network architecture from chapter 3.

5.2 Conclusion

In this master thesis, we have presented a novel way to do learning-based local path planning and high-level flight control. Using 3D Convolutional Variational Autoencoders, a regularised latent space representation is created from the immediate environment. Our work indicates that the agent is able to extract information about its position relative to the objects surrounding it. This lets the agent infer a geometric understanding of its immediate environment, and from there, be able to some degree execute a collision-free trajectory. We demonstrated in chapter 3 how well a CVAE is able to compress the TSDF point clouds under a given set of assumptions. The biggest challenge was to generalise from the training environment in which the TSDF point clouds were collected to unseen environments. We reached the limit to how well the proposed autoencoder architecture can generalise and used this as a hard upper limit for the later sections of this thesis. The training of the learning-based agents, as explained in chapter 4, resulted in agents able to infer a of geometric understanding of the robot's immediate environment from the compressed latent space. This pipeline represents an End-to-end method that combines both traditional local path planning and high-level flight control. From the raw point cloud inputs from the LiDAR to the control outputs of the low-level MAV controllers, the whole pipeline is able to run at a constant frequency of approximately 12 Hz. This is a very interesting property, as it means that the run time of our algorithm does not increase or decrease over time or with the complexity of the immediate environment. However, significant work still remains on both the perception and the control part of this implementation for it to consistently provide collision-free trajectories with comparable performance to the graph-based path planner we use as the baseline.

5.3 Discussion and Recommendations for Further Work

This implementation of local path planning has shown promising results, and it is the authors' impression that this implementation is worth further research. Being a completely novel approach, there still remains work for this method to be competitive. This method is not competitive in terms of consistently solving the collision avoidance problem compared to the graph-based local path planner we use as the baseline. We will now discuss what we believe to be the most important challenges with this implementation, and what we believe will help solve them.

5.3.1 Ensure Generalisability of the Encoder

One of the biggest challenges of this implementation is ensuring that the latent space of the autoencoder is consistent and generalised. This means that if the autoencoder observes something it has not seen before, it will be able to infer a corresponding latent space that will be consistent with respect to the types of environments it has seen before. Right now, we see that we are pushing the limits to how well the autoencoder is able to generalise. If we train the autoencoder directly in the environment in which the agent will be trained later, we get a consistent latent space at the cost of heavy overfitting. We

tried to circumvent this by making a well-distributed dataset with which the autoencoder is trained. However, we observe that we still are not reaching the needed level of generalisability for the agent to be able to consistently make optimal decisions based on the latent space. This means that, already before the imitation learning process is initiated, the input latent space is already non-optimal.

If one were to guarantee an optimal and consistent latent space, one would need to manually label the training data and the objects therein. This would enable the training process to monitor which types of objects get reconstructed correctly, while also optimising the recall of obstacles in the reconstruction. This would enable the algorithm to be optimised for maximum recall of objects, instead of recall of voxels. In the latter case, a small and thin object will be prioritised less than a massive object, as the massive object represents a larger proportion of the overall voxels. This is the case for our thesis. Ideally, failing to reconstruct thin objects should be penalised just as much as objects with comparatively larger sizes. If one could know in advance what types of combinations of obstacles would be reconstructed correctly, the agent could be trained in a constructed world where one had the guarantee of the latent space being consistent. The obvious drawback to this solution is the fact that the needed labelling would represent a huge amount of work to be useful. If one only had a small dataset, the training of the agent itself wouldn't generalise, but only overfit on this small dataset. It is therefore apparent that the size of this labelled dataset would need to be immense for it to have any academic value.

5.3.2 Memory of Objects Disappearing From Direct View

As discussed in chapter 3, one of the limiting factors for true 3-dimensional flight of the MAV is the limited observability due to the constrained field of view of the LiDAR, as well as the process of creating the local map. This results in the set of valid directions of travel for the MAV being strictly constrained to the field of view of the LiDAR. To remedy this, one should implement a LiDAR with a much larger field of view to increase the amount of directly observable space around the MAV. The coverage of the LiDAR cannot be total due to the presence of the MAV hardware. Thus we need an integrating approach that ensures that objects that disappear from direct view are remembered. This can be solved by ensuring that the TSDF generation process does not discard points that belong to objects that escape the directly observable region, or by working with a sliding window of TSDF point cloud observations. Assuming a static environment, this could make sure that the MAV would have a much wider range of valid directions to travel.

5.3.3 Challenges with Respect to the Imitation Learning and the Reinforcement learning Problems

Balancing a learning-based method on top of input data that can be unpredictable and without guaranteed performance is a difficult task. Cases where the latent space is inconsistent can lead to situations where a proposed action from the expert leads the MAV to what appears to be obstacles from the perspective of the agent. This problem is elaborated on in chapter 4.6. Ensuring that the agent is only trained on valid data points could be a way of completely decoupling the performance of the autoencoder from the ability of the agent to understand the latent space, but removing all data points where the encoder is inconsistent would make the agent reliant on perfect input data. When playing the finished agent, the encoder would still be imperfect, and the agent would probably be less able to

handle the new input space. Further efforts into this implementation should focus on making a reliable and consistent latent space representation.

5.3.4 Validation Loss for the Agent

The implementation of validation loss could be a useful tool to ensure that the agent does not overfit to the training environment. Generalisation is an essential aspect of the system, and implementing validation loss is a simple method to ensure the best performance from the agent. This could be implemented by observing the loss of the agent at given intervals during the training process in an environment in which the agent has not seen before.

5.3.5 Ambiguous Learning Samples From the Expert Planner

The multiple head agent, discussed in chapter 4.6.1, may be implemented as it could solve the issue of the agent being optimised to collide directly into objects if there exist any ambiguities with respect to which side of an object the agent should navigate. This would greatly improve the learning process by removing examples of behaviour which directly violated the hard constraints on collision avoidance. By optimising a set of neural networks which corresponded to the direction of travel in two separate half-planes in the x-y plane of the MAV, the outputs of these networks could be ranked so that the agent took a definite choice as to which direction the MAV should proceed.

Since our implementation is completely memoryless, the choice of action in a given timestep is totally independent of both the preceding and subsequent actions. If the learning-based agent were to include some memory capacity, it could be trained to ensure that a sequence of actions is consistent and non-oscillatory. It may also have understood the oscillatory nature of the expert planner better, recognising that deciding confidently to go a certain direction would reduce the ambiguities from the expert planner and result in a more consistent path.

5.3.6 Randomising the Environment

Increasing the generalisability of the agent requires data from a wide array of different environments. To ensure a more general policy, the environment should be shuffled randomly at regular intervals. These shuffles should be random to ensure no bias from the developers will affect the dataset, but should have some underlying structure guaranteeing that objects will be placed between most of the points in which the MAV and the goal spawn. This is to ensure that collision avoidance trajectory samples dominate the data set. There necessarily will be a limit as to how complex an environment we are able to generalise the model to recognise. Therefore, it is apparent that the scope of the *generalisability* of our models is constrained to a certain class of environment with a certain class of objects. Navigating inside a large cave will necessarily require a very different model than navigating in an office environment. Therefore, generalisability, in this case, refers to the model's ability to operate with environments in which the placement of obstacles is completely random.

Appendix A

Acronyms

LiDAR Laser imaging, detection, and ranging

MDP Markov Decision Process

ANN Artificial Neural Network

CNN Convolutional Neural Network

VAE Variational Autoencoder

CVAE Convolutional Variational Autoencoder

RL Reinforcement Learning

IL Imitation learning

DAGGER The Data Aggregation algorithm

TRPO Trust region policy optimization

PPO Proximal policy optimization

GPS Global Positioning System

LiDAR Light detection and ranging

MPC Model predictive control

NMPC Nonlinear model predictive control

SDF Signed Distance Field

TSDF Truncated Signed Distance Field

KL-divergence Kullback-Leibler divergence

MAV Micro Aerial Vehicle

UAV Unmanned Aerial Vehicle

RGB Red, Green, Blue

LQR Linear-Quadratic Regulator

A.1 Algorithms

Algorithm 1 One-step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

1: Input: a differentiable policy parametrization $\pi(a|s,\theta)$
2: Input: a differentiable state-value function parametrization $\hat{v}(s,w)$
3: Parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
4: Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0)
5: **loop** forever (for each episode):
6: Initialize S (first state of episode)
7: $I \leftarrow 1$
8: **loop** while S is not terminal (for each time step):
9: $A \sim \pi(\cdot|S,\theta)$
10: Take action A, observe S',R
11: $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$ (if S' is terminal, then $\hat{v}(S',w) \doteq 0$)
12: $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S,w)$
13: $\theta \leftarrow \alpha^\theta I \delta \nabla ln\big(\pi(A|S,\theta)\big)$
14: $I \leftarrow \gamma I$
15: $S \leftarrow S'$

Algorithm 2 Trust-region Policy Optimization[80]

1: Input: initial policy parameters θ_0 and initial value function parameters ϕ_0
2: Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
3: **for** k = 0,1,2,... **do**
4: Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment
5: Compute rewards-to-go $\hat{R}_t$
6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{π_k}
7: Estimate policy gradient as: $\hat{g}_k = \frac{1}{|D_k|} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$
8: Use the conjugate gradient algorithm to compute $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k$, where $\hat{H}_k$ is the Hessian of the sample average KL-divergence
9: Update the policy by backtracking line search with: $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k$, where $j \in \{0,1,2,...,K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint
10: Fit value function by regression on mean-squared error: $\phi_{k+1} = \arg\min_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \big(V_\phi(s_t) - \hat{R}_t\big)^2$, typically via some gradient descent algorithm

Algorithm 3 Proximal Policy Optimization, Actor-Critic style[81]

```
1: for iteration = 1,2,... do
2:     for iteration = 1,2,...,N do
3:         Run policy π_old in environment for T timesteps
4:         Compute advantage estimates Â_t, ..., Â_T
5:     Optimize surrogate L w.r.t θ, with K epochs and minibatch size M ≤ NT
6:     θ_old ← θ
```

Algorithm 4 The Rapidly-Exploring Random Graph algorithm [85]

```
1: V ← {x_init}; E ← ∅
2: for i=1,...,n do
3:     x_rand ← SampleFree_i
4:     x_nearest ← Nearest(G = (V, E), x_rand)
5:     x_new ← Steer(x_nearest, x_rand)
6:     if ObstacleFree(x_nearest, x_new) then
7:         X_near ← Near(G = (V, E), x_new, min{γ_RRG(log(card(V))/card(V))^{1/d}, η})
8:         V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new), (x_new, x_nearest)}
9:         for all x_near ∈ X_near do
10:            if CollisionFree(x_near, x_new) then
11:                E ← E ∪ {(x_near, x_new), (x_new, x_near)}
   return G = (V, E)
```

Algorithm 5 The DAgger algorithm [86]

```
1: Initialize D ← ∅
2: Initialize π̂_1 to any policy in Π
3: for i=1 to N do
4:     Let π_i = β_i + (1 − β_i)π̂_i.
5:     Sample T-step trajectories using π_i
6:     Get dataset D_i = {(s, π*(s))} of visited states by π_i and actions given by expert
7:     Aggregate datasets: D ← D ∪ D_i.
8:     Train classifier π̂_{i+1} on D
9: return best π̂_i on validation
```

A.2 Specifications

Vertical resolution	64 channels
Horizontal resolution	512
Range	120
Vertical field of view	45 °
Precision	± 0.7-5cm
Points per second	1,310,720
Rotation rate	10 Hz

Table A.1: The technical specifications for the Ouster OS1-64 LiDAR [61]

Bibliography

[1] *Department of Engineering Cybernetics*. URL: https://www.ntnu.edu/itk.

[2] Eilef Olsen Osvik. *Reward shaping in Quadrotor Control Using Deep Deterministic Policy Gradients*. 2020.

[3] Martin Aalby Svalesen. *Robustness in Deep Reinforcement Learning for quadrotor control*. 2020.

[4] *Autonomous robots lab*. URL: https://www.autonomousrobotslab.com/.

[5] *DARPA Subterranean challenge*. URL: https://www.subtchallenge.com/.

[6] Martin Aalby Svalesen. *Pointcloud Convolutional Variational Autoencoder*. 2021. URL: https://github.com/martiasv/Pointcloud_Convolutional_VAE.

[7] Tung Dang, Marco Tranzatto, Shehryar Khattak, Frank Mascarich, Kostas Alexis, and Marco Hutter. "Graph-based subterranean exploration path planning using aerial and legged robots". In: *Journal of Field Robotics* 37.8 (2020), pp. 1363–1388.

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

[9] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. "Robot Operating System (ROS): The Complete Reference (Volume 1)". In: ed. by Anis Koubaa. Cham: Springer International Publishing, 2016. Chap. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. URL: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[10] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[11] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[13] Hui-Zhong Zhuang, Shu-Xin Du, and Tie-Jun Wu. "Real-time path planning for mobile robots". In: *2005 International Conference on Machine Learning and Cybernetics*. Vol. 1. IEEE. 2005, pp. 526–531.

[14] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The international journal of robotics research* 30.7 (2011), pp. 846–894.

[15] Sikang Liu. "Motion planning for micro aerial vehicles". In: (2018).

[16] Hugh Cover, Sanjiban Choudhury, Sebastian Scherer, and Sanjiv Singh. "Sparse tangential network (SPARTAN): Motion planning for micro aerial vehicles". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 2820–2825.

[17] M. Dharmadhikari, T. Dang, L. Solanka, J. Loje, H. Nguyen, N. Khedekar, and K. Alexis. "Motion Primitives-based Path Planning for Fast and Agile Exploration using Aerial Robots". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 179–185. DOI: 10.1109/ICRA40945.2020.9196964.

[18] R. Reinhart, T. Dang, E. Hand, C. Papachristos, and K. Alexis. "Learning-based Path Planning for Autonomous Exploration of Subterranean Environments". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1215–1221. DOI: 10.1109/ICRA40945.2020.9196662.

[19] Helen Oleynikova, Michael Burri, Zachary Taylor, Juan Nieto, Roland Siegwart, and Enric Galceran. "Continuous-time trajectory optimization for online UAV replanning". In: *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2016, pp. 5332–5339.

[20] Sikang Liu, Michael Watterson, Kartik Mohta, Ke Sun, Subhrajit Bhattacharya, Camillo J Taylor, and Vijay Kumar. "Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-d complex environments". In: *IEEE Robotics and Automation Letters* 2.3 (2017), pp. 1688–1695.

[21] Brian MacAllister, Jonathan Butzke, Alex Kushleyev, Harsh Pandey, and Maxim Likhachev. "Path planning for non-circular micro aerial vehicles in constrained environments". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 3933–3940.

[22] B. Lindqvist, S. S. Mansouri, A. Agha-mohammadi, and G. Nikolakopoulos. "Nonlinear MPC for Collision Avoidance and Control of UAVs With Dynamic Obstacles". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6001–6008. DOI: 10.1109/LRA.2020.3010730.

[23] Sina Sharif Mansouri, Christoforos Kanellakis, Emil Fresk, Bjorn Lindqvist, Dariusz Kominiak, Anton Koval, Pantelis Sopasakis, and George Nikolakopoulos. *Subterranean MAV Navigation based on Nonlinear MPC with Collision Avoidance Constraints*. 2020. arXiv: 2006.04227 [cs.RO].

[24] S. S. Mansouri, C. Kanellakis, B. Lindqvist, F. Pourkamali-Anaraki, A. -a. Agha-mohammadi, J. Burdick, and G. Nikolakopoulos. "A Unified NMPC Scheme for MAVs Navigation With 3D Collision Avoidance Under Position Uncertainty". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 5740–5747. DOI: 10.1109/LRA.2020.3010485.

[25] Tao Jia, Shaohuan Han, Ping Wang, Wenyuan Zhang, and Yawu Chang. "Dynamic obstacle avoidance path planning for UAV". In: *2020 3rd International Conference on Unmanned Systems (ICUS)*. 2020, pp. 814–818. DOI: 10.1109/ICUS50048.2020.9274865.

[26] Julien Margraff, Joanny Stéphant, and Ouiddad Labbani-Igbida. "UAV 3D path and motion planning in unknown dynamic environments". In: *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2020, pp. 77–84. DOI: 10.1109/ICUAS48674.2020.9214057.

[27] Shikai Shao, Yu Peng, Chenglong He, and Yun Du. "Efficient path planning for UAV formation via comprehensively improved particle swarm optimization". In: *ISA transactions* 97 (2020), pp. 415–430.

[28] Hamidreza Heidari and Martin Saska. "Collision-free trajectory planning of multi-rotor UAVs in a wind condition based on modified potential field". In: *Mechanism and Machine Theory* 156 (2021), p. 104140.

[29] Matthew Collins and Nathan Michael. "Efficient Planning for High-Speed MAV Flight in Unknown Environments Using Online Sparse Topological Graphs". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 11450–11456.

[30] *Ouster OS1-64 Lightweight LiDAR*. URL: https://ouster.com/products/os1-lidar-sensor/.

[31] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. "Voxblox: Incremental 3D Euclidean Signed Distance Fields for on-board MAV planning". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 1366–1373. DOI: 10.1109/IROS.2017.8202315.

[32] Hyeong Ryeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. "Rviz: a toolkit for real domain data visualization". In: *Telecommunication Systems* 60.2 (2015), pp. 337–345.

[33] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, Sept. 2004, pp. 2149–2154.

[34] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[35] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[36] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[37] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[38] C Van Der Malsburg. "Frank Rosenblatt: principles of neurodynamics: perceptrons and the theory of brain mechanisms". In: *Brain theory*. Springer, 1986, pp. 245–248.

[39] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

[40] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[41] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: *Master’s Thesis (in Finnish), Univ. Helsinki* (1970), pp. 6–7.

[42] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. *Visualizing the Loss Landscape of Neural Nets*. 2018. arXiv: 1712.09913 [cs.LG].

[43] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[44] Davide Scaramuzza and Zichao Zhang. “Visual-inertial odometry of aerial robots”. In: *arXiv preprint arXiv:1906.03289* (2019).

[45] Ji Zhang and Sanjiv Singh. “LOAM: Lidar Odometry and Mapping in Real-time.” In: *Robotics: Science and Systems*. Vol. 2. 9. 2014.

[46] Shaojie Shen, Yash Mulgaonkar, Nathan Michael, and Vijay Kumar. “Multi-sensor fusion for robust autonomous flight in indoor and outdoor environments with a rotorcraft MAV”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 4974–4981.

[47] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. “OctoMap: An efficient probabilistic 3D mapping framework based on octrees”. In: *Autonomous robots* 34.3 (2013), pp. 189–206.

[48] Ken Museth. “VDB: High-resolution sparse volumes with dynamic topology”. In: *ACM transactions on graphics (TOG)* 32.3 (2013), pp. 1–22.

[49] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. “Voxblox: Incremental 3D Euclidean Signed Distance Fields for on-board MAV planning”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 1366–1373. DOI: 10.1109/IROS.2017.8202315.

[50] Michel Breyer, Jen Jen Chung, Lionel Ott, Roland Siegwart, and Juan Nieto. “Volumetric Grasping Network: Real-time 6 DOF Grasp Detection in Clutter”. In: *arXiv preprint arXiv:2101.01132* (2020).

[51] Roland Pugliese, Thomas Konrad, René Zweigel, and Dirk Abel. “Object Detection and Potential Field-based Trajectory Planning on LiDAR 3D-Point Clouds for UAV in Uncertain Environments”. In: *Proceedings of the 32nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2019)*. 2019, pp. 2622–2636.

[52] Daniel Ricao Canelhas, Erik Schaffernicht, Todor Stoyanov, Achim J Lilienthal, and Andrew J Davison. “Compressed voxel-based mapping using unsupervised learning”. In: *Robotics* 6.3 (2017), p. 15.

[53] Sina Sharif Mansouri, Christoforos Kanellakis, Björn Lindqvist, Farhad Pourkamali-Anaraki, Ali-Akbar Agha-Mohammadi, Joel Burdick, and George Nikolakopoulos. “A unified nmpc scheme for mavs navigation with 3d collision avoidance under position uncertainty”. In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 5740–5747.

[54] C. Tu, E. Takeuchi, A. Carballo, and K. Takeda. "Point Cloud Compression for 3D LiDAR Sensor using Recurrent Neural Network with Residual Blocks". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 3274–3280. DOI: 10.1109/ICRA.2019.8794264.

[55] Kal Backman, Dana Kulić, and Hoam Chung. *Learning to Assist Drone Landings*. 2021. arXiv: 2011.13146 [cs.RO].

[56] Mark Pfeiffer, Michael Schaeuble, Juan Nieto, Roland Siegwart, and Cesar Cadena. "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots". In: *2017 ieee international conference on robotics and automation (icra)*. IEEE. 2017, pp. 1527–1533.

[57] Juncheng Liu, Steven Mills, and Brendan McCane. "Variational Autoencoder for 3D Voxel Compression". In: *2020 35th International Conference on Image and Vision Computing New Zealand (IVCNZ)*. IEEE. 2020, pp. 1–6.

[58] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. "Dynamic Graph CNN for Learning on Point Clouds". In: *ACM Trans. Graph.* 38.5 (Oct. 2019). ISSN: 0730-0301. DOI: 10.1145/3326362. URL: https://doi.org/10.1145/3326362.

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

[60] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

[61] Ouster. *High-resolution OS1 lidar sensor: robotics, trucking, mapping*. https://ouster.com/products/os1-lidar-sensor/. [Online; accessed 5-May-2021]. 2021.

[62] K. Alexis, G. Nikolakopoulos, and A. Tzes. "On Trajectory Tracking Model Predictive Control of an Unmanned Quadrotor Helicopter Subject to Aerodynamic Disturbances". In: *Asian Journal of Control* 16.1 (2014), pp. 209–224. DOI: https://doi.org/10.1002/asjc.587. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/asjc.587. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/asjc.587.

[63] M Islam, M Okasha, and M M Idres. "Dynamics and control of quadcopter using linear model predictive control approach". In: *IOP Conference Series: Materials Science and Engineering* 270 (Dec. 2017), p. 012007. DOI: 10.1088/1757-899x/270/1/012007. URL: https://doi.org/10.1088/1757-899x/270/1/012007.

[64] K. Alexis. "Resilient Autonomous Exploration and Mapping of Underground Mines using Aerial Robots". In: *2019 19th International Conference on Advanced Robotics (ICAR)*. 2019, pp. 1–8. DOI: 10.1109/ICAR46387.2019.8981545.

[65] R. Reinhart, T. Dang, E. Hand, C. Papachristos, and K. Alexis. "Learning-based Path Planning for Autonomous Exploration of Subterranean Environments". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1215–1221. DOI: 10.1109/ICRA40945.2020.9196662.

[66] Yu Fan Chen, Miao Liu, Michael Everett, and Jonathan P How. "Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 285–292.

[67] Mark Pfeiffer, Samarth Shukla, Matteo Turchetta, Cesar Cadena, Andreas Krause, Roland Siegwart, and Juan Nieto. "Reinforced imitation: Sample efficient deep reinforcement learning for mapless navigation by leveraging prior demonstrations". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 4423–4430.

[68] Abdur R Fayjie, Sabir Hossain, Doukhi Oualid, and Deok-Jin Lee. "Driverless car: Autonomous driving using deep reinforcement learning in urban environment". In: *2018 15th International Conference on Ubiquitous Robots (UR)*. IEEE. 2018, pp. 896–901.

[69] Rogerio Bonatti, Ratnesh Madaan, Vibhav Vineet, Sebastian Scherer, and Ashish Kapoor. "Learning Visuomotor Policies for Aerial Navigation Using Cross-Modal Representations". In: *arXiv preprint arXiv:1909.06993* (2019).

[70] Mayur J. Bency, Ahmed H. Qureshi, and Michael C. Yip. *Neural Path Planning: Fixed Time, Near-Optimal Path Generation via Oracle Imitation*. 2019. arXiv: 1904.11102 [cs.RO].

[71] Daniel Dugas, Juan Nieto, Roland Siegwart, and Jen Jen Chung. "NavRep: Unsupervised Representations for Reinforcement Learning of Robot Navigation in Dynamic Human Environments". In: *arXiv preprint arXiv:2012.04406* (2020).

[72] Lucia Liu, Daniel Dugas, Gianluca Cesari, Roland Siegwart, and Renaud Dubé. "Robot Navigation in Crowded Environments Using Deep Reinforcement Learning". In: (2020).

[73] Zijian Hu, Kaifang Wan, Xiaoguang Gao, Yiwei Zhai, and Qianglong Wang. "Deep reinforcement learning approach with multiple experience pools for uav's autonomous motion planning in complex unknown environments". In: *Sensors* 20.7 (2020), p. 1890.

[74] Jiankun Wang, Wenzheng Chi, Chenming Li, Chaoqun Wang, and Max Q.-H. Meng. "Neural RRT*: Learning-Based Optimal Path Planning". In: *IEEE Transactions on Automation Science and Engineering* 17.4 (2020), pp. 1748–1758. DOI: 10.1109/TASE.2020.2976560.

[75] David Ha and Jürgen Schmidhuber. "Recurrent world models facilitate policy evolution". In: *arXiv preprint arXiv:1809.01999* (2018).

[76] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. "Deep spatial autoencoders for visuomotor learning". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 512–519. DOI: 10.1109/ICRA.2016.7487173.

[77] David Hoeller, Lorenz Wellhausen, Farbod Farshidian, and Marco Hutter. "Learning a state representation and navigation in cluttered and dynamic environments". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5081–5088.

[78] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. *Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Dense and Sparse Reward Environments*. 2020. arXiv: 1910.04281 [cs.LG].

[79] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[80] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[81] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[82] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. "An Algorithmic Perspective on Imitation Learning". In: *Foundations and Trends in Robotics* 7.1-2 (2018), pp. 1–179. ISSN: 1935-8261. DOI: 10.1561/2300000053. URL: http://dx.doi.org/10.1561/2300000053.

[83] Wen Sun, Arun Venkatraman, Geoffrey J. Gordon, Byron Boots, and J. Andrew Bagnell. "Deeply AggreVaTeD: Differentiable Imitation Learning for Sequential Prediction". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, June 2017, pp. 3309–3318. URL: http://proceedings.mlr.press/v70/sun17d.html.

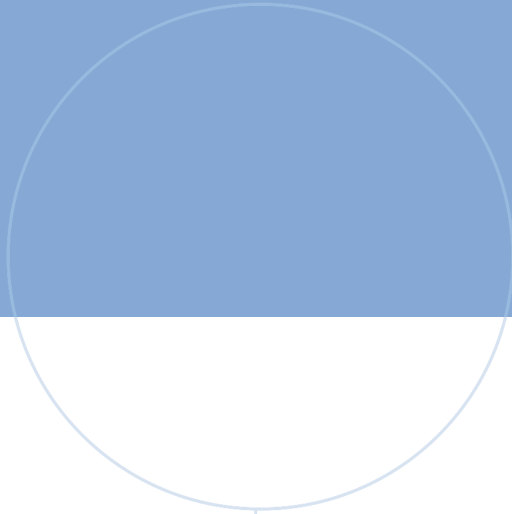
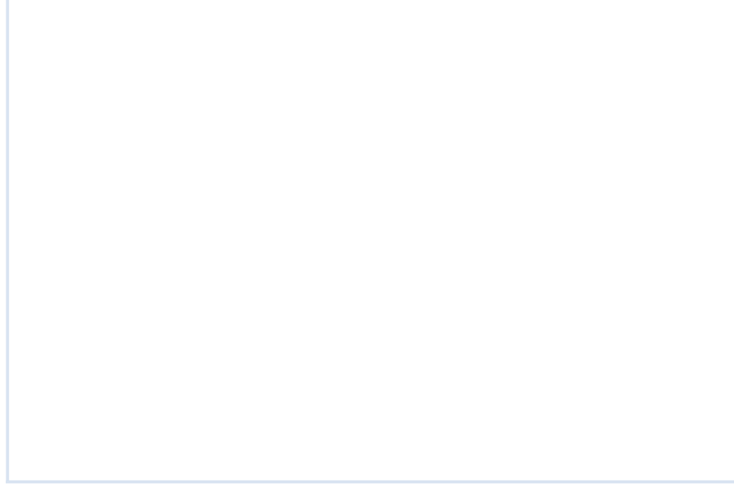
[84] Andrew Y Ng, Stuart J Russell, et al. "Algorithms for inverse reinforcement learning." In: *Icml*. Vol. 1. 2000, p. 2.

[85] Sertac Karaman and Emilio Frazzoli. "Sampling-based motion planning with deterministic μ-calculus specifications". In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE. 2009, pp. 2222–2229.

[86] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 627–635.

[87] C. Wang, J. Wang, J. Wang, and X. Zhang. "Deep-Reinforcement-Learning-Based Autonomous UAV Navigation With Sparse Rewards". In: *IEEE Internet of Things Journal* 7.7 (2020), pp. 6180–6190. DOI: 10.1109/JIOT.2020.2973193.

[88] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: https://www.sciencedirect.com/science/article/pii/089360809190009T.

NTNU

Norwegian University of
Science and Technology