Master's thesis

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

Sondre Holm Fyhn

Learning-Based Navigation in Cluttered Environments Using A Deep Collision Prediction Network

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Dr. Konstantinos Alexis
June 2021

NTNU
Norwegian University of
Science and Technology

Sondre Holm Fyhn

Learning-Based Navigation in Cluttered Environments Using A Deep Collision Prediction Network

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Dr. Konstantinos Alexis
June 2021

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

NTNU
Norwegian University of
Science and Technology

Abstract

Safe and reliable navigation of unmanned aerial vehicles (UAVs) has received increasing attention over recent years, largely due to a significant price-drop of onboard sensors, actuators and embedded computers powerful enough to run state-of-the-art technologies. Resilient aerial robotic autonomy depends on the robot's capacity to plan actions that efficiently optimize the robot's mission objectives while ensuring the safety of the robot.

Current state-of-the-art methods for autonomous exploration use high-dimensionality sensors to perform collision-free path or motion planning by querying an online reconstructed map of the environment which a) is computationally intensive and b) is increasingly more challenging in dynamic settings. The motivation behind this thesis is to construct a method that directly uses raw sensor data to control the drone's action without needing to reconstruct a map in real-time. Specifically, from a depth image, the planner presented in this thesis outputs directly from a forward velocity and a yaw angle command, replacing the need for a mapping module and a position controller. Moreover, since the planner is realized by a neural network, it can utilize the parallel computing capability of the GPU to speed up the planning time, allowing fast replanning when dealing with dynamic obstacles or severely cluttered environments.

As such, this thesis proposes a framework for a deep collision prediction model that can be used for local motion planning of aerial robots. The prediction model, $P(o_t, a_{t:t+H})$, provides the collision probability over an H-long action horizon, $a_{t:t+H}$, and a depth image, o_t. Local motion planning is done by evaluating a set of H-step motion primitives, where the model determines the best trajectory based on which motion primitive that has the lowest average probability of colliding with the environment.

Deep learning-based methods may require hundreds of thousands of data samples for training. Simulation engines are a powerful tool that quickly generates vast amounts of data at a low cost, enabling the utilization of deep neural networks for resilient aerial robotic autonomy. The framework presented in this thesis is divided into three components: (i) a self-labeling data generation method in a simulated environment, (ii) the creation and the training of the collision prediction network and (iii) a local planner which uses the model to navigate in cluttered environments while being exclusively motivated by intrinsic objectives. The model is tested with perfectly simulated depth images, noisy depth images and filtered depth images, as well as with

real-life filtered depth images. The model is also studied in different types of environments, including different levels of difficulty, with unfamiliar obstacles and with unfamiliar environments. The results from the study are encouraging and present the deep collision prediction network as a novel method of performing motion planning for aerial robots.

Sammendrag

Trygg og pålitelig navigasjon med ubemannede luftfartøy har de siste årene fått økt oppmerksomhet, mye grunnet en betydelig prisreduksjon av ombord-sensorer, aktuatorer og innvevde datamaskiner kraftige nok til å kjøre toppmoderne teknologi. Motstandsdyktig autonomi for flyvende roboter avhenger av robotens kapasitet til å planlegge handlinger som effektivt optimaliserer robotens oppdragsmålsettinger samtidig som den forsikrer egen sikkerhet.

Dagens toppmoderne metoder for autonom utforskning av omgivelsene bruker høydimensjonalitetsensorer til å gjennomføre kollisjonsfrie baner eller bevegelsesplanlegging ved å søke gjennom et kontinuerlig oppdatert rekonstruert kart av omgivelsene, noe som a) er beregningsmessig krevende og b) er økende vanskelig i dynamiske miljøer. Motivasjonen bak denne masteroppgaven er å konstruere en metode som direkte bruker ubehandlede sensormålinger til å kontrollere dronens handlinger uten å måtte rekonstruere et kart i sanntid. Mer spesifikt, fra et enkelt dybdebilde, vil planleggeren som er presentert i denne masteroppgaven gi en bane basert på en foroverhastighetskommando og en girvinkelkommando, og dermed erstatte behovet for en kartleggingsmodul og en posisjonskontroller. Dessuten, siden planleggeren er realisert ved et nevralt nettverk, kan den utnytte den parallelle beregningsevnen til en GPU for å redusere planleggingstiden. Dette muliggjør raskere omplanlegging i tilfeller med dynamiske objekter eller i svært tette omgivelser.

Følgelig foreslår denne masteroppgaven et rammeverk for et dypt kollisjonspredikasjonsnettverk som kan bli brukt for lokal bevegelsesplanlegging av flyvende roboter. Prediksjonsmodellen, $P(o_t, a_{t:t+H})$, gir kollisjonssannsynligheten over en H-lang handlingshorisont, $a_{t:t+H}$, og et dybdebilde, o_t. Lokal bevegelsesplanleggelse blir gjort ved å evaluere et sett med bevegelsesprimitiver som er delt i H antall deler, hvor modellen avgjør den beste banen basert på hvilken bevegelsesprimitiv som har den laveste gjennomsnittlige sannsynlighet til å krasje med omgivelsene.

Dyp læring-baserte metoder kan kunne kreve hundretusener dataprøver for trening. Simulering er et kraftig verktøy som hurtig kan generere store mengder data til en lav kostnad, og som muliggjør bruken av dype nevrale nettverk for motstandsdyktig autonomi av flyvende roboter. Rammeverket presentert i denne masteroppgaven er tredelt og består av: (i) en metode for generere selv-markerende data i simulerte omgivelser, (ii) dannelsen og treningen av et kol-

lisjonprediksjonsnettverk og (iii) en lokal planlegger som bruker modellen til å navigere i tette omgivelser mens den er utelukkende motivert av egen sikkerhet. Modellen er testet med perfekt simulerte dybdebilder, støyete dybdebilder, filtrerte dybdebilder, og til slutt med filtrerte dybdebilder fra virkeligheten. Modellen er også studert i forskjellige typer miljøer, med økende vanskelighetsgrad, ukjente hindringer og i ukjente omgivelser. Resultatene er lovende og presenterer det dype kollisjonspredikasjonsnettverket som en ny metode til å gjennomføre bevegelseplanlegging for flyvende roboter.

Preface

This master's thesis constitutes the culmination of my work completed at the Norwegian University of Science and Technology (NTNU) through the spring of 2021. Supervised by Prof. Dr. Konstantinos Alexis and with close cooperation with Ph.D. researcher Huan Nguyen, this thesis summarizes the methodologies used and presents the corresponding findings. Most of the proposed methods which are utilized in this thesis have, historically, not existed for a long time due to limitations in data resources and computational performance, especially regarding the technological advances of micro aerial vehicles. However, significant progress in recent years is enabling new technologies and has created a flourishing open-source community. Inspired by the great advances in relevant technologies, it was desired to create a system that could predict whether a drone is on a collision course or not based solely on past experiences. As such, this thesis aims to contribute to the field of motion planning control by applying a deep neural network learning approach. At the time of writing, there are extensive plans for a publication of this work.

Although this thesis offers a thorough presentation of the theory adopted with a special focus on supervised learning in artificial neural networks, it has been designed under the assumption that the reader inhabits prerequisite knowledge with respect to machine learning, control theory, simulation and optimization.

There are several contributions supplementing and assisting the development of this thesis. Firstly, the idea behind autonomous navigation of micro aerial vehicles in cluttered environments arose during the development of the project thesis written in the fall of 2020, where waypoint navigation of micro aerial vehicles in obstacle-free space using reinforcement learning was studied [1]. Secondly, the framework this thesis builds upon is created by various NTNU Autonomous Robots Lab (ARL) members and has been crucial for the development of this thesis. The ARL also provided a powerful computer which the network could train on with different configurations. Lastly, NTNU has provided an office space and a Dell OptiPlex 7070 i7-9700 computer equipped with Ubuntu 18.04 LTS and ROS Melodic Morenia.

Acknowledgment

Throughout this thesis, I have received a great deal of support and assistance. First, I want to thank my supervisor, Prof. Dr. Konstantinos Alexis, whose expertise and guidance have been invaluable during the thesis. I also want to thank Ph.D. researcher Huan Nguyen for all the assistance I have received and for patiently answering all my never-ending questions.

I then want to thank my study friends, who have brightened my day through our regular lunch and coffee breaks. Finally, I want to thank my friends and family for all of the support I have received during this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Safe and reliable navigation of autonomous unmanned aerial vehicles (UAVs) is a challenging task. The ability to successfully fly and navigate in cluttered environments is crucial for unlocking many robotic applications such as surveillance [2], search and rescue [3], industrial inspection [4] and so on. The increasing focus of using flying robots is largely due to a recent significant price-drop of onboard sensors, actuators and embedded computers. State-of-the-art technologies, like simultaneous localization and mapping (SLAM) [5, 6] and so on, are more available and easier to implement much due to a growing open-source community.

Resilient aerial robotic autonomy depends on the robot's capacity to plan actions that efficiently optimize the robot's mission objectives while ensuring the safety of the robot. This is referred to as optimizing the robot's *extrinsic goals* while ensuring the robot's *intrinsic motivations*. The traditional method of tackling the task of safe and reliable navigation is a two-step interleaved process consisting of (i) localization and mapping (through aiding measurements, such as LiDAR or a stereo camera), and (ii) computation of control commands to achieve obstacle avoidance while navigating towards a goal [7].

There are still applications where autonomous navigation is not fully explored due to difficulties caused by the environment [8]. Such applications could be autonomous exploration in sensor-degraded or dynamic environments, where current state-of-the-art technology (SLAM) is not thriving. Great appearance changes, homogeneous, long and narrow environments, or dynamic scenes can constrain even the most advanced SLAM algorithms and cause the system to make unrecoverable errors. In the case of dynamic environments, the complexity of resilient aerial robotic autonomy increases severely. Dynamic objects are required to be singled out from raw sensor data and their motion must be predicted. All the while, there must exist an information bridge to the controllers [9]. The task of navigating unknown cluttered environments requires an efficient combination of spatio-temporal filtering and trajectory optimization [10].

Depth images are highly affected by measurement noise [11] that depends on the characteristics of the environment. The proposed method needs to be robust to the noisy measurements so that the robot is able to avoid obstacles successfully.

Current state-of-the-art methods perform collision-free path or motion planning by querying an online reconstructed map of the environment which a) is computationally intensive and b) is increasingly more challenging in dynamic settings. Due to these reasons, the typical model-based pipeline for autonomous navigation in cluttered environments either assumes the use of a reactive planner in combination with a position controller to quickly react to oncoming obstacles in static environments [12], or tailored systems that have controllers that directly integrate velocity measurements of the quadrotor into the control formulation [13]. Other approaches involve fitting a velocity model on dynamic objects as a cost map of a trajectory optimizer [9, 14]. Deep learning-based approaches remove the need for feature engineering. Considerable research has been done in the field of safe and reliable navigation using learning-based approaches [15, 16, 17, 18]. Recent research have focused on closing the sim-to-real gap with policy training happens through simulation and synthetically generated depth images. This is mainly because there is a challenging reality gap, where reality could be much more unpredictable and noisy than in simulation. Deep learning-based methods may require hundreds of thousands of data samples, and simulation engines are a powerful tool for quickly generating vast amounts of data at a low cost.

Motion planning is of key importance for advanced applications such as autonomous exploration. Autonomous exploration of subterranean environments challenges traditional technologies and the level of resilient autonomy that can be achieved, as the scenes are often long, narrow and sensor-degrading. Relevant applications for autonomous exploration of subterranean environments might be search and rescue missions, monitoring, inspections and cave exploring. Traditional approach runs SLAM to simultaneously build the map of the environment and localize the robot within the map, then a planner uses the map and the robot's state to plan collision-free paths. Although excellent under a wide range of conditions [19], this approach requires significant computational time to map the environment, rendering it ineffective to deal with dynamic obstacles. Moreover, the planner module is also prone to the drift of the map and the robot's pose.

State-of-the-art methods for autonomous exploration use high-dimensionality sensors to create a map of the environment. A planner derives collision-free paths by querying the map and sends the commands to the robot. The motivation behind this thesis is to construct a method that directly uses raw sensor data to control the drone's action without needing to reconstruct a map in real-time. Specifically, from a depth image, the planner presented in this thesis outputs directly a forward velocity and a yaw angle command, replacing the need for a mapping module and a position controller. Moreover, since the planner is realized by a neural

network, it can utilize the parallel computing capability of the GPU to speed up the planning time, allowing fast replanning when dealing with dynamic obstacles or severely cluttered environments.

1.2 Objective and Method

This master thesis aims to develop a collision prediction model, $P(o_t, a_{t:t+H})$, where the inputs of the model are the current depth image, o_t, and some action sequence, $a_{t:t+H}$, where H is the horizon length, and the action sequence's corresponding trajectory is within the robot's field of view (FoV). The output of the prediction model is the collision probability over the H-long horizon. The probability model is then used to evaluate a set of H-step motion primitives, where the action sequence with the lowest average collision probability is executed in a receding horizon fashion to enable autonomous navigation in cluttered environments. The thesis encapsulates relevant research and theory for achieving this, in addition to presenting a comprehensive framework for local motion planning using a deep collision prediction network. The framework presented in this thesis is divided into three components: (i) a self-labeling data generation method in a simulated environment, (ii) the creation and the training of the collision prediction network and (iii) a local planner which uses the model to navigate in cluttered environments while being exclusively motivated by intrinsic objectives. The model is tested with perfectly simulated depth images, noisy depth images and filtered depth images, as well as with real-life filtered depth images.

The work in this thesis is inspired by the work done in the preceding semester during the project thesis [1], where navigation in obstacle-free space using deep reinforcement learning was studied. The long-term goal of this work is to create safe and reliable navigation that can be used for autonomous navigation in cluttered environments. This can benefit aerial robotic autonomy in a multitude of applications, including exploration in challenging environments, package delivery, target tracking and more.

1.3 Outline

- **Chapter 1: Introduction**. An introduction to the field of motion planning for aerial vehicles and the motivation behind why an alternative solution is desired, as well as a introductory note on the objective and the method of this thesis.

- **Chapter 2: Related Literature**. A presentation of relevant literature in the field of motion planning and its utilization for advanced missions such as autonomous exploration, and methods for learning-based navigation, with a focus on aerial vehicles.

- **Chapter 3: Theoretical Background**. A detailed technical summary of supervised learning, artificial neural networks, convolutional neural networks and recurrent neural networks. This chapter includes the theoretical background information required when developing a deep collision prediction network with a learning-based approach.
- **Chapter 4: Experimental Setup**. This chapter presents the software frameworks used in the master and the quadrotor platform.
- **Chapter 5: Proposed Approach**. The proposed approach for developing a deep collision prediction model. In-depth detail of the data generation process, the prediction model training and the model as a means for enabling autonomous navigation through local motion planning.
- **Chapter 6: Evaluation Studies**. Evaluation studies of the collision prediction model's performance during training and during local motion planning, as well as when used as a trajectory planner for navigating in different types of cluttered environments. Lastly, the model performance on real-life depth images is presented.
- **Chapter 7: Discussion.** A discussion around the results from the evaluation studies and the typical collision causes.
- **Chapter 8: Conclusion and Further Work**. The conclusion of the thesis and a brief discussion around what future work should involve.
- **Bibliography.**
- **Appendix A: Acronyms.**
- **Appendix B: ROS Topics Graph.**
- **Appendix C: Environments.**

Chapter 2

Related Literature

2.1 Motion Planning

There has been extensive work in the field of motion planning for UAVs. The prime approach of path planning for UAVs involves a policy for searching admissible paths in an online reconstruction of the local environment that optimizes some specified extrinsic goals while ensuring intrinsic motivations. For the simplest case of path-planning in cluttered environments, which is arriving at a pre-defined location while avoiding obstacles on the way, the prevailing methods involve sampling-based methods where either the robot's configuration space is sampled [20] or the robot's control space is sampled [21]. Karaman and Frazzoli [20] presented in 2011 the probabilistic roadmaps* (PRM*) algorithm and the rapidly-exploring random trees* (RRT*) algorithm, which are provably asymptotically optimal variations of PRM [22] and RRT [23]. Asymptotically optimal variations of the algorithms means that the returned solution's cost converges almost surely to the optimum. Their work also showed that the computational complexity of the new algorithms is within a constant factor of that of their probabilistically complete (but not asymptotically optimal) counterparts.

Real-world sensors are imperfect as they generate both random and systematic errors due to the sensor itself and degradation in the environment. Continuous methods like the Gaussian processes [24] have been widely used. Regular volumetric discretization in occupancy grids, like the Octomap presented by Hornung et al. [25], is usually sufficiently robust and fast. However, the octotrees in the octomap have an increasing complexity of $O(\log(n))$ for insertion and look-up, where n is the tree depth. Recent work in the field addresses this issue and utilizes voxel hashing for fast look-ups. In 2017, Oleynikova et al. [26] presented Voxblox: a volumetric mapping library-based mainly on truncated signed distance fields (TSDFs). TSDFs are fast to build and smooth out sensor noise over many observations and are designed to produce surface meshes. Voxblox was created for micro aerial vehicles (MAVs) that require fast and flexible local planning and that replans when new parts of the map are explored. This is typical for

MAVs that operate in unstructured and unexplored environments. Euclidean signed distance fields (ESDF) are combined with trajectory optimization methods to obtain obstacle distance information. The voxblox algorithm incrementally builds ESDFs from TSFDs.

2.2 Motion Planning for Advanced Missions

The overall problem of resilient planning relates to the UAV's ability to identify admissible paths that optimize some extrinsic goals iteratively. An admissible path is regarded as a path that ensures some intrinsic objectives, such as the robot's safety, and accounts for the system's limitations. Aerial robotics planning has developed from seeking a known goal destination to satisfying more complex mission objectives like autonomous exploration. This field has received a significant amount of attention in recent years. Some of the main areas that have been investigated include frontier-based exploration, sampling-based methods and learning-based techniques.

Frontier-based exploration is a collision-free motion planning method that guides the robot and involves identifying frontiers and respective robot configurations. Nuske et al. [27] present an autonomous exploration method used for river exploration. The MAV navigates purely with onboard sensing, with no GNSS aiding measurements and no prior map of the environment. They present a high-level goal-point extraction strategy that uses multivariate cost maps to maximize the information obtained during the exploration. The exploration algorithm uses multiple layers of information to maximize the river area to be explored. They also introduce SPARTAN-lite, a variation of SPARTAN (sparse tangential network) [28] that exploits geodesic properties on smooth manifolds of a tangential surface around obstacles. This helps the exploration algorithm to plan rapidly through free space.

In sampling-based methods, a path search is conducted in a sample-based manner to identify admissible and collision-free paths. Bircher et al. [29] proposed in 2016 a path planning algorithm for autonomous exploration of unknown space using a receding horizon "next–best–view" (NBVP) scheme. In an online computed tree, the method finds the best branch and the quality of which is determined by how much of the unmapped area that still can be explored. The method uses RRT to search for collision-free paths in the environment to find the best path, and then commands the robot to its first vertex. This is then repeated in a receding horizon manner. Dang et al. [30] presented in 2020 a novel graph-based subterranean exploration path planning method that is attuned to key topological properties of subterranean settings. The method is structured around a bifurcated local- and global-planner architecture. The local planning utilizes an RRT graph to efficiently find collision-free paths that optimize exploration while also respecting the robots' traversability constraints and dynamic limitations. The global planner works alongside the local planner to incrementally build a sparse global graph. The global planner is engaged when the robot must be repositioned or is commanded to return home. The

architecture of the global planner is built on the assumption that the subterranean environment is multi-branched and consists of tunnel-like networks, which often lead to dead-ends. Dharmadhikari et al. [31] proposed a local planner that uses motion primitives to identify the admissible paths. The method searches the configuration space while exploiting the dynamic flight properties of small aerial robots to tailor the combined need for large-scale exploration and confined environments while having limited endurance. The sampling-based exploration methods are often prone to be stuck in local minima, resulting in sub-optimal trajectories, and possibly not reaching global coverage. Inspired by these challenges, Schmid et al. [32] proposed an RRT*-inspired online informative path planning algorithm that expands a tree of candidate trajectories. The use of a single objective function allows the algorithm to achieve global coverage.

Studies regarding learning-based path planning policies for autonomous exploration have in recent years received much attention as they are starting to outperform state-of-the-art technologies. Reinhart et al. [33] utilizes the graph-based path planner presented by Dang et al. [30] as an expert learning in an imitation learning scheme. The trained policy is capable of autonomously exploring mine drifts and tunnels. The algorithm receives only a short window of measurements from a LiDAR sensor and achieves behavior similar to the expert learner with more than a magnitude reduction in computational cost.

2.3 Learning-Based Navigation

Path planning for aerial robots should consider the intrinsic motivations, such as safety or other objectives that do not relate directly to the specified mission, as well as the extrinsic objectives. The intrinsic motivations are important for the robot to execute the mission reliably. Achtelik et al. [34] proposed in 2014 to specifically exploit the information on uncertainty while simultaneously accounting for the robot's dynamics for realistically computing a path. For the path to be realistic, neither the uncertainty of the robot's perception nor the vehicle's dynamics can be ignored. They make use of rapidly exploring random belief trees (RRBT) to identify paths that lead to a waypoint. This is done by evaluating offline multiple path hypotheses in a known map to select the path exhibiting the motion required to estimate the robot's state accurately while inherently avoiding motion in unobservable modes.

Kahn et al. [35] implemented uncertainty-aware reinforcement learning to enable complex, adaptive behavior to be learned automatically for autonomous robotic platforms, such as UAVs or unmanned ground vehicles (UVGs). The mobile robot learns to navigate in an a priori unknown environment while avoiding collisions. When the robot knows little to nothing about the environment during training, high-speed collisions are more likely to occur and can ultimately damage the robot. Therefore, they saw it necessary for a successful learning method

to proceed cautiously and only to experience low-speed collisions until it gained more confidence. Therefore, they present an uncertainty-aware model-based learning algorithm that estimates the probability of collisions and a statistical estimate of uncertainty. The predictive model is based on a neural network and has high-bandwidth sensors such as a camera. The results achieved are successful in simulation and in real-world applications, both for UAVs and for UGVs. To realize safe, uncertainty-aware navigation, they proposed a model-based learning approach where the robot learns a collision probability model and uses the model's uncertainty to adjust its navigation strategy. This approach has served as one of the strongest inspirations for this thesis. A speed-dependent collision cost in combination with uncertainty-aware collision estimates resulted in a risk-averse navigation strategy. Prior work in the field of autonomous navigation has considered safety at training time by using an expert learner, such as the motion predictive control (MPC) algorithm with ground truth state information [36]. This is not advantageous for real-world deployment, as access to the ground truth state can not be assumed. This especially yields true in the case of GNSS-denied environments such as in subterranean scenes. Further work has since been done by Kahn et al. [37], and in 2020, they introduced an end-to-end self-supervised learning-based navigation system called "BADGR". Instead of human supervision to teach the robot to navigate in the environment, typically by mimicking an expert learner or providing labeled data, the robot's own past experience provides retrospective self-supervision. For several salient navigational objectives, like preferring less cluttered environments or smooth over bumpy terrain, the robot can use its experience and retrospectively label it so as to learn a predictive model for these objectives. The robot may therefore learn about the *affordances* in the environment using raw visual perception. Mobile robot navigation is usually regarded as a geometrical problem, where the robot's objective is to plan a collision-free path in a perceived environment. This is not desired because a purely geometric view of the world can be problematic in many navigation applications. An example of this is when a mobile robot is moving in a field of tall grass and believes that a given path over the grass is untraversable based on the geometric view of the scene, while in reality, it is feasible. BADGR is trained with self-supervised off-policy data gathered in real-world environments without the need for simulation or human supervision and can navigate in urban and off-road environments with geometrically distracting obstacles. A key feature of the algorithm is that it also incorporates terrain preferences, generalizes to novel environments and that it continues to improve "on-the-go" while gathering more data.

In 2018, Loquercio et al. [38] proposed DroNet; a convolutional neural network that can safely drive a drone through the streets of a city. The network is a fast 8-layers residual network that produces a steering angle and a collision probability from a single image input. Together, these two outputs make the UAV recognize dangerous situations and navigate the environment while avoiding obstacles. A vital aspect in which this work stands out is how they collect the

training data. There is a significant challenge in collecting enough data in an unstructured outdoor environment, such as a city, without simulation. Having an expert UAV pilot to gather data is not ideal due to the great amount of data required and, more importantly, the risk of involving other vehicles, pedestrians or similar in dynamic urban environments. Therefore, the work presented by Loquercio et al. [38] used data collected by cars and bicycles, which are already integrated into the urban environments. The results from the study were a highly generalized model that successfully could fly at relatively high altitudes and in indoor environments, such as parking lots and corridors as well.

Hoeller et al. [10] presented in March 2021 a method for learning state representation and navigation in cluttered and dynamic environments. In their work, they present a learning-based pipeline that realizes local navigation with the quadrupedal robot ANYmal [39] in cluttered environments with static and dynamic environments. A sequence of filtered depth images and the current trajectory of the camera are fused, forming a model of the world using state representation learning. The output of the module is fed into a target-reaching and collision-avoiding policy training with reinforcement learning. The key factor of their work is the state representation which helps bridge the reality gap, enabling successful sim-to-real transfer. Their experiments show that the system handles noisy depth images and successfully avoids dynamic and static obstacles in simulation and real-life situations. The model is also trained so that it has local spatial awareness. Situation awareness is obtained from depth images. Other work feeds RGB images or 2D LiDAR data through a neural network and learns from an expert learner [38, 40, 41], from a reinforcement learning scheme [42, 35] or from self-supervised learning [37]. RGB-based methods do not explicitly consider dynamic environments, which cause purely reactive policies that may be reacting too slowly for approaching obstacles. The use of depth images in a simulation-based learning setting, as the work presented by Hoeller et al. [10], helps lower the reality gap and reduces the complexity of the data and thereby the complexity of the neural network (by lowering the input space). The depth images are encoded through convolutional variational autoencoder (VAE) [43] and the model is trained with simulation images and depth images from the real robot. A sequence of processed depth images forms a sequence of latent vectors, which are fed into a long short-term memory (LSTM) unit with the camera trajectory. The model is maximizing the log-likelihood of the measured latent vectors. A KL-divergence term is included to make the predicted distribution consistent. The representation module can be perceived as a state estimator similar to a non-linear Kalman filter. The next latent can be predicted based on the current belief state, the previous inferred latent and the action input. This is commonly referred to as dreaming [44] in the literature.

Chapter 3

Theoretical Background

This chapter aims to present the fundamental theory required for developing a deep collision prediction network, enabling local obstacle avoidance for navigation in cluttered environments. As the obstacle avoidance principles used to accomplish this are based on several disciplines within machine learning and deep neural networks, the subsequent sections aim to present some of the essential pillars for making this achievable.

In the following sections, artificial neural networks is used as an umbrella term for deep neural networks, deep networks and other similar variations of this, as all the artificial networks presented are assumed multilayered and, hence, deep. Special cases of deep neural networks, such as convolutional neural networks and recurrent neural networks, will be discussed more in detail.

3.1 Supervised Learning

The defining characteristic of supervised learning is the availability and quality of training data. As the name invokes, there is an underlying idea of a *supervisor* that instructs an agent using labels associated with the training data. The labeled training data has clear criteria for the model optimization, making supervised learning techniques more powerful than unsupervised learning techniques [45]. Unlike supervised learning, unsupervised learning techniques have no labeled training data and must uncover hidden data structure and patterns independently. Supervised learning is analogous to humans learning from past experiences to enablef knowledge that prepares for the coming situations. Computers learn from data collected in the past to be capable of making intelligent decisions for the future.

It is common to distinguish supervised learning problems as either classification problems or regression problems. A classification problem is either a multiclass problem where the output variables belong in categorical classes or a binary problem with boolean output variables. A regression problem has a real value output variable, which could, for instance, be "dollars", "time"

or similar. Supervised learning algorithms intend to induce models based on labeled training data that can be used to classify unlabeled data. After a model is trained with the training data, it is evaluated with a separate data set referred to as the validation data. This is done to measure how well the model generalizes to data it has never seen before.

The accuracy of the model is calculated by:

$$\text{Accuracy} = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}} \tag{3.1}$$

One of the fundamental assumptions of machine learning is the relationship between the training data set and the validation data set. The assumption is that the training data distribution should be identical to the distribution of the validation data [46]. Significant violations will not surprisingly result in poor accuracy, which is intuitive as the model is not prepared for understanding the validation data set. High validation accuracy requires training data to be sufficiently representative of the validation data. The learning process of the model with training and validation can be seen in fig. 3.1.

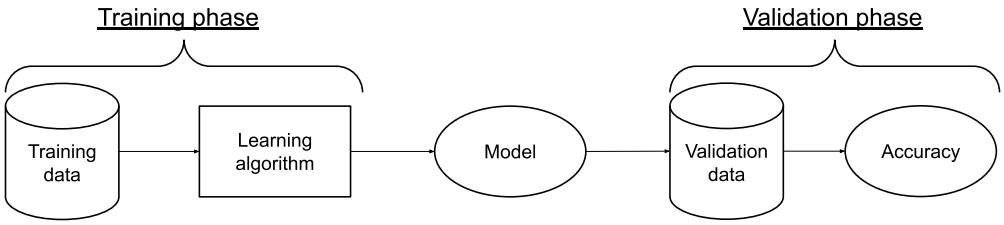


Figure 3.1: The learning process of a typical supervised learning scheme with training and validation.

Machine learning represents a field of computer science where machines have the ability to map an input variable, x, to an output variable, y, without being told explicitly how to do so. In supervised learning, which is a subset of machine learning, the agent is given training data from a labeled data set on the form $\{(x_0, y_0), (x_1, y_1), ..., (x_N, y_N)\} \in \mathscr{D}$, where N is the number of training examples and the data set in contained in $\mathscr{D}$. The overall goal of supervised learning is to find a mapping function $g : X \rightarrow Y$, where X is the set of all possible inputs $\{x_0, x_1, ..., x_{|X|} \in X\}$ and Y is the set of all possible classes $\{y_0, y_1, ..., y_{|Y|} \in Y\}$. The mapping function, g, exists in what is usually referred to as the hypothesis space [47] and it is sometimes convenient to represent using a scoring function $f : X \times Y \rightarrow \mathbb{R}$, such that

$$g(x) = \arg\max_y f(x, y), \tag{3.2}$$

where $f \in F$ is the space of scoring functions and $g \in G$ is the space of candidate functions. Depending on the input, output and the relationship between them, various methods of build-

ing these approximators exist. There are two basic approaches to determining g or f: Empirical risk minimization (ERM) and structural risk minimization (SRM). A loss function $L : X \times Y \rightarrow \mathbb{R}^{\geq 0}$ is defined to measure how well a function fits the training data. The loss of predicting the classification variable, $\hat{y}$, of the training data (x_i, y_i) is defined as $L(y_i, \hat{y})$. The training data is assumed to consist of independent and identically distributed pairs. The risk function gives the expected value of the loss as:

$$R(g) = \int L\big(y, f(x, g)\big) dP(x, y). \tag{3.3}$$

The goal is to minimize $R(g)$ over the class of functions $f(x, g)$. The joint probability distribution $P(x, y) = P(y|x)P(x)$ is unknown. The development of this risk function is what differentiates the two minimization methods ERM and SRM. The two methods are briefly explained in the following subsections.

3.1.1 Empirical Risk Minimization

Empirical risk minimization is recognized as a special form of standard convex optimization and is one of the most powerful tools in applied statistics [48]. It is a fundamental idea within machine learning, as it seeks the risk function that best fits the training data. The ERM method resembles many classical minimization methods, such as the least square method [49] and the maximum likelihood estimation method [50]. Under the assumption that g is a conditional probability distribution, $P(y|x)$, and the loss function is the negative logarithm likelihood, $L(y, \hat{y}) = -\log P(y|x)$, the ERM equals the maximum likelihood estimation method. It is constructed based on the training data and assumes that the function $f(x, \hat{g})$ minimizes

$$E_{\text{train}}(g) = R_{\text{ERM}}(g) = \frac{1}{N} \sum_{i}^{N} L\big(y_i, g(x_i)\big). \tag{3.4}$$

This results in a risk value close to its global minimum. The method is referred to as an empirical method due to the loss function being calculated from a sample of the domain set X, where the domain set is assumed to cover every possible input perfectly. As such, there exists a barrier between empirical error and true error. If the model is trained so that the function $f(x, \hat{g})$ minimizes the empirical error but not the true error, overfitting has occurred. This happens if the subset of X is not adequately representative for the entire data set or if G contains a vast amount of candidate functions [51]. If so, the ERM learning algorithm will eventually memorize the training data without producing a well-generalized model. In other words, the model will overfit. This could lead ERM to have high variance and poor generalization capabilities. This barrier between true and empirical error is useful because it allows ERM to guarantee, with a certain confidence, an upper bound to the error of the model [52].

3.1.2 Structural Risk Minimization

Structural risk minimization seeks to prevent overfitting by including a regularization penalty, $\lambda J(g)$, to the risk function, $E_{\text{train}}(g)$. The regularization penalty represents the complexity of the model. The lower the complexity of $J(g)$ is, the smaller is the value. The method is drawing inspiration from William of Ockham's principle of parsimony, meaning that the method prefers simpler functions over more complex ones. The punishment will force the model into being as simple as possible. The regularization penalty evaluates the bias-variance trade-off and is often expressed as the Euclidean distance of the weights (L2-norm) or as the Manhattan distance (L1-norm).

$$R_{\text{SRM}}(g) = E_{\text{train}}(g) + \lambda J(g). \tag{3.5}$$

λ controls the bias-variance trade-off, meaning it minimizes the training error and the expected gap between the training error and the validation error. When $\lambda = 0$, the empirical risk minimization has low bias and high variance. When λ is large, the empirical risk minimization will have high bias and low variance.

3.1.3 Imbalanced Data Sets

The true accuracy and precision of classification predictive modeling are heavily reliant on how well the data set contains representative class labels for the given examples. An imbalanced classification problem occurs when the distribution of known class labels within the data set is skewed or biased. The distribution can be slightly skewed or heavily skewed. For instance, the distribution of real-life credit card transaction fraud would typically be heavily skewed, as most transactions are done legally. Given a binary data set where 95% of the samples are negative and 5% of the samples are positive, the model would achieve 95% accuracy by predicting negative for all samples, which is not desired. Therefore, when dealing with imbalanced data sets that are not dealt with, accuracy is no longer an interesting metric to measure, as it can be more deceiving than informative. Measuring the *precision* and the *recall* are more relevant when analyzing the model's performance on heavily skewed data sets. Precision is measured as the percentage of predicted positives that were correctly classified and recall is measured as the percentage of actual positives that were correctly classified. They are given as:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}, \quad \text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}. \tag{3.6}$$

A true positive (TP) sample is when the predicted class is positive and the true class is also positive. In contrast, false positive (FP) is when the predicted class is positive, but the true class

is negative. This logic also applies to true negative (TN) and false negative (FN) predictions. Therefore, it is desired to minimize the false positives and the false negatives and maximize the number of true positives and true negatives. A much-used method of visualizing the performance of a classification model is through a confusion matrix, as illustrated in fig. 3.2.

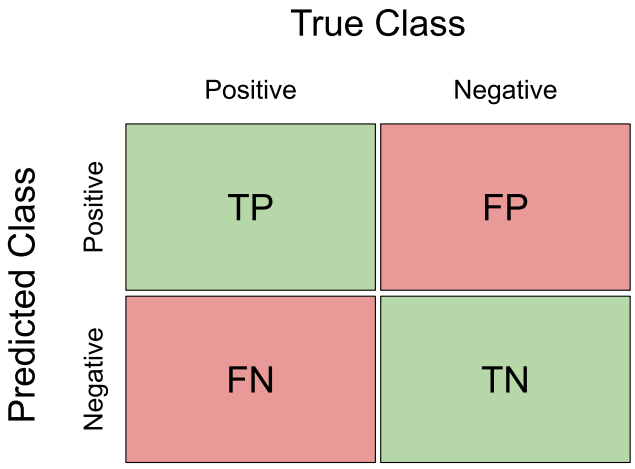


Figure 3.2: Illustration of a confusion matrix, representing the distribution of true and false negatives and positives.

A well-established method of measuring the accuracy of binary classification problems is to measure the model's harmonic mean of the precision and the recall. This metric is commonly referred to as the $F1$-score and is given as:

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{\text{TP} + \frac{1}{2}(\text{FP} + \text{FN})}. \tag{3.7}$$

When the data set is imbalanced, the model will have adequate information about the majority class but insufficient information about the minority class. This leads to high misclassification errors for the minority class and is a problem as it typically is the minority class on which predictions are most important. Possibly neglecting or ignoring the minority class completely could have potentially catastrophic outcomes. Given an imbalanced data set used for training a collision prediction network, a misclassification of the unrepresented label might lead to the model predicting "no collision" even though a collision is likely. Class weighting and oversampling are among the most used techniques to tackle the imbalanced data set problem and will be more discussed in section 3.2.4. If the minority class has few examples, class weighting could be introduced to make the model "pay more attention" to the underrepresented class. A related approach is to oversample the minority class. Oversampling the minority class could be

achieved by duplicating examples from the minority class in the training data set. As such, class distribution can be balanced without the need to provide additional information to the model.

3.2 Artificial Neural Networks

Artificial neural networks (ANNs) are adaptive systems inspired by the functioning system of the biological brain. ANNs are systems that can modify their internal structure to adapt to a function objective [53]. This property is well suited when implementing machine learning techniques, as the goal is, in most cases, to map some input value to the desired output. ANNs are particularly well suited for modeling continuous and non-linear problems, as they can reconstruct the fuzzy logic [54] that governs the optimal behavior of these problems. Therefore, one could perceive the use of neural networks as an optimization algorithm. The ANN architecture is inspired by the structure and function of the brain's neural network. An ANN consists of artificial neurons which are interconnected and arranged in various layers. Much like the functioning of the synapses in a biological brain, each artificial neural connection transmits a signal to other neurons. In the case of ANNs, this signal is always a real scalar, usually within some limit. The layers are categorized into three parts: the input layer, the hidden layers and the output layer. As an example of this: Given an image classification problem, the input layer would be the image, the output layer would be some classification, and the hidden layers would contain the neurons that discover the patterns in the image that separates the possible classes. The typical artificial neural network architecture is shown in fig. 3.3, and a closer illustration of the artificial neuron is shown in fig. 3.4.

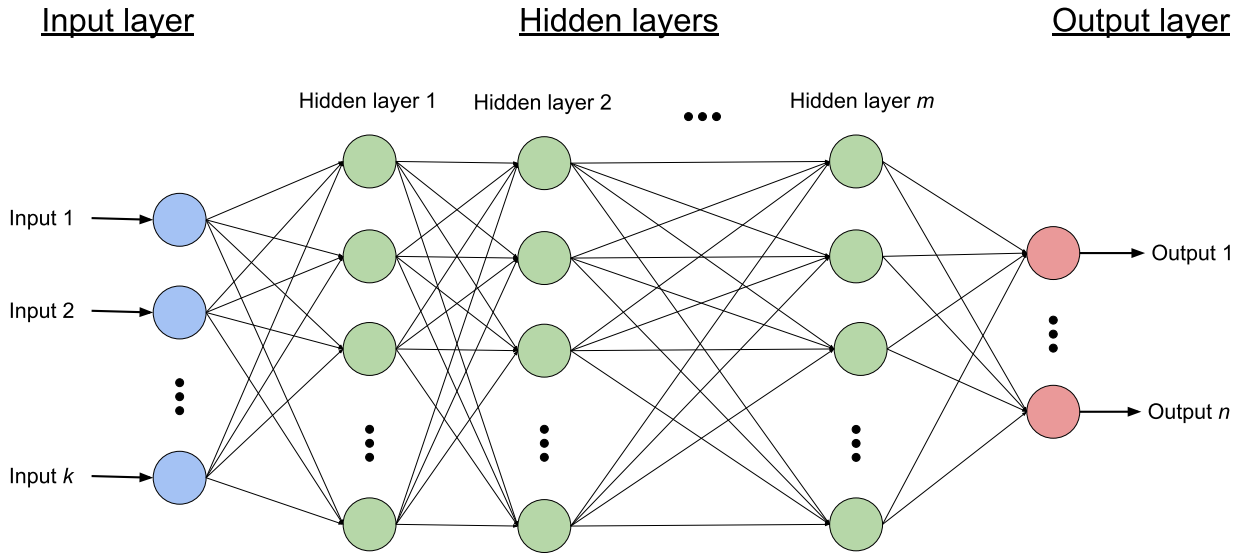


Figure 3.3: The artificial neural network architecture.

The connection between two neurons is called an *edge*. The output of each neuron is computed by some non-linear function of the sum of its weighted inputs and a bias term. Each neuron has weights, $\mathbf{w} = \{w_1, w_2, ..., w_n\}^T$, that corresponds to the neuron's input signals $\mathbf{x} = \{x_1, x_2, ..., x_n\}^T$. The weights are the core of the artificial neural networks, as these are the ones that adapt as the model learns. The information that flows between the neuron layers needs to be converted so that the output of one neuron can be used as an input in another neuron. An activation function, $\varphi(\cdot)$, is placed at the output of each neuron and brings the important property of handling the non-linearity in a neural network.

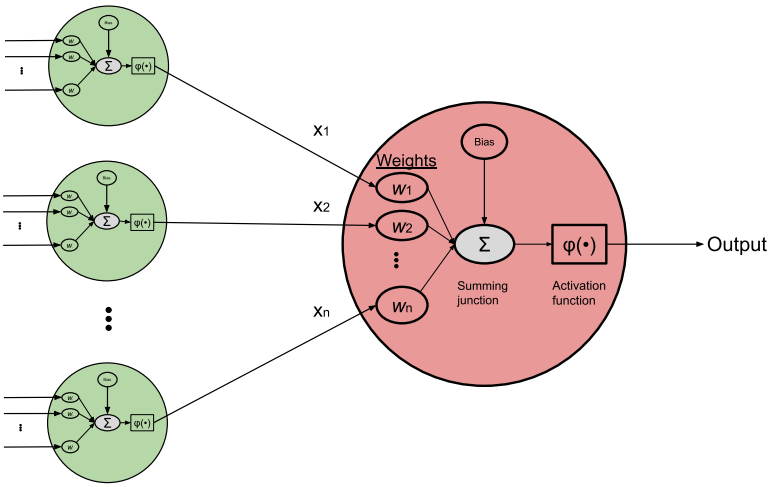


Figure 3.4: Neuron in artificial neural network.

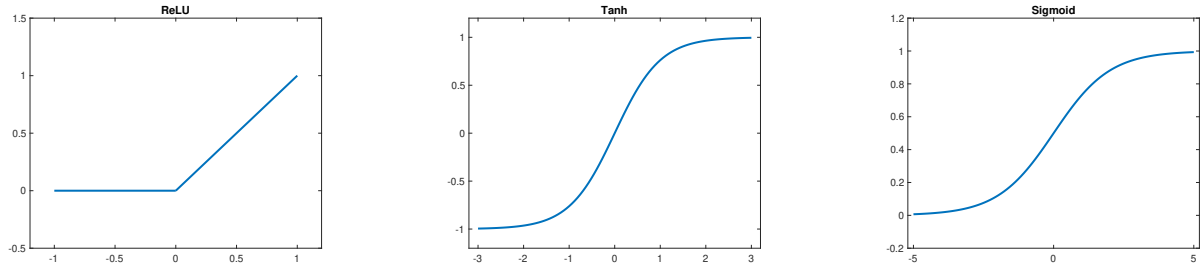
In 1958, Frank Rosenblatt introduced the simplest structure of an artificial neuron called the *perceptron* [55]. The activation function in a perceptron is a simple Heaviside step function, making each neuron either *on* or *off*. Therefore, the output of a perceptron is given as

$$y = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0 \\ 0, & \text{otherwise.} \end{cases}$$

The perceptron was mainly used for linear classification problems, where there was a binary output value. However, structuring these perceptrons in layers would expand its utilization area into more challenging classification problems. This was first presented by Rumelhart et al. [56] in 1986. The structured layers of perceptrons were coined a "multilayer perceptron" (MLP) and enabled higher-level feature extraction from input data. Changing the Heaviside step function increased the flexibility and limited the number of neurons required. Today, the MLP is the basis of artificial neurons, and the output is given as

$$y = \varphi(\mathbf{w}^T\mathbf{x} + b) \tag{3.8}$$

There exist several different activation functions. Amongst the most popular ones are the rectified linear unit (ReLU), the sigmoid function and the tanh function. The sigmoid function is sometimes denoted "σ" in this thesis. The shape of each function is shown in fig 3.5. The sigmoid and tanh activation functions have a similar structure, especially around $x = 0$, but the tanh function is zero-centered and allows negative values, whereas the sigmoid function does not. The sigmoid function is rarely used as an activation function in the hidden layers of a deep neural network as it is computationally expensive, not zero-centered and tends to shift the gradients towards zero. The last property is called the vanishing gradient problem and is due to the network's depth and the activation shifting the value to zero. However, the sigmoid function is more often used as the activation layer between the last hidden layer and the output layer. This is typically if the output should be some value between $[0, 1]$, for instance, in the case of percentage scores. tanh has similar properties as the sigmoid function, except being zero-centered. ReLU is computationally efficient and does not cause the vanishing gradient problem. However, the ReLU activation function can cause neurons to die, as the output is zero for all inputs less or equal to zero. Another inconvenience with ReLU is that positive inputs are not saturated, which sometimes can lead to unusable neurons. Several other activation functions have been introduced to tackle these issues but will not be considered in this thesis. Honourably mentions are leaky ReLU [57], parametric exponential linear unit [58] and Hard-Swish [59].



(a) ReLU. (b) tanh. (c) Sigmoid.

Figure 3.5: The ReLU, tanh and sigmoid activation functions.

3.2.1 Artificial Neural Network and Supervised Learning

One can think of supervised learning as an optimization problem, where the objective is to optimize a mapping function so that the input is categorized to some output in the best possible manner. Therefore, it is favorable to use artificial neural networks, which essentially is a self-learning optimization method. The model uses a given algorithm to navigate the space of possible sets of weights the model may utilize to make good predictions. Typically, the algorithm used for training the model is some variation of a stochastic gradient descent optimization algorithm, where the weights are updated using backpropagation of an error algorithm. The gradient descent refers to the error gradient, and the gradient descent algorithm seeks to modify the weights so that the error between the predicted output and what the actual output should be, is minimized. To measure this difference, there exists a *loss* function and a corresponding *loss* value, also popularly known as a *cost* function (with the corresponding *cost* value). More on the concepts of the gradient descent algorithm and backpropagation are presented in Section 3.2.2 and Section 3.2.3. In the context of optimization theory, the loss works as an objection function, where it is used to evaluate a set of weights, which would correspond to a candidate function. The way the loss function stands out is that it reduces all the various good and bad aspects of a complex system down to a single scalar value. This allows the possible candidate functions to be ranked and compared [60]. Choosing a loss function that captures the properties of the problem can be challenging, although it is essential. It is important that the loss function chosen faithfully represent the design goals. Specifying the goal of the learning scheme is, therefore, especially important.

The Loss Function

Numerous functions exist to estimate the error of a set of weights in an artificial neural network. As briefly mentioned in Section 3.1.1, the maximum likelihood estimation (MLE) method seeks to minimize the dissimilarity between an empirical distribution of the data set with the unknown validation data set. The MLE method is often a preferred estimator for machine learning

applications because it is consistent and efficient [61]. Under appropriate conditions, the MLE method is assumed consistent because; as the number of training data approaches infinity, the MLE of a parameter converges to the true value of the parameter.

In regards to classification problems, the goal is to map an input variable to a class label. By introducing cross-entropy [62], the problem can be modelled as predicting the probability of a sample. This means the output will be a "score" for how likely it is that the input variable should be labeled by which classification label. For instance, given an animal recognition model, the classification output of a blurry image input of a dog could be "75% dog". As such, the model has the ability to express the output with a certain degree of confidence. In a binary classification problem, where the output is either *true* or *false*, the output would be how likely the model evaluates the output as either positive or negative. This could, for instance, be how likely a model evaluates a trajectory to cause a collision. By using MLE, cross-entropy would seek to find a set of weights that minimizes the difference between the model's predicted distribution and the distribution of the probabilities given in the training data set. For binary classification problems, this is referred to as binary cross-entropy. Cross-entropy is often referred to as logarithmic loss because given a distribution q relative to the distribution p over a given set. It is defined as

$$H(p,q) = -E_p\big[\log q\big]. \tag{3.9}$$

3.2.2 Backpropagation

Backpropagation is a widely used algorithm for training ANNs and is considered to be the "workhorse" of how neural networks learn. The algorithm works by computing the gradient of the loss function with respect to each weight. It is computed one layer at a time in a backward fashion, hence its name. Consider the notation showed in fig. 3.6.

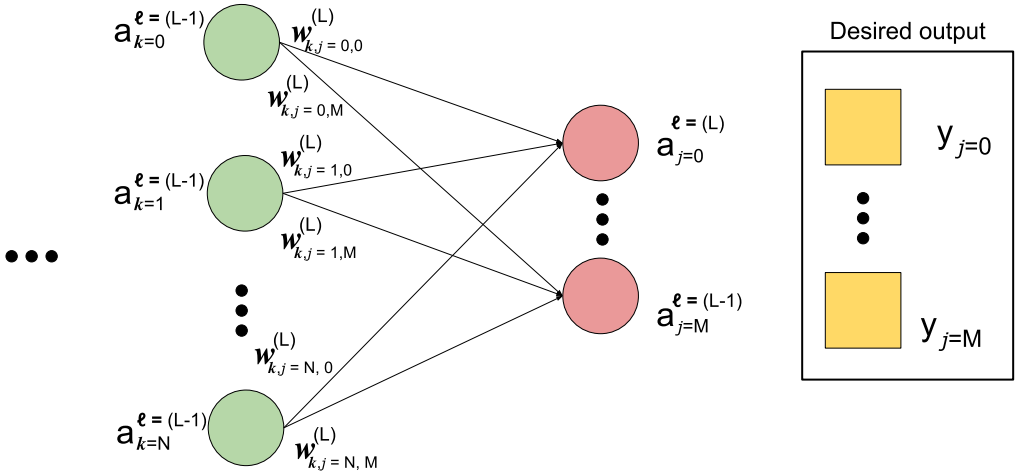


Figure 3.6: A subset of an ANN showing the naming convention for activations, weights and the desired output.

The weight is denoted $w_{kj}^{(l)}$ for the connection between the k^{th} neuron in the $(1-l)^{th}$ layer to the j^{th} neuron in the l^{th} layer. Similar notations are used for the biases and the activations. $b_j^{(l)}$ is the bias for the j^{th} neuron in the l^{th} layer. $a_j^{(l)}$, where $j = \{0,1,...,M\}$, is the activation of the j^{th} neuron in the l^{th} layer. The same yields for the activation $a_k^{(l)}$, where $k = \{0,1,...,N\}$. With this notation, the activation $a_j^{(l)}$ is related to the activations in the previous layer $(1-l)$. This can be written as:

$$a_j^{(l)} = \varphi\left(\sum_{k=0}^{N}\left(w_{kj}^{(l)}a_k^{(l-1)} + b_j^{(l)}\right)\right), \tag{3.10}$$

where φ is the activation function. By rewriting this into a vectorized form, where each layer is combined into a vector and the weights as a $(k \times j)$ weight matrix, eq. (3.10) can be written as:

$$a^{(l)} = \varphi(w^{(l)}a^{(l-1)} + b^{(l)}), \tag{3.11}$$

The weighted input to the neurons in layer l in vecotrized format is calculated as:

$$z^{(l)} \equiv w^{(l)}a^{(l-1)} + b^{(l)}. \tag{3.12}$$

Or alternatively, as a sum of each neuron:

$$z_j^{(l)} \equiv \sum_{k=0}^{N}\left(w_{kj}^{(l)}a_k^{(l-1)} + b_j^{(l)}\right). \tag{3.13}$$

However, from here, only the vectorized version will be used. The activation of the output layer

can therefore be written as:

$$a^{(L)} = \varphi\big(z^{(L)}\big), \tag{3.14}$$

and the candidate function as:

$$g(a^{(0)}) = a^{(L)} = \begin{bmatrix} a^{(L)}_{j=0} \\ \vdots \\ a^{(L)}_{j=M} \end{bmatrix}, \tag{3.15}$$

where $a^{(0)}$ is the input layer, referred to as $\mathbf{x}$ previously .

The desired output is denoted as $y = [y_{j=0}, \ , y_{j=1}, \ , y_{j=M}]^T$. The loss function is denoted $C(a^{(L)}, y)$, as not to confuse with the layer number L. The goal of backpropagation is to compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the loss function with respect to any weight w or bias b in the network. The error in the j^{th} neuron in the l^{th} layer is defined as $\delta^{(l)}_j$. The backpropagation algorithm is a procedure to calculate $\delta^{(l)}_j$ and relate this error to the partial derivatives $\frac{\partial C}{\partial w^{(l)}_{kj}}$ and $\frac{\partial C}{\partial b^{(l)}_j}$. The error is defined as:

$$\delta^{(l)} \equiv \frac{\partial C}{\partial z^{(l)}_j} \tag{3.16}$$

and is expressed in a vectorized form as $\delta^{(l)}$. The backpropagation algorithm is based on four fundamental equations that enables computation of both the error and the gradient of the loss function:

$$\delta^{(L)} = \nabla_a C \circ \varphi'(z^{(L)}), \tag{3.17a}$$

$$\delta^{(l)} = \big((w^{(l+1)})^T \delta^{(l+1)}\big) \circ \varphi'(z^{(l)}), \tag{3.17b}$$

$$\frac{\partial C}{\partial b} = \delta, \tag{3.17c}$$

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}, \tag{3.17d}$$

where $\circ$ is element-wise vector multiplication, sometimes referred to as the Hadamard product. Eq. (3.17a) is for computing the error in the output layer L. $\nabla_a C$ is the vector whose components are the partial derivatives $\frac{\partial C}{\partial a^{(L)}_j}$. Eq. (3.17b) is an equation for the error in the l^{th} layer in terms of the next layer $(l+1)$. These two equations are enough for computing all the errors, $\delta^{(l)}$, in the network. This is because eq. (3.17a) gives the error for the last layer and eq. (3.17b) gives the previous error. This can therefore be repeated in a backward fashion until all errors are calculated. Eq. (3.17c) is an equation for the rate of change of the loss with respect to any bias

in the network, where it is understood that δ is evaluated in the same neuron as the bias b. Eq. (3.17d) is an equation for the rate of change of the loss with respect to any weight in the network. It is understood that the subscripts denote that a_{in} refers to the input activations and that δ_{out} refers to the outgoing error.

3.2.3 The Gradient Descent Algorithm

The gradient descent algorithm is an optimization technique that occurs in the backpropagation phase when training a deep neural network. The goal of the gradient ascent algorithm is to continuously resample the gradient of the model's parameter to minimize the loss function. The gradient descent algorithm works iteratively to find a set of weights, $\mathbf{w}$, that minimizes a cost function, $C(\mathbf{w})$. Finding the minimum of a function explicitly is not always feasible for concave functions, especially with increasing complexity. Certainly not when the input space is vast, as is the case in many deep neural network applications. Given an initial starting point, the gradient descent algorithm locates a local minimum by moving in the direction of the current point's gradient, $-\nabla C(\mathbf{w})$. There is no guarantee that the local minimum that the gradient descent algorithm finds is the global minimum of the function. The step size is proportional to the gradient to help the algorithm prevent overshooting. This means that as the weights found provides a loss value close to a local minimum, the step size becomes smaller. An illustration of how the gradient descent algorithm iteratively works is shown in fig. 3.7.

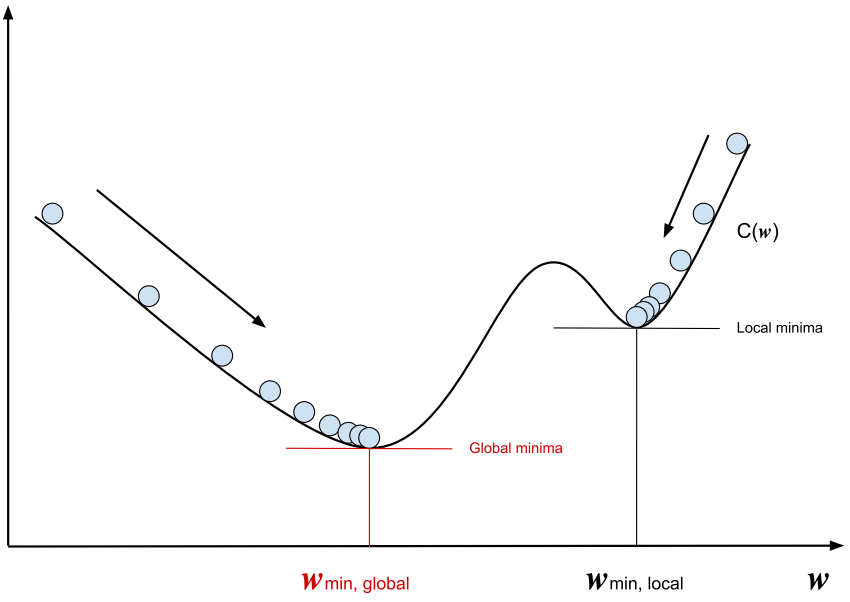


Figure 3.7: A simplified illustration of the gradient descent illustration, where the algorithm seeks to find a local minimum of the cost function $C(\mathbf{w})$, by adjusting the weights $\mathbf{w}$.

An important comment regarding the illustration is that it is drastically simplified, as the input space easily could be in the thousands and that the biases are neglected. These simplifications are necessary for the sake of the illustration's value. The loss function in the illustration, $C(\mathbf{w})$, refers to the loss function given as $L(y_i, g(x_i))$ when applied in neural networks. The reason the loss function is written as $C(\mathbf{w})$ in the fig. 3.7 is for the illustrational purposes.

If the algorithm uses a gradient from a subset of the entire data set instead of the data set as a whole, it is referred to as a stochastic gradient descent method. This is usually the case since deep neural networks require a large amount of training data, and processing the data in batches increases computational speed drastically. The stochastic gradient descent algorithm is presented in algorithm 1.

Algorithm 1: Stochastic Gradient Descent Pseudo Code

Result: Local minima

Initialize $\mathbf{w} := 0^{m-1}$, $b := 0$

for $t \in [1, ..., T]$ **do**

  Draw a random example from the dataset $(x_i, y_i) \in \mathscr{D}$

  Compute $g(x_i)$

  Compute the loss $\mathscr{L}_i := L(y_i, g(x_i))$

  Compute the gradients $\nabla\mathbf{w} = \nabla_{\mathscr{L}_i}\mathbf{w}$, $\nabla b := \frac{\partial \mathscr{L}_i}{\partial b}$

  Update the parameters $\mathbf{w} := \mathbf{w} + \nabla\mathbf{w}$, $b := b + \nabla b$

3.2.4 Dealing With Imbalanced Data Sets in Artificial Neural Networks

As discussed in section 3.1.3, an imbalanced data set occurs when examples of one class outnumber examples in another class or classes, possibly to a great extent. This could result in the model predicting biased towards the majority class and perform poorly on the minority class. An example of this could be when training a model to detect credit card fraud, where some hundreds of transactions are fraudulent, while many thousand are not. As the structure of ANNs is now more familiar, this subsection aims to discuss some of the methods on how to deal with imbalanced data sets in supervised learning. The most important aspect in dealing with imbalanced data is to know how skewed the data set is. This means counting how many samples there are of each class in the data set. In the case of a binary problem, this is usually referred to as how many positive and negative samples that exists in the data set.

An intuitive and rather quick method of dealing with imbalanced data is to set the correct initial bias to the model's output layer. Adjusting the last layer's bias will help speeding up the convergence. By setting the initial bias, the model does not have to spend time the unnecessary

time it takes to learn that whether a positive sample is likely or not. The correct initial bias can be derived from:

$$
\begin{aligned}
p_0 &= \frac{pos}{pos+neg} = \frac{1}{1+e^{-b_0}}, \\
b_0 &= -\log_e(\frac{1}{p_0} - 1), \\
b_0 &= \log_e(\frac{pos}{neg}).
\end{aligned} \tag{3.18}
$$

With the naive bias initialization, the expected initial loss would be $\log_e(2) \approx 0.693$. However, with the bias initialization, the initial expected loss would be around:

$$
-p_0 \log_e(p_0) - (1-p_0)\log(1-p_0). \tag{3.19}
$$

Another method of dealing with imbalanced data sets is to introduce class weight optimization. This makes the model weigh the minority class more heavily than the majority class. Class weights will cause the model to "pay more attention" to samples from the under-represented class. This is important as the model has adequate information about the majority class but insufficient about the minority class. The class weights will modify the training algorithm to take the skewed distribution of the classes into account. Given the imbalanced data set consisting of fraudulent credit card transactions, as mentioned earlier, the class weights for positive and negative samples would be very different. There is, however, a trade-off with class weights. Even though the model (with applied class weights) will predict more true positive samples in the fraudulent credit card example, it will also predict more false positives, meaning that the model will have lower accuracy but higher recall. False negatives would cause fraudulent transactions to go through, while false positives would cause an email to be sent to the customer asking them to verify their card activity. There is a cost to either type of error, although in this case, the costs are higher with false negatives. A consideration of the trade-offs should be done given the application. In the case of a binary classification problem, the class weights could be set as:

$$
w_0 = \frac{1}{neg} \cdot \frac{(pos+neg)}{2}, \tag{3.20a}
$$

$$
w_1 = \frac{1}{pos} \cdot \frac{(pos+neg)}{2}. \tag{3.20b}
$$

where scaling by $\frac{pos+neg}{2}$ helps keep the loss to a similar magnitude.

3.2.5 Monte Carlo Dropout

Dropout is a regularization technique used during the training phase of a neural network that helps prevent overfitting. As stated earlier, overfitting happens when the learning algorithm memorizes the training data without producing a well-generalized model. Meaning it performs well on the training data but poorly on unfamiliar data. This occurs if the subset of the labeled training data, X, is not representative (enough) for the entire data set or if the set of candidate functions, G, is vast. Dropout is a technique where a different set of neurons, at each step, are switched off. Each neuron has a probability, p, of being switched off at each time step. The probability is referred to as the *dropout rate* and is usually between $[0, 0.5]$. An illustration of how dropout would be in an ANN during training is shown in fig. 3.8.

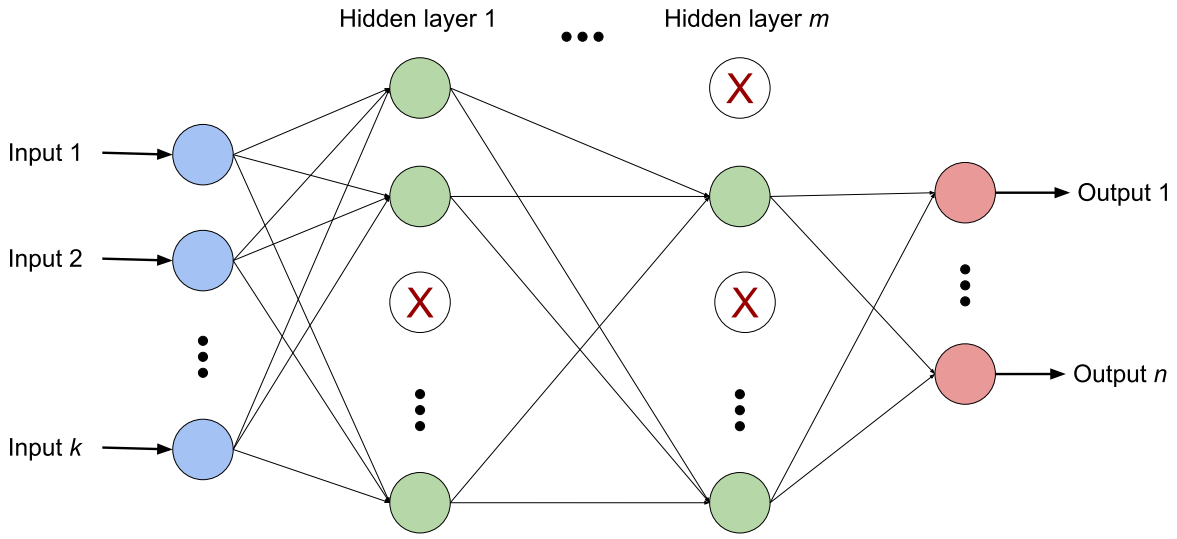


Figure 3.8: Artificial neural network with dropout regularization.

There are two main reasons why using dropout is beneficial for training. The first reason is that the information spreads more evenly in the network, meaning each neuron cannot expect input from the same neurons every time during training since each of its sources could disappear at any time. As a result, the network becomes less sensitive to input changes and achieves better generalization. The second reason is that the outcome of the network represents an averaging resemblance of many slightly different neural networks, where each separate network has trained on a single batch.

Monte Carlo dropout (MCD) is a variation of dropout proposed by Yarin Gal and Zoubin Ghahramani in 2016 [63]. The MCD method is a dropout method as a Bayesian approximation. MCD samples a network from the space of all possible models. Monte Carlo dropout is, contrary to regular dropout, also kept on during the validation phase. It is kept on so that it can gener-

ate multiple different predictions for each subset of the network. MCD can thereby improve prediction accuracy and can be used to estimate model uncertainty

3.3 Convolutional Neural Networks

A convolutional neural network (CNN) is a deep artificial neural network commonly used for analyzing two-dimensional or three-dimensional image data. Two-dimensional image data is typically grayscale images such as depth images, and three-dimensional image data is typically color images (which has three channels; red, green and blue). The name "convolutional neural network" indicates that the network applies a convolution, which is a mathematical operation that produces a third function $(f * g)$ that expresses how the shape of one function, f, is modified by another, g. CNNs has its name since it is a specialized type of a deep artificial neural network that uses convolution in at least one layer. The motivation for CNNs is to enable machines to perceive images, much like humans do. This could, for instance, be to classify a handwritten number. Traditionally, a very challenging problem for machines as each handwritten number is unique. However, CNNs take advantage of the hierarchical patterns in the data and assemble patterns of increasing complexity by using smaller patterns discovered by the network. For classifying handwritten numbers, a CNN would ultimately make a network that recognizes features in handwritten numbers to make classification possible. These features could be fairly obvious, such as a straight horizontal line indicating that the number is "1" and so on. The features could be much more abstract and make little sense for the human eye to perceive.

CNNs are inspired by receptive fields found in biological processes. The optimization function of a CNN is to optimize the *kernels*, also referred to as filters. The convolution is a linear operation that calculates the scalar output of the dot product between an array of input data (part of the image) and a two-dimensional array of weights, where the weights are the kernel. In the animal cortex, individual cortical neurons respond to stimuli in restricted regions, called the receptive fields [64]. The receptive fields of different neurons partially overlap so that they cover the whole visual field, which is the idea behind the kernels in CNNs. The process of transforming the input to a feature map using a systematically applied kernel is illustrated in fig. 3.9. The kernel used in the illustration would detect vertical lines. There are numerous different kernels with respect to both shape and magnitude. Which kernel to use depends on the application. The objective of a CNN is to find these kernels by itself through training. One alteration to this process is to include *strides*, which would make the kernel jump over a step in the input. In the illustration below, the stride is 1, meaning that the (2×2)-kernel would overlap. With a stride of 2, there would be no overlap. A brief comment worth noting is that the kernel applied is not a mathematical convolution, as used in digital signal processing or other related mathematical fields, but *cross-correlation*. However, in the community, it is referred to as convolution.

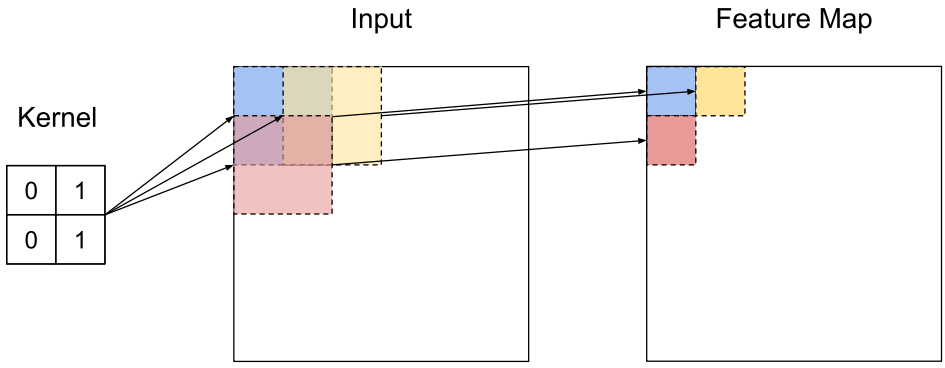


Figure 3.9: A vertical (2×2)-kernel applied to a two-dimensional input to create a feature map. The kernel is applied three times, represented with blue, red and yellow. In reality, it would cover the entire input. The process has a stride of 1, making the kernel overlap during the feature retrieval of the input. The overlapping places in the two-dimensional input are colored by the mix of the kernel colors.

A typical structure of CNNs is to have fully connected layers in the beginning, which creates a greater depth of the input. It is common to apply the kernels on the multi-dimensional outputs of the fully connected layers. This results in multi-channeled feature maps as well since the kernel must have the same depth as the input image. The convolutional layers are usually *pooling layers*. Max-pooling and min-pooling are the most popular pooling layers, where they extract the maximum value or the minimum value of the subpart of the image, respectively. It is common to stack these layers multiple times to extract lower-level features deeper into the network. The deeper into the network, the harder it would be for a human to sense what the network is abstracting. The first hidden layers could make sense for the human eye, as they could be more obvious features, like vertical or horizontal lines.

3.4 Recurrent Neural Networks

A recurrent neural network (RNN) is an ANN that allows previous outputs to be used as inputs while having hidden states, meaning they introduce a sense of memory into the network. RNNs allow a network not to start from "scratch" at every iteration, meaning it does not throw away everything it has learned between every new situation. As of now, only traditional neural networks have been considered, so-called *feed-forward* ANNs. Feed-forward networks are somewhat limited, as it takes in a fixed size input that limits its usage in situations where a sequence of inputs should be considered and where the size of the sequence is not predetermined. RNNs address this issue by having loops in the networks, allowing information to persist. Consider the network structure given in fig. 3.10, where the input and output at time t is given as X_t and h_t respectively. The hidden states of the network given as A, are often referred to as the special units.

The illustration presents alternative representations useful when explaining how the RNNs hold information from the past. RNNs can be thought of as multiple copies of the same network, where each copy passes a message to its successor. In fig. 3.10 this is presented by unrolling the recurrent loop into a chain-like structure.

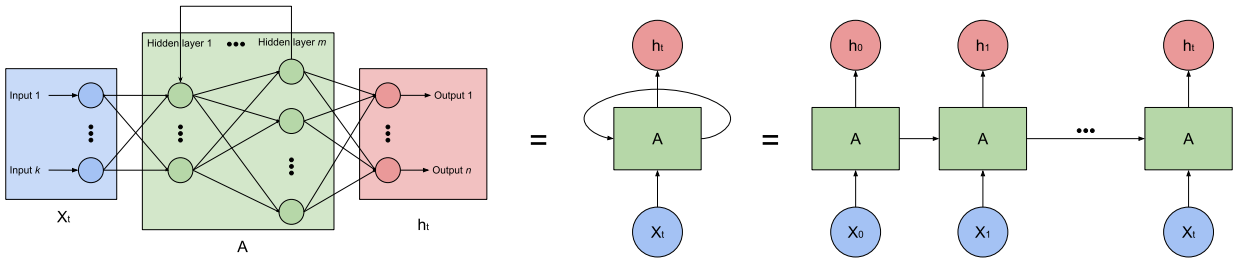


Figure 3.10: Alternative illustrations of the recurrent loops in a recurrent neural netowrks. Illustration inspiration from http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

RNNs are popular for tasks where data is handled sequentially, such as natural language processing, image captioning, translation applications, or even generating highly realistic handwriting samples or creating Wikipedia[1] articles [65]. Consider a model that suggests the next word in a sentence, as often found in modern devices such as smartphones. If one of the earlier words reveals that the sentence's verb should be in the past tense, the model should consider this word when predicting the next words. Consider the following sentence:

> *After a long day at school, Timmy ran all the way home.*

The correct time clause of the verb is given by the *time word* "after". The model will therefore be influenced by what has already occurred earlier in the sequence. This example regarding grammar is relatively poor since there is a set of given rules and a predefined correct word. RNNs are more often utilized if the answer is not necessarily correct but likely given in a clue from the past. An example of this could be:

> *Timmy misses the time when he was an exchange student in Italy. Next week, he is going to meet his friends from his exchange time at a restaurant. Timmy's favourite dish is pasta.*

In this sentence, the model that predicts the next word might guess pasta (or maybe some other Italian dish) since previous sentences reveal that Timmy has a positive association with Italy. The RNN is designed to take a series of inputs with no predetermined limit on size, so even though "Italy" was mentioned two sentences prior, it still affected the predicted word. Fig. 3.11 is an attempt to present this.

[1] https://www.wikipedia.org/

Naturally, the same logic applies in many other fields, where clues from the past may indicate something about the future.

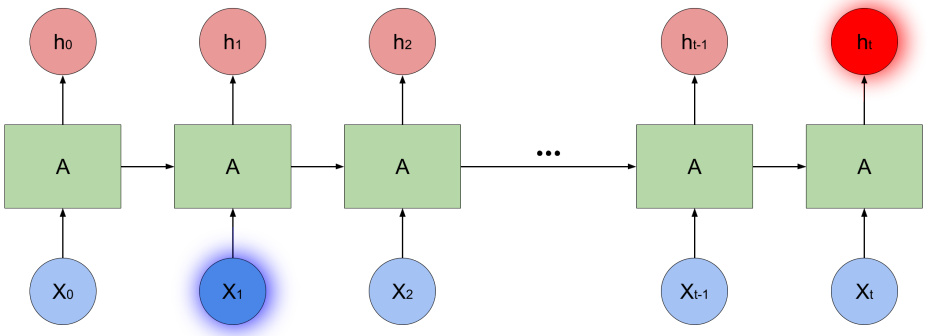


Figure 3.11: Illustration of a RNN where an input in the past affects an output in the future.

As the gap between the revealing inputs and later predictions grows, the RNN becomes unable to connect the information. Although theoretically possible to achieve with simple RNNs, it is in practice a very challenging problem. This issue is explored in depth by Bengio et al. [66]. Long short-term memory (LSTM) was later introduced by Hochreiter and Schmidhuber [67] as a method capable of learning long-term dependencies. LSTMs are, as of today, the de-facto method of implementing RNN.

3.4.1 Long Short-Term Memory

The long short-term memory algorithm was proposed to include long-term memory in RNNs. RNNs had at the time significant results but suffered from short-term memory, rapidly growing complexity and time inefficiency. LSTM is a recurrent neural network architecture in conjunction with a gradient-based learning algorithm that learns to bridge time intervals in few iterations. This is achieved by enforcing a constant error flow through the internal states of special units. The special units correspond to the modification to the neural network A as presented in the figures above. To understand how these special units work, it is advantageous first to look at how simple RNNs are structured. Simple RNNs have simple, special units, hence their name. An illustration of a simple RNN with output at time h_t, is presented in fig. 3.12. The special unit consists of a tanh-layer, meaning that the activation function of the neurons is tanh. The output at time t can therefore be written as:

$$h_t = \tanh\big(W[h_{t-1}, x_t] + b\big), \tag{3.21}$$

where b is some bias and W is the trained weights. The notion $[h_{t-1}, X_t]$ refers to the two inputs being concatenated.

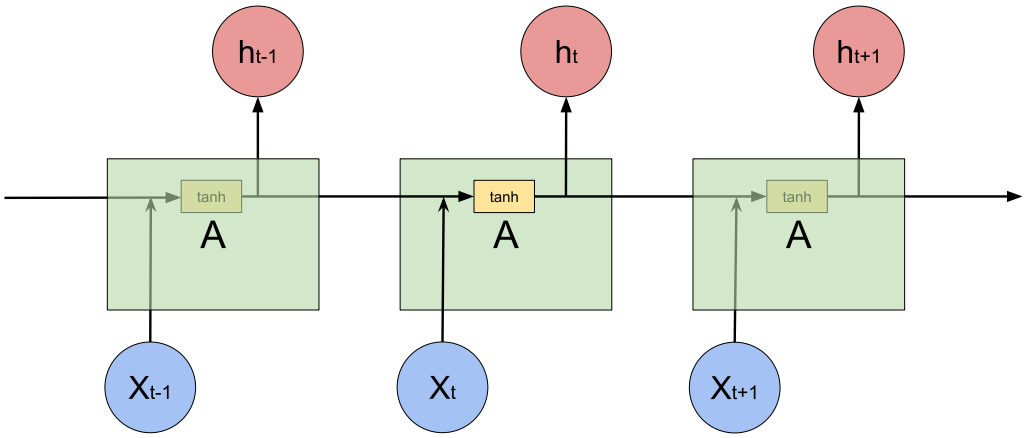


Figure 3.12: Strucutre of simple RNN special unit. Illustration inspiration from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

The LSTM has a similar chain-like structure as the simple RNN structure but contains four network layers and the "cell state". The LSTM special unit architecture can be seen in fig. 3.13 The cell state, $C_0, ..., C_t, C_{t+1}, ...$, runs through the entire chain, much like the output, $h_0, ..., h_t, h_{t+1}, ...$ However, it is much less complex and contains only some minor linear interactions. It is easy for information to flow unchanged in the cell state, but it can be modified through carefully regulated gates. These gates consist of a neural network layer, either a sigmoid layer, denoted σ in fig. 3.13 or a tanh-layer. The major difference between the sigmoid activation function and the tanh activation function is that the sigmoid function has an output between $[0, 1]$, while tanh has an output between $[-1, 1]$. The activation functions are discussed in more detail in section 3.2.

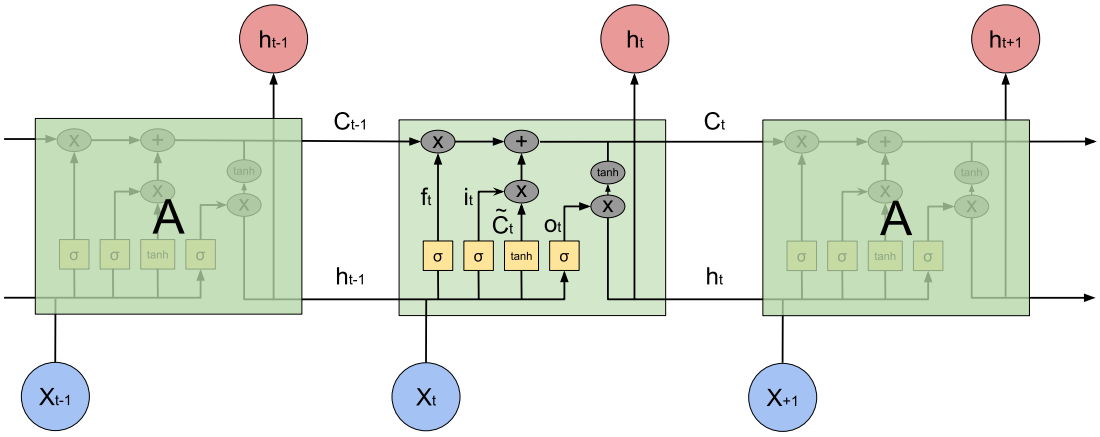


Figure 3.13: Structure of a LSTM special unit. Illustration inspiration from http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Consider the special unit structure shown in fig. 3.13. The neural layers are represented as yellow quadrilaterals and point-wise linear operations as gray ellipses. The first sigmoid layer

is usually called the "forget gate" and is denoted f_t. The forget gate is given in eq. (3.22). This is the layer where updated information replaces what the model already knows. Consider the sentence about Timmy's relation to Italy and the model predicting his favorite dish to be pasta. If a sentence later mentions something about Timmy's friend's positive relation to Spanish cuisine, the model should forget about the Italian relation and change to Spanish instead. This logic would happen in the forget state. The reason it is a sigmoid and not a tanh at the forget gate is that the output should be between $[0,1]$ since there is a point-wise multiplication between f_t and $_{t-1}$. $f_t = 1$ would mean forget everything and replace with new information and $f_t = 0$, would mean toss the new information. The forget gate is given as:

$$f_t = \sigma\big(W_f[h_{t-1}, X_t] + b_f\big). \tag{3.22}$$

The two next neural layers determine what to keep in the cell state. The second part is a tanh-layer which creates a set of candidate values, which are values that could be added to the cell state. This layer is denoted $\tilde{C}_t$ and given in eq. (3.23b). The first part, which is similar to the forget gate, is usually called the "input gate" and is denoted i_t and given in eq. (3.23a). This layer decides how much each new candidate value should be used to update the state cell.

$$i_t = \sigma\big(W_i[h_{t-1}, X_t] + b_i\big), \tag{3.23a}$$

$$\tilde{C}_t = \tanh\big(W_C[h_{t-1}, X_t] + b_C\big). \tag{3.23b}$$

The cell state can now be updated. This is done by a linear point-wise operation between the outputs of the two gates and the previous cell state. The old state is first multiplied with the forget-gate, essentially forgetting what the sigmoid layer has deemed worth forgetting. Then, the point-wise product of the input gate and the candidate values are added. This is seen in eq. (3.24). In terms of the word predictor model mentioned earlier, this would be where the model forgets about Timmy's relation to Italy and updates with the friend's relation to Spain.

$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t. \tag{3.24}$$

The last part is called the output layer, denoted o_t, and determines the special unit's output. The output layer is a filtered and conservative version of the cell state. The output layer is a sigmoid layer, much like the input state and the forget state and given as:

$$o_t = \tanh\big(W_o[h_{t-1}, X_t] + b_o\big). \tag{3.25}$$

Finally, the output of the special unit can be defined as:

$$h_t = o_t \circ \tanh\big(C_t\big). \tag{3.26}$$

3.5 Deep Collision Prediction Networks

A deep collision prediction network, as created in this thesis, determines the possibility of a collision given relevant information about the moving robot's current position in respect to the environment and its planned trajectory. In this thesis, the relevant information is a depth image at the current time step, o_t and an H-long action sequence over the current time step, $a_{t:t+H}$. Given these inputs, the prediction model provides a prediction of how likely a collision is for each action in the action sequence. The collision prediction is useful because it allows the evaluation of several action sequences prior to executing them. This means that a trajectory planner can choose the trajectory with the lowest possibility of colliding before commanding the UAV where to go. The previous sections and subsections provide the necessary theoretical background for understanding how the network works.

A collision prediction network with a depth image and an action sequence as inputs can have an architecture as shown in fig. 3.14. The neural network takes the current depth image as input and processes it with a CNN-block and several fully connected layers. This forms the initial hidden states for a recurrent LSTM layer. The LSTM recurrent unit takes an action horizon with length H in a sequential fashion. The output of the LSTM is fed through some fully connected units and has a sigmoid layer between the last hidden layer and the output layer. The prediction model produces H outputs where each output corresponds to an action in the action sequence input. The last activation layer is a sigmoid function because the output should be squashed between $[0,1]$ to represent probability. The output is a collision prediction ranging from 0% – 100% for every action step provided to the network, and each possibility corresponds to the corresponding action in the action sequence. A possible path-planning algorithm can use this model to evaluate several different trajectories over a depth image and choose the best trajectory based on a set of the trajectories' corresponding action sequences.

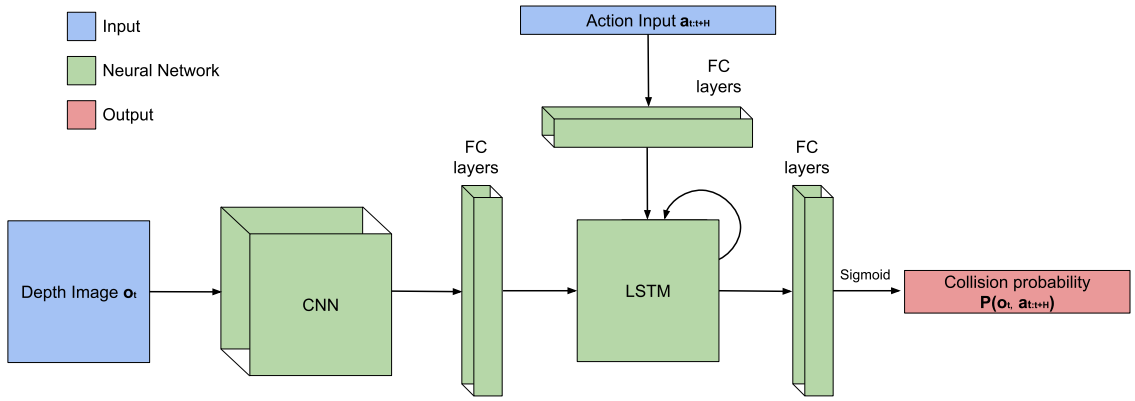


Figure 3.14: Illustration of a deep collision predictive model with a depth image input and an action sequence input which may be used as the pipeline for a learning-based navigation policy. A depth image is fed through a CNN and several fully connected layers (denoted FC in the figure). This forms the hidden states for an LSTM-block, which takes an H-long action sequence as input. This is further processed through some fully connected layers and a sigmoid layer. The output of the model is the collision prediction for each action in the H-long action sequence.

Chapter 4

Experimental Setup

4.1 Software Frameworks

This section aims to present the various software frameworks used in the thesis. Several frameworks were exploited in order to do underlying tasks such as communication, simulation and network training. All of the software used in this thesis are open-source.

4.1.1 Gazebo

Gazebo is a three-dimensional dynamic robotics simulator, which builds on the Open Dynamics physics engine [68] and the OpenGL rendering [69]. The simulator engine supports sensor simulation and actuator control, which is especially important in this thesis as simulated depth images are used to train the network. Sensors, environmental links, robotic links and joints and the environmental scenes are represented through universal robot description format (URDF) files. A URDF file for describing the drone encapsulates a detailed description of physical properties, such as motor thrust, mass and so on. The files also describe links to the world frame and a detailed three-dimensional geometrical description of the flying robot, making realistic visualization and collision handling is possible. Gazebo offers several essential components of a rigid object at high fidelity, such as inertia, mass, friction and various sensors.

The Gazebo extension RotorS [70] has been a key component throughout the work carried out in this thesis. RotorS is an extension for Gazebo that is specialized in micro aerial vehicles (MAVs) and provides packages necessary to simulate the flying robot in cluttered environments. Gazebo allowed for efficient data generation necessary when training the network and making efficient testing possible during development without having to risk real-life drones. It allowed testing of specific situations that would have been significantly more difficult in real-life settings, such as navigating a 10m × 100m corridor filled with obstacles. The similarity between simulation and real-life applications makes for a more or less seamless transfer when realizing

the model, with only minor adjustments required.

4.1.2 ROS

The robotic operating system (ROS) [71] is a set of software libraries and tools which are tailored for developing robotic applications. ROS is an open-source robotics middleware suite and an essential building block for numerous advances within research and development in today's technology. The name "ROS" can be somewhat misleading, as the system is not an operating system but a collection of frameworks and services for heterogeneous computer clusters, such as low-lever control, message-passing, package management and hardware abstraction.

The message-passing and package management that ROS offers contribute to seamless and straightforward communication between processes. The processes are called *topics* and the different parts of the robotic system may subscribe to or publish to a specific topic. For instance, sensory measurements are typically subscribed to and low-level control topics are typically being published on. Low-level control of robotic devices is also facilitated due to the device drivers and hardware abstractions that ROS provides.

ROS's core functionality is augmented by various tools that enable visualization, data recording, navigating package structures, automating configuration and setup processes, and flow diagrams of topic connections. ROS also supports a variety of programming languages, such as C++ and Python. This is beneficial since it allows different parts of the system to be programmed in different languages. Because ROS supports several programming languages, it also provides rich documentation and has a thriving community.

ROS is also embedded into Gazebo, which allows for a smooth transition between simulation and real-world applications. Gazebo also provides services like pausing and unpausing the environment, which is beneficial during training and evaluation. The work done in this thesis has been conducted based on ROS Melodic Morenia with Ubuntu 18.04 LTS. A *ros graph* containing all the ROS topics and their relations used in this thesis are shown in Appendix B.

4.1.3 TensorFlow

TensorFlow [72] is a free end-to-end open-source software library for building, training and deploying machine learning models developed by the Google Brain team. TensorFlow constructs computational, high-level structured graphs that allow automatic differentiation of neural networks. High-level APIs are also available, such as TENSORFLOW.KERAS, which is TensorFlow's implementation of the highly popular Keras API [73]. Keras offers the implementation of essential building blocks in a neural network, such as fully connected layers, pooling layers, dropout and batch normalization as well as optimizers, objective functions, activation functions, regularizers and several more. Keras also offers functionality such as data pipelines and estimators.

4.2 Quadrotor Platform

The quadrotor used in this work is the resilient micro flyer (RMF) created by De Petris et al. [74]. The RMF is a collision-tolerant small aerial robot tailored to traversing and searching within highly confined environments. The robot is lightweight and agile while still having a collision-tolerant design. It has a small size with a 0.32m diameter. The physical drone is capable of 14 minutes of endurance while weighing less than 500g. A digital twin of the RMF, has been used in RotorS in this thesis. The real quadrotor and its digital twin simulated in RotorS are both compatible with ROS. The local motion planning system based on the collision prediction network could be transferred from the simulated drone to the real drone with only minor adjustments required. The real-life RMF and its digital twin are shown in fig. 4.1.

(a) The real-life RMF. (b) The RMF's digital twin.

Figure 4.1: The RMF and its digital twin. Images taken from (a) https://www.autonomousrobotslab.com and (b) https://tiralonghipol.github.io/poldepetris.

4.2.1 Hardware and Sensors

The physical RMF is equipped with NVIDIA Jetson Xavier NX[1], which is a lightweight system-on-module (SOM) capable of running deep neural networks in parallel and processing multiple high-resolution sensors. The SOM is optimized for running on full artificial intelligence systems. Included in the SOM are a GPU (NVIDIA Volta with 384 NVIDIA CUDA cores and 48 Tensor cores), a CPU(6-core NVIDIA Carmel ARM v8.2 64-bit CPU 6 MB L2 + 4 MB L3) and memory (8 GB 128-bit LPDDR4x 51.2GB/s).

The RMF is also supplied with an inertial measurement unit (IMU), consisting of a gyroscope, an accelerometer and a magnetometer. The IMU provides measurements in three axes and provides translational and rotational information essential for flight. The drone is also equipped with Intel RealSense Depth Camera D455[2], which has an ideal range between 0.6m

[1] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/
[2] https://www.intelrealsense.com/depth-camera-d455/

to 6.0m and a field of view of $87° \times 58°$. The depth resolution can be up to 1280×720 with a frame rate up to 90fps. The depth accuracy is $< 2\%$ at 4m. Even though the depth image resolution can be high, it is not desired for the neural network as the increased input space would be more computationally expensive. Due to the increased complexity, the depth image resolution is (480×270) in this work. The D455 is also equipped with an RGB camera.

Chapter 5

Proposed Approach

This chapter presents the proposed approach for creating the framework that enables learning-based navigation for MAVs in cluttered environments using local motion planning with a deep collision prediction network. The proposed framework consists of three parts: (i) a self-labeling data generation method, (ii) the collision prediction network creation and training and (iii) a local motion planner that enables navigation in cluttered environments while being exclusively motivated by intrinsic objectives.

5.1 Data Generation

When generating data for supervised learning, it is essential to gather a sufficiently large and diverse dataset that facilitates for a well-generalized model and that prevents overfitting. The data generation method created in this thesis generates data from a given number of episodes, where the UAV is initialized at a random position within the environment at every episode. The obstacles within the environment are shuffled at every tenth episode to create a diverse dataset. An episode is defined from the UAV's initialization to a collision or a "time-out" has occurred. The time before the episode is timed out is set to 30 seconds during the data generation. A 10m×100m corridor environment with five different types of obstacles is used as the environment. During the shuffling of the obstacles, the number of each obstacle in the environment, their placement, sizes and orientations about the z-axis are randomized within some limitations. The five different shapes used as obstacles during the data generation are spheres, vertical stones-shapes, pyramids, U-shapes and ⊥-shapes. The corridor environment is shown in fig. 5.1 and the different shapes are shown in fig. 5.2.

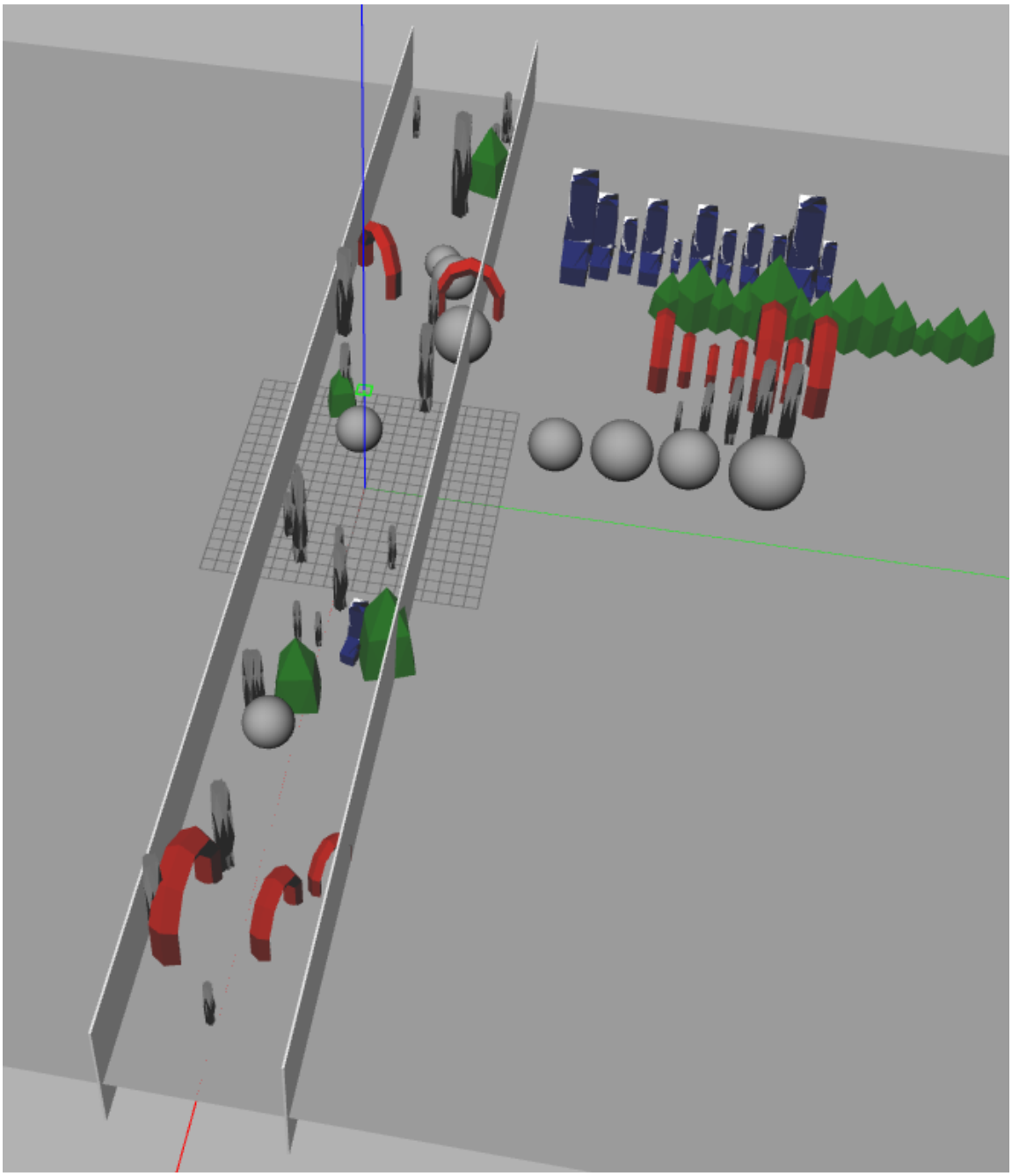
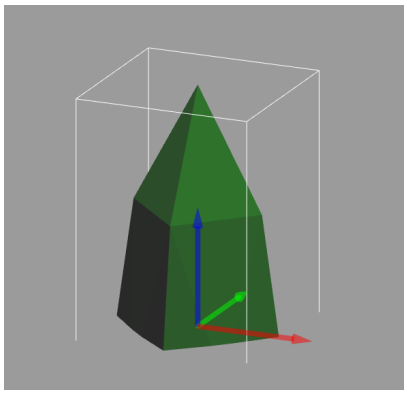
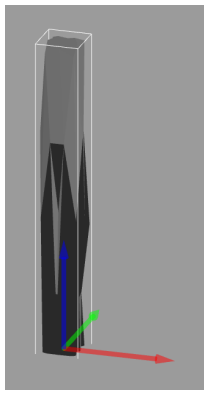


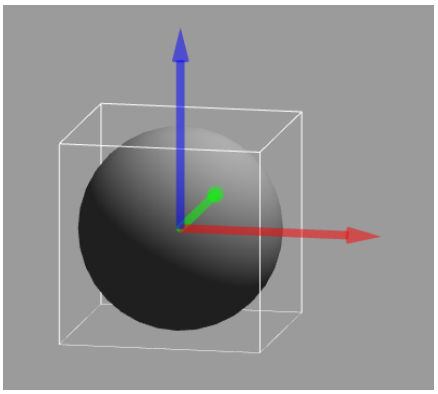
Figure 5.1: The 10m×100m corridor environment randomly cluttered with obstacles. The obstacles inside the corridor are randomly drawn from the set of obstacles found outside the environment. The UAV is randomly initialized within the environment at every episode and the environment is shuffled at every tenth episode.



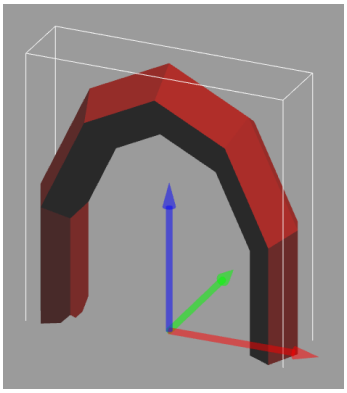
(a) Pyramid.



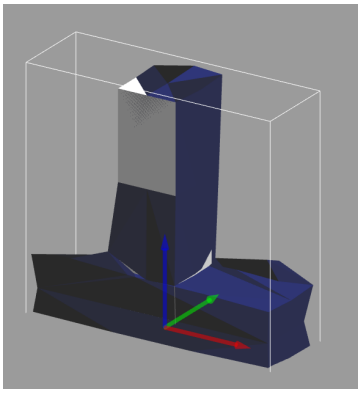
(b) Stone.



(c) Sphere.



(d) U-shape.



(e) ⊥-shape.

Figure 5.2: The shapes used as obstacles in the simulated training environment.

The UAV's motion is calculated by sampling a random relative reference yaw angle that is within the robot's field of view and a velocity in UAV's x-direction in the body frame. The UAV's relative yaw angle is adjusted to match the randomly sampled one and the randomly selected velocity is applied generate the motion.

The trajectory is given by the UAV's initial state, the relative reference yaw angle and the desired motion duration, which is set to 72 action steps. The desired velocity is $v_x = (1 \pm d)\frac{\text{m}}{\text{s}}$, where d is a random velocity between $[-0.25, 0.25]\frac{\text{m}}{\text{s}}$. The random velocity is added for greater variation in the data set, which helps the trained model generalize better. The length of the trajectory is also influenced by the velocity, as the number of actions in the trajectory is fixed. The algorithm for the motion primitive trajectory generation is presented in Algorithm 2. An assumption made is that the algorithm assumes acceleration to be achieved immediately with no regard for the time constant of the pitch dynamics. The robot's position at every trajectory generation is $p_0 = (0,0,0)$, as it is defined the robot's body frame. The action sequence that is fed into the network uses the relative yaw angle and is denoted $\text{commands}_{\text{evaluate}}$ in the aglortihm. The actions which are fed to the UAV uses the reference yaw angle and is denoted $\text{commands}_{\text{execute}}$.

Algorithm 2: Motion Primitive Trajectory Generation

Result: Motion Primitives

Draw a random yaw angle within FoV, ψ_{relative}.

Measure the robot's current yaw-angle ψ.

Compute the reference yaw-angle $\psi_{\text{reference}} = \psi_{\text{relative}} + \psi$.

Compute a forward velocity with a random disturbance:

$v_x = (1 \pm 0.25)\frac{\text{m}}{\text{s}}$.

Define the position, the velocity and the acceleration in body frame:

$p_0 = (0,0,0)^T$,

$a_0 = (0,0,0)^T$,

$v_0 = R_z(\psi_{\text{relative}}) \cdot (v_x, 0, 0)^T$, where R_z is the rotation matrix about the z-axis.

Initialize the empty states and the commands lists

Define the update frequency $t = 0$.

for $i \in$ number of steps before replan **do**

- $t += \frac{1}{15}$
- $p = \frac{a_0}{2} t^2 + v_0 t + p_0$
- $v = a_0 t + v_0$
- $a = a_0$
- states$[i] = (p, v, a)$
- commands$_{\text{execute}[i]} = (v_x, 0, 0, \psi_{\text{reference}})$
- commands$_{\text{evaluate}[i]} = (v_x, \psi_{\text{relative}})$

Data is saved every 0.4m traveled (referred to as a *keyframe*) or in the case of a collision. This is done to avoid redundant data in the data set. The action data is saved at every action executed as it is desired to inspect the whole action sequence between the keyframes. The data saved are the robot's states, the depth images, the actions executed, the collision labels, and the actions executed that correspond to a keyframe. The relevant states that are saved are the position, the velocity, the acceleration and the orientation. The collision label is converted to a 72-long list where the elements in the list correspond to the actions executed. The list contains zeros until the element which represents the action that caused a collision was executed. From this point, the list contains the ones. This is done so that the network can know which action in the action sequence that caused the collision. The states are saved, even though they are not explicitly used in the network. There are several reasons for saving the states, and the obvious being that if the model should be changed (for instance, by including some states as the input to the network to improve prediction accuracy), it should not be necessary to generate the data over, which can be rather time-consuming. Another reason is that it can be good for data analysis of the data set. The number of steps in the trajectory is 72. This is chosen based on the maximum depth

range, the maximum velocity and the trajectory step frequency. The trajectory step frequency is set to match the depth image frequency of 15Hz. The number of action steps in the trajectory is calculated as:

$$\frac{6\text{m}}{1.25\text{m/s}} \cdot 15\text{fps} = 72. \tag{5.1}$$

When all the self-labeled data has been generated, they are processed to TensorFlow Records data type (TF Record) so that they can be used in the network model created with TensorFlow. However, before doing this, the action sequence list must be fixed. After a collision has occurred, there are no recorded actions to fill the rest of the 72-long action sequence list. This is an issue as the network requires a fixed action sequence length during training. This issue is fixed in the post-processing script of the data generation. As the rest of the 72-long actions sequence list is empty (unless it collides after exactly 72 steps), random forwarded velocities (x-direction) and random yaw commands are filled into these empty spaces. The reason for appending these random actions is to force the network to predict the collision label ('1') after the real collision, no matter what are the remaining actions in the action sequence.

5.2 Collision Prediction Network Model

The collision prediction network created during this thesis determines the collision probability for each action in an action sequence given the robot's current depth image and some planned trajectory's action sequence that is within the robot's FoV. The trajectory is represented by an action sequence where each action consists of a forward velocity (x-direction) and a yaw angle, where both are expressed in the robot's body frame. The depth image input has a size of 480×270 pixels. Based on the current depth image and several possible trajectories within the robot's FoV, local motion planning for navigating the cluttered environment is achieved based on the model evaluating different trajectories and choosing the one with the lowest average probability of a collision. The network architecture is illustrated in fig. 5.3 and is inspired by the work done by Kahn et al. [37]. The convolutional neural network used in this thesis is inspired by the CNN used by Loquercio et al. [38]. This CNN builds on the ResNet-8 architecture proposed by He et al. [75]. The CNN which is used in this thesis is illustrated in fig. 5.4. The optimizer used in the model is the Adam optimizer [76] with a learning rate of $1 \cdot 10^{-3}$ and a decay rate of $1 \cdot 10^{-5}$.

In order for the model to handle the noisy depth images from the depth camera, online filtering is required. The filtering algorithm implemented is inspired by the work done by Ku et al. [77]. In this thesis, however, the kernel size used in the dilation is increased several times to handle the depth image's characteristic noise. This noise often has large blind spots caused by reflecting surfaces like windows or water surfaces. A trade-off with increasing the kernel size is that smaller objects will appear larger and the navigation in tight environments might be

worse. Given the perfect depth measurements in the simulation, the filtering is not done and the input to the CNN the raw depth image. Different evaluation studies with and without noise and filtering are done in Chapter 6.

The action horizon input to the network consists of 18 actions, which corresponds to every fourth action in the 72-long action sequence described in the previous section. The input horizon is shortened because a 72-long input horizon would be too long for the LSTM to handle properly. By taking every fourth action, the distance represented by the trajectory would still be the same, meaning that the collision prediction would still be able to predict collisions that are further away. Taking every fourth action will also remove much of the redundant data in the action sequence that is fed into the LSTM.

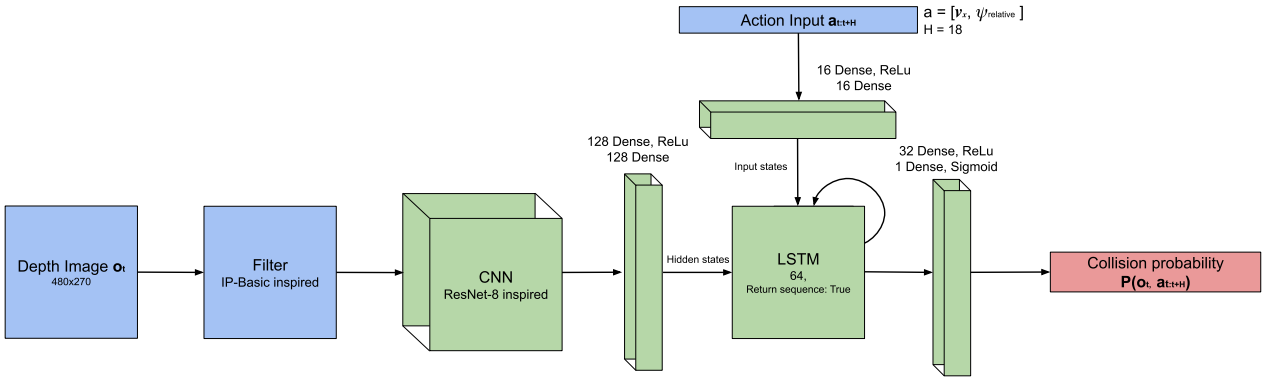
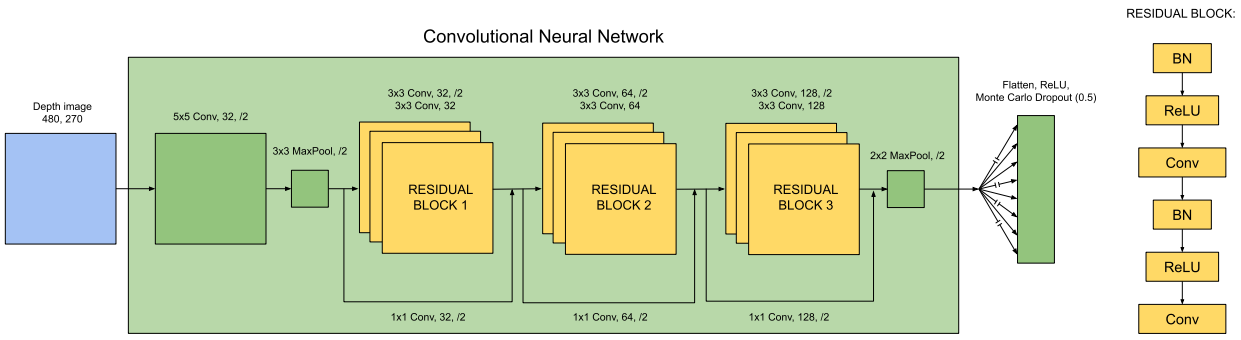


Figure 5.3: An illustration of the deep collision prediction network developed in this thesis. This model is proposed as a means to perform local motion planning for learning-based navigation using a depth image and an action sequence as inputs. The depth images are filtered based on the IP-basic filtering algorithm [77] and the CNN architecture is inspired by the CNN used by Loquercio et al. [38]. The action sequence input has an 18-long horizon and consists of a forward velocity (x-direction in the robot frame) and a yaw angle that is within the robot's field of view.



(a) (b)

Figure 5.4: Illustration of the CNN used in the collision prediction network. (a) is the entire CNN and how it fits in the network. The shared part of the architecture consists of a ResNet-8 with three residual blocks. (b) is a closer illustration of the residual blocks, where BN means "batch normalization". ReLU is the activation and the kernel size and the stride for the Conv-layers are specified in (a). The notation in the figure refers to the first value being the kernel's size, the second being the number of filters, and the third being the stride if it is different from 1.

5.2.1 Accounting for Imbalanced Data Set

The data set generated consists of 60.000 episodes, where 50.000 is used as training data and 10.000 is used as validation data. A total of 4.044.438 actions were executed in the data set, where 655.930 of the actions are associated with a collision and 3.388.508 are not. In other words, 16.22% of all actions executed are associated with collisions and 83.78% actions are not. This is a rather skewed data set, as discussed in Section 3.1.3 and in Section 3.2.4, and would bring some issues like high misclassification errors for the minority class, which in this case means "collision". This is not desired because if not handled properly, the model might easier predict "no-collision" than "collision" as it is more likely given the data set. Two approaches have been used to address this in the thesis: i) adjusting the output bias and ii) include class weights. The output bias is given as (where "pos" refers to "collision" and "neg" refers to "no-collision"):

$$\text{bias}_{\text{out}} = \log_e\left(\frac{n_{\text{pos}}}{n_{\text{neg}}}\right) = \log_e\left(\frac{655.930}{3.388.508}\right) \approx -1.64, \tag{5.2}$$

and the class weights are:

$$w_{\text{neg}} = \frac{1}{n_{\text{neg}}} \cdot \frac{n_{\text{total}}}{2} = \frac{1}{3.388.508} \cdot \frac{4.044.438}{2} \approx 0.597, \tag{5.3a}$$

$$w_{\text{pos}} = \frac{1}{n_{\text{pos}}} \cdot \frac{n_{\text{total}}}{2} = \frac{1}{655.930} \cdot \frac{4.044.438}{2} \approx 3.08. \tag{5.3b}$$

The equations are closer discussed in section 3.2.4 and presented in eq. (3.18) and eq. (3.20). A custom loss function that accounts for the class weights has been created. In addition, custom

training and validation functions have also been created.

5.3 Local Motion Planning for Autonomous Navigation

There are several methods of utilizing the collision prediction network to perform local autonomous navigation. The method used in this thesis is to sample several random trajectories in the robot's FoV, evaluate each of them with the current depth image and choose the one that is the least likely to collide with the environment. The trajectory that is the least likely of a collision could either be chosen as the trajectory with the lowest average probability or the trajectory whose maximum collision probability value is the lowest across the evaluated trajectory set. Other methods like prioritizing earlier collision probabilities with some exponential function or similar are possible, yet not explored much in this work. The same trajectory generation method that was used in the data generation is reused when sampling the random trajectories. Therefore, every trajectory is 72 action steps long, where every fourth action step is fed into the network. Every trajectory is fed through the network several times, and the average collision probability is used when evaluating the trajectories. In this work, the best trajectory is defined as the trajectory with the lowest average collision probability, and the algorithm for finding this is shown in Algorithm 3.

Algorithm 3: Motion Planner Algorithm

Result: Best current trajectory

Get the current robot states and the depth image $\mathbf{s}_t$, $\mathbf{o}_t$

for $i \in$ number of paths to evaluate **do**

    Compute several random trajectories **trajectories**$[i] \leftarrow$ **compute_trajectory(**$\mathbf{s}_t$**)**

for trajectory $\in$ trajectories **do**

    Run the collision predictor several times and compute the average **predictor(**$\mathbf{o}_t$**, trajectory)**

    Compute the average probability for each trajectory **MEAN(collision probabilities)**

Find the index that corresponds to the lowest average probability **index_best**

Find the trajectory based on the best index **trajectories[index_best]**

Chapter 6

Evaluation Studies

The following sections will present the evaluation studies of the model's performance during training, the trained model's ability to perform local motion planning and the model's ability to navigating different types of cluttered environments. Lastly, the model will be tested with real-life depth images. The average computational GPU-time for the CNN-part of the network is 25ms and 7ms for the RNN-part (with 18 LSTM cells) when evaluating 1024 trajectories ten times each, using the Nvidia GeForce RTX 2060.

6.1 Model Performance

The model has trained over 500 epochs, meaning it has trained over the entire data set 500 times. The precision, the recall and the $F1$-score metrics for the training and for the validation are shown in fig. 6.1-6.4. Not surprisingly, the model's performance had more variation in the accuracy, recall, precision and loss during the validation phase, especially for the earlier epochs. This is because the validation data set contains unseen data and the model is not yet sufficiently generalized from the training data set at the early epochs.

As can be seen in fig. 6.1, the accuracy of the model during the training is a lot more consistent than during the validation. However, as the model trains, there is less variation in the accuracy of the validation data and the accuracy of the validation data becomes similar to the accuracy of the training data. This means that the model generalizes the categorization of true positives and true negatives of the validation data well. After 500 epochs, the training accuracy and the validation accuracy equal (0.990). It is worth noting that the training accuracy is rather high after a single epoch. This indicates that the training data set should be sufficiently large. However, this observation should not be mixed with the assumption that the training data set is sufficiently diverse. The similarity between the training accuracy and the validation accuracy indicates the sufficient diverseness of the training data set relative to the validation data set.

The recall, as seen in fig. 6.2, converges close to the maximum value after few epochs, indi-

cating that the model quickly learns to categorize false negatives. The training and validation recall closely follows each other during nearly all the 500 epochs. The recall value after 500 epochs is 0.979 for the validation data and 0.980 for the training data. This similarity is good because it suggests that the model generalizes the categorization of false negatives to the validation data well. After one epoch, however, the recall value is still high for the training data (0.934), while it is much lower for the validation data (0.477). This suggests that the model is not well-generalized after only one epoch, as is expected.

The precision values are shown in fig. 6.3. The precision metric (sometimes referred to as the positive predictive value) indicates that the model learns false positives as the epochs increase. The precision for the validation data has more variations than the precision for the training data, especially for lower epochs. This is expected as the model is learning new methods of categorizing false positives during training. The precision value for the validation data varies between $[0.673, 0.976]$ for all epochs. As the epochs increase, the precision becomes more stable and predictable. After 500 epochs, the precision values are 0.957 for the validation data and 0.956 for the training data, which means that the model generalizes its ability to categorize false positives well to the validation data set. The precision is especially spiky compared to the recall because precision measures the false positives and the recall measures the false negatives. There are more negative cases than positive in the data set, so the ability to categorize false positives is more difficult than learning to categorize false negatives.

The $F1$-score takes both the false positives and the false negatives into account and thereby represents the learning of an imbalanced data set better than either recall or precision measurements on their own. The $F1$-score is given in eq. (3.7) and shown in fig. 6.4. The $F1$-score for the validation data after the first epoch is 0.628 while it is 0.913 for the training data. After 500 epochs, it is 0.968 for the validation data and the same for the training data, 0.968.

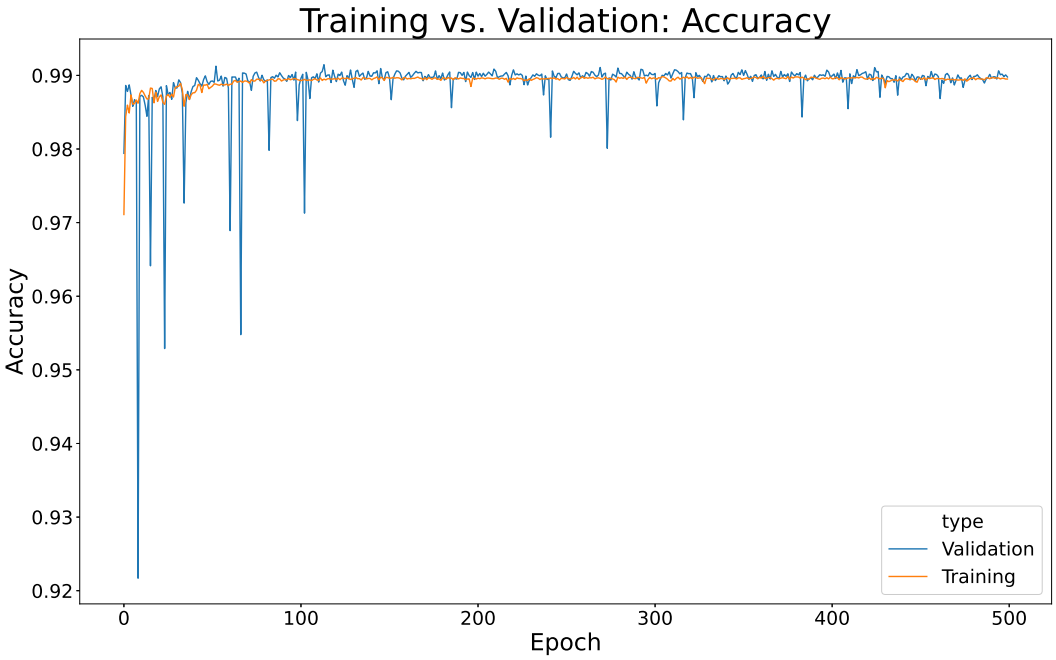


Figure 6.1: Accuracy for the validation data and the training data over 500 epochs.

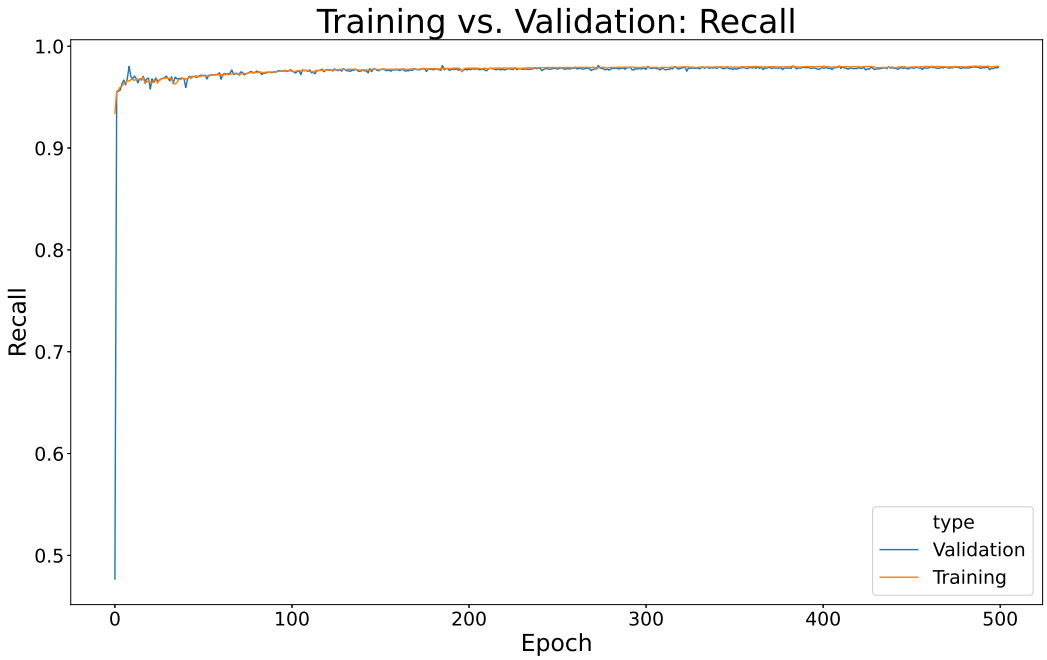


Figure 6.2: Recall for the validation data and the training data over 500 epochs.

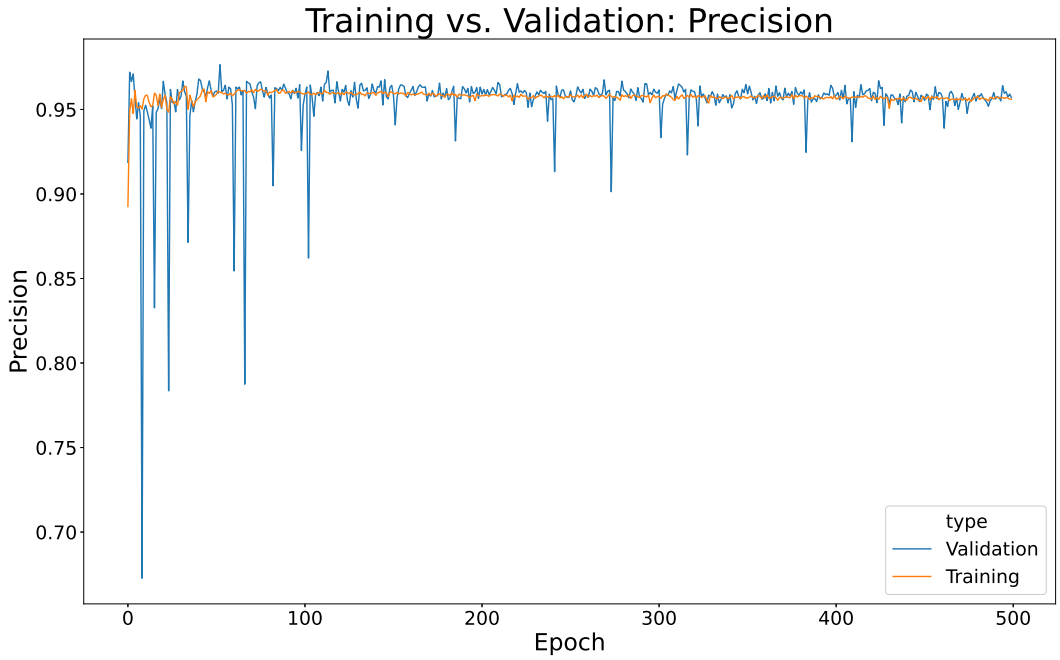


Figure 6.3: Precision for the validation data and the training data over 500 epochs.

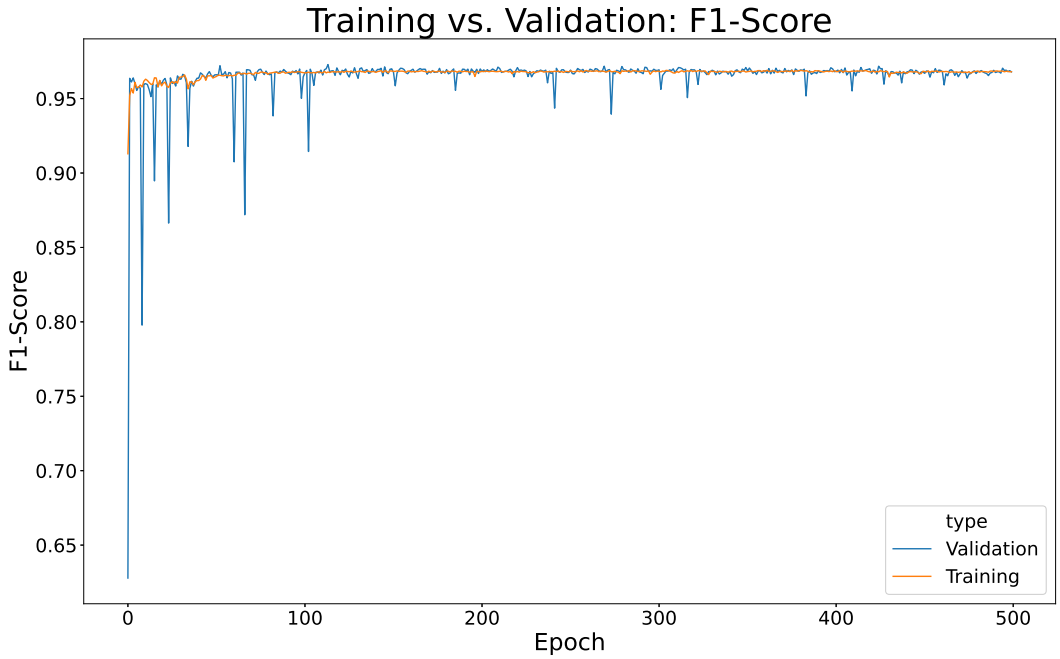


Figure 6.4: $F1$-score for the validation data and the training data over 500 epochs.

The training loss and the validation loss for the epochs are shown in fig. 6.5. The model

overfits slightly, as can be seen by the training loss still slowly decreasing while the validation loss is not. The validation loss after 500 epochs is 22.773 and the training loss after 500 epochs is 21.663. When the validation loss is significantly different from the training loss, as is the case during the spikes in fig. 6.5, the model on the current epoch has overfitted the training data and generalizes poorly to the validation data. The spikes of the validation loss gradually decrease in magnitude and frequency as the number of epochs increases, which indicates that the model is training well.

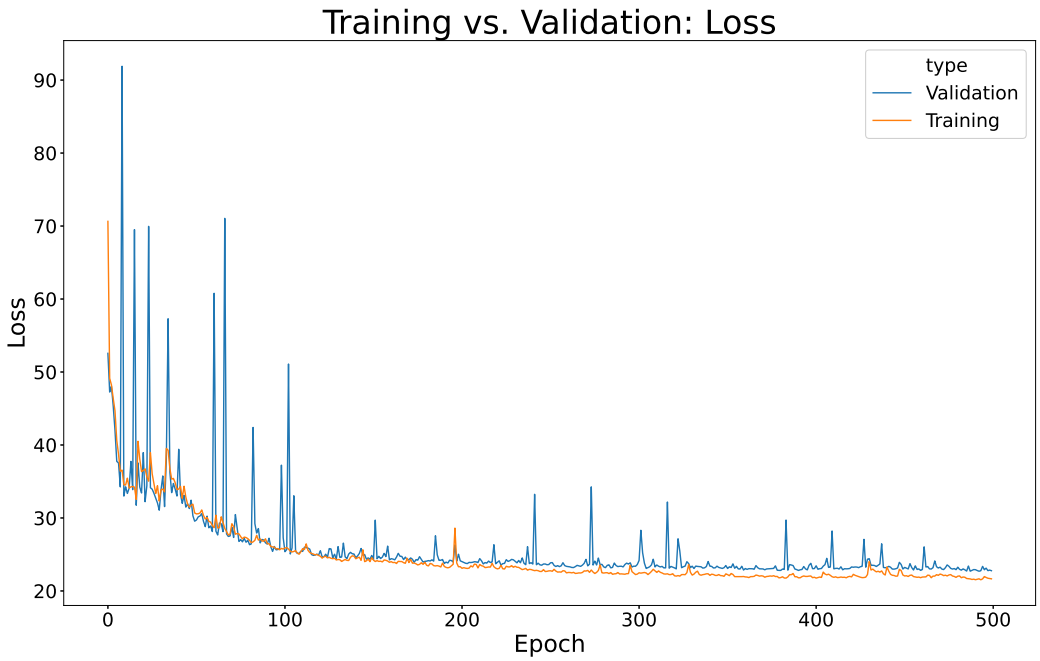


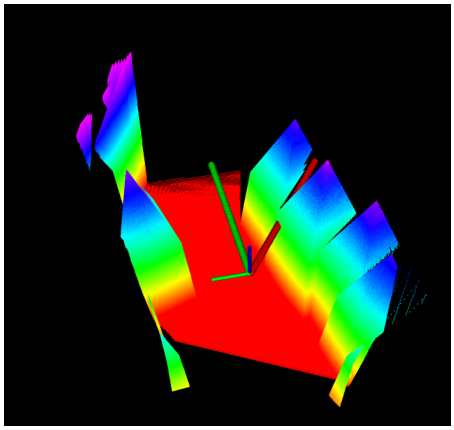
Figure 6.5: Loss for the validation data and the training data over 500 epochs.

6.1.1 Local Motion Planning

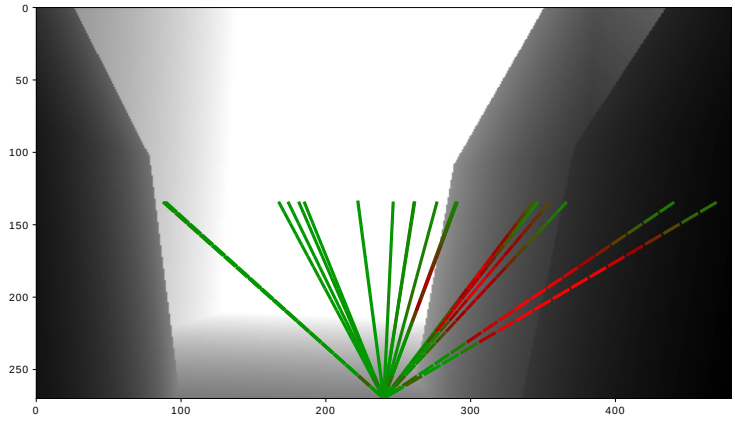
In this subsection, the model's ability to perform local motion planning will be presented by 3D images (obtained from Rviz [78]) of the drone with the evaluated trajectories and the evaluated trajectories displayed over the drone's current depth image. The model's trajectory evaluation in a tight environment can be seen in fig. 6.6. In this case, 20 trajectories are evaluated, which can be seen in fig. 6.6a. The best possible trajectory is colored green and the worst possible trajectory is colored red. As can be seen from the figure, the model evaluates the best trajectory as a collision-free path and the worst trajectory as a path with a certain collision into the obstacle to the right. All of the evaluated trajectories are displayed over the drone's current depth image in fig. 6.6b. Every fourth action in the 72-long action sequence is fed into the network, meaning a total of 18 actions fed into the network per trajectory. This is presented in the figure as 18 tra-

jectory segments for every trajectory. Each trajectory segment is colored based on the segment's collision probability, where red indicates a high collision probability and green represent a low collision probability. This allows a greater understanding of when the model predicts a collision and is illustrated more evidently in the top-down 3D figure in fig. 6.7a, where the same coloring scheme applies. The correlation between when in a given trajectory the model suggests a collision and the actual placement of the obstacles means that the model learns to connect a forward velocity and a yaw angle to a depth image measurement on its own and thereby learns a sense of time. Not only is it able to predict whether a collision would occur in a trajectory, but also when it would occur.

Interestingly, if fig. 6.6b is studied more in detail, one can observe that the collision probability is low after the model has predicted a certain collision. This phenomenon is closer discussed in Section 5.1, and is due to the random completion of the 72-long action sequence during the data generation that is used in the model training.



(a) The best trajectory colored green and the worst colored red.



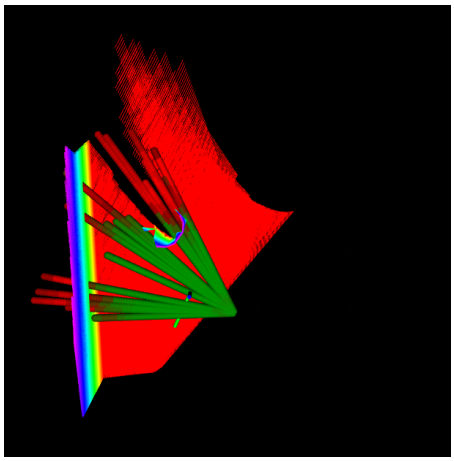
(b) The 20 random trajectories evaluated over the current depth image.

Figure 6.6: The model's evaluation of 20 random trajectories. (a) is a 3D-image showing the trajectory that the model evaluates as the least likely to cause a collision colored green and the trajectory with the highest probability of collision colored red. (b) shows all the 20 random trajectories evaluated over the drone's current depth image. The color of the trajectory segments represents the collision probability of every fourth action in the 72-long action sequence.

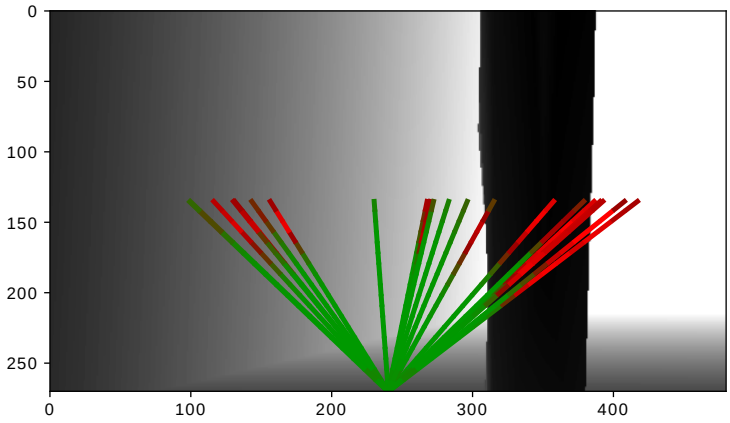
It can be slightly challenging to see in fig. 6.6 that the evaluated trajectories also have an average velocity between $[0.75, 1.25]\frac{\text{m}}{\text{s}}$, which affects the trajectory's length. This can be more easily seen in the top-down view in fig. 6.7a, where the best trajectory's length is clearly shorter than the worst ones. As shown in the figure, the shorter trajectory is advantageous in the current situation. It can also be seen that the drone has started moving with this trajectory. All trajectories have the same number of steps, 72, which is why the length of the trajectories represents the trajectory's velocity. The different trajectory velocities are not as clearly seen in fig. 6.7b, which

is the corresponding depth image view for fig. 6.7a.

The trajectories over the depth image are plotted as they are because it is necessary to transform the 3D trajectory to a 2D plot that can be placed over the depth image and clearly display the different trajectory angles. It can be argued that the length of the trajectories could be shortened as well, but this is not done as it could give a false indication of where the trajectory is heading as the trajectory, in reality, is into the plane. The different trajectory velocities are the leading reason why trajectories that have approximately the same angles have different collision probabilities. This can be seen in fig. 6.7b. The trajectory to the left of the image has a lower collision probability while heading towards the wall like the trajectories nearby. If this is studied in relation with fig. 6.7a, it can be seen that the specific trajectory's velocity makes it sufficiently short enough, so it barely reaches or even fail to reach the wall. In contrast, the trajectories with similar angles have high velocities, and collision with the wall is likely.



(a) Different trajectory lengths for the 20 evaluated paths.



(b) The 20 random trajectories evaluated over the current depth image.

Figure 6.7: The model's evaluation of 20 random trajectories where it is highlighted that the paths have different velocities and thereby lengths. (a) is a top-down view of the paths in 3D, showing that the model chooses the shortest path. (b) is the corresponding depth image with the 20 trajectories.

6.1.2 Model Activation

Gradient-weighted class activation mapping (Grad-CAM) [79] is used to visualize the model's last CNN-layer's activation. Grad-CAM is a "visual explanation" for decisions from a large class of CNN-based models, making them more transparent. In short, the Grad-CAM uses the gradients of the target class into the last CNN-layer (which is the max pool layer before the flatten layer in fig. 5.4) to produce a coarse localization map highlighting the important regions in the image for predicting the concept. In the case of the collision prediction model used in this the-

sis, this would essentially mean: "Which areas of the depth image causes a high collision probability given the depth image and the action sequence". The heatmap has a resolution of (8,5), which matches the resolution of the output layer of the CNN. The low resolution compared to the depth image (which has a resolution of (480,270)) is the reason why the heatmap is coarse.

Fig. 6.8 shows the model activation for a depth image and the trajectory that the model evaluates as the most likely to cause a collision. The depth image has a pyramid-shaped obstacle close to the center and a wall to the right. The trajectory deemed the worst is the one which is heading straight into the pyramid-obstacle. This means that the area around the pyramid will significantly impact the collision prediction model more than other areas, given the action sequence. Interestingly the ground and the wall are also highlighted as possible collision areas. Fig. 6.9 shows the model activation for a depth image and the trajectory that the model evaluates as the least likely to cause a collision. As can be seen in the figure, there is no activation in the CNN's last layer. This means that there is no (or very low) probability of colliding with obstacles in the depth image given the action sequence.

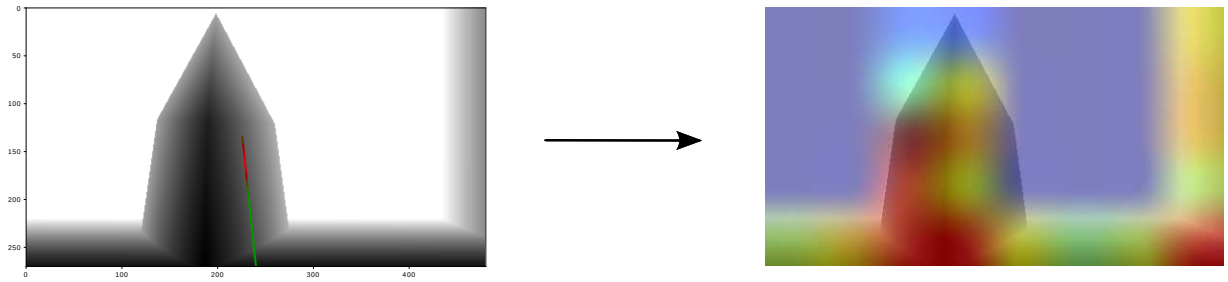


Figure 6.8: Grad-CAM visualization of the trajectory that the model evaluates as the most likely to cause a collision.

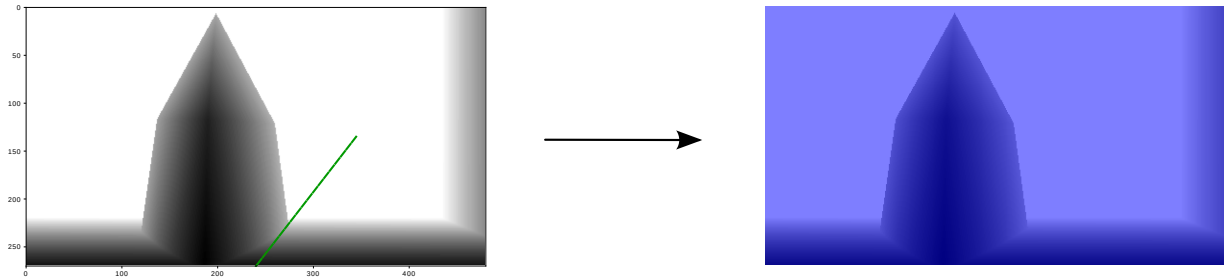


Figure 6.9: Grad-CAM visualization of the trajectory that the model evaluates as the least likely to cause a collision.

Other Grad-CAM visualizations of the worst and best trajectories are given in fig. 6.10 and fig. 6.11. In this case, the drone is close to a wall to its left and an obstacle is within the field of

view in the middle of the depth image, but at a distance. The worst trajectory crashes into the wall to the left. This can be seen as a dark red spot in the corresponding heatmap, indicating that a collision in this area is very likely. Worth noting is that the model also ignores the object in the middle of the depth image with the best trajectory. In this case, the model is still affected by the wall to its left. This is probably because the wall is so close that it would affect the model regardless of the trajectory. The best trajectory is in a collision-free path but close to the obstacle. Possible reasons why there is no activation for the obstacle in the middle could be that (i) the trajectory is sufficiently short so that it ends before the obstacle or (ii) the model cares little about how great of a margin it passes an obstacle with as long as it is not colliding with it. The latter reason is more discussed in Chapter 7.

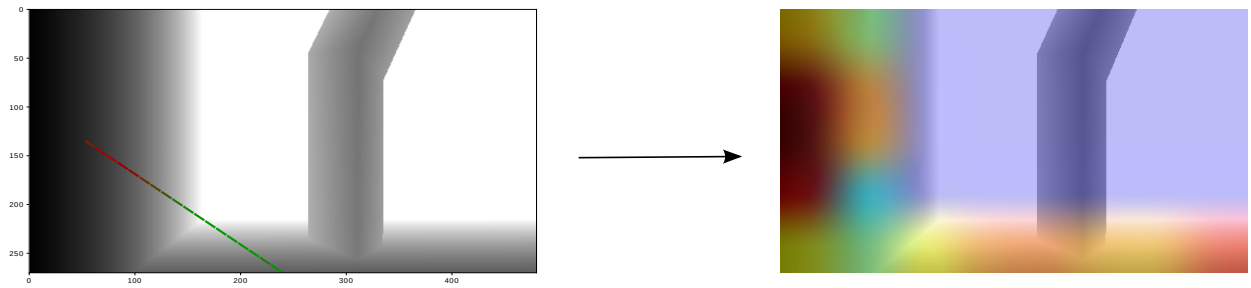


Figure 6.10: Another Grad-CAM visualization of the trajectory that the model evaluates as the most likely to cause a collision.

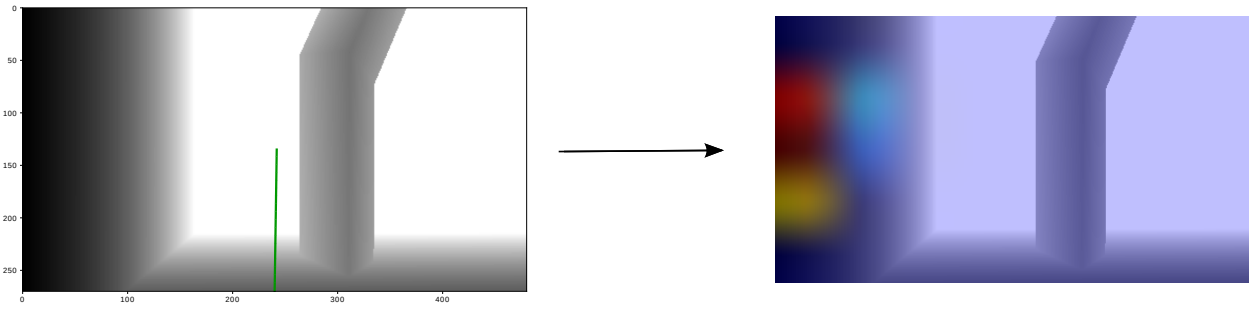


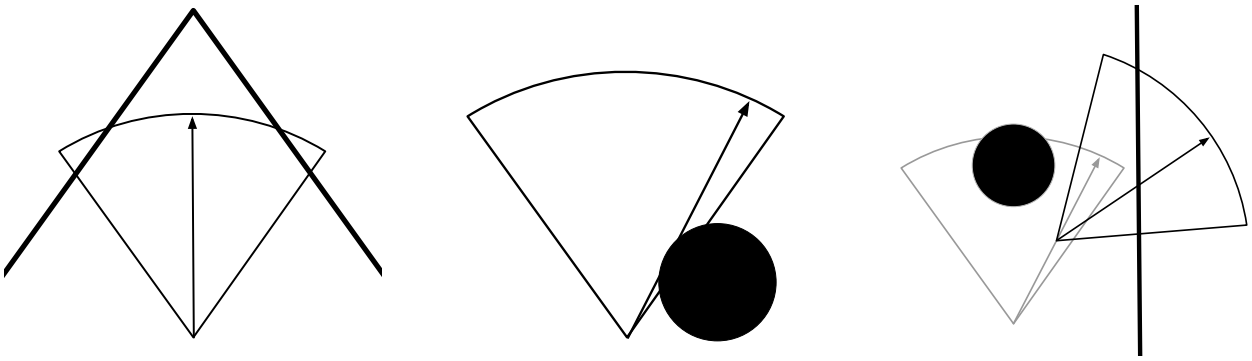
Figure 6.11: Another Grad-CAM visualization of the trajectory that the model evaluates as the least likely to cause a collision.

6.2 Navigating in Simulated Cluttered Environments

In this section, the model will be used to enable collision-free navigation in cluttered environments. There are no waypoints or other extrinsic goals for the drone to achieve, only an intrinsic motivation of not colliding with the environment. Therefore, the trajectories chosen will be

solely based on having the lowest average collision probability at the given time. The following subsections will present the model's performance in different types of environments. Each environment is tested over 50 episodes. Time-of-flight, distance traveled and the number of action steps executed are recorded for each episode and presented for each study. The drone has a fixed height of 2m above the ground and the (x, y, z)-initialization at each episode is specified for each given environment. The model evaluates 1024 possible trajectories within the robot's FoV and each action sequence is fed through the network 10 times to calculate the average collision probability. The eight first actions in the 72-long action sequence are executed, imitating a rollout scheme. Every episode has an upper time limit of 125s, and when the time-of-flight has passed the time limit, the episode is finished.

There are three main reasons for collisions: (i) the drone does not see the end of a corner due to its fairly short depth image range (6m), making the model believe that the end of the corner is the best path. When the drone first discoverers the end of the corner, it does not have the necessary space to maneuver out of the situation, as every trajectory has a minimum of $0.75\frac{\text{m}}{\text{s}}$ forward velocity and a yaw angle between $[-43°, 43°]$. These collisions are referred to as *corner collisions* and are possibly strongly underrepresented in the training data, making them hard for the model to discover. (ii) the drone changes its course to avoid an obstacle, but as the drone passes by the obstacle, it replans its trajectory with a new depth image. In this depth image, the obstacle is barely visible in the side-view of the drone's limited FoV (86°) and is possibly lost in the processing of the depth image, making it invisible to the drone. Therefore, the model will sometimes choose a trajectory that barely scratches an obstacle as the drone is passing by. These types of collisions are referred to as *pass-by collisions* and are a consequence of only using the current depth image in the network and not a depth image sequence where previous depth images are also considered. (iii) the drone successfully maneuvers away from an obstacle with a sharp turn, but as the depth image updates, the drone finds itself in a situation it cannot maneuver out from due to the momentum it gained from the previous sharp turn. These types of collisions are referred to as *blind collisions*. The three different types of collisions are illustrated in fig. 6.12. The different types of collisions and the trade-off between computational efficiency and better situational awareness will be discussed in greater detail in chapter 7. Other types of collisions also occur, such as objects being too small for the model to detect or other cases where the drone suddenly finds itself in a situation it cannot maneuver out from.



(a) Corner collision. (b) Pass-by collision. (c) Blind collision.

Figure 6.12: The three most common types of collisions, where the drone's FoV and movement are shown. (a) illustrates a typical corner collision where a collision is inevitable, (b) illustrates a pass-by collision where the model scratches an obstacle as it is passing by as it does not see it and (c) illustrates a blind collision where the previously rapid turn caused the drone to get into an inevitable collision which it could not predict with the prior depth image. The reason the last trajectory in (c) is not all the way to the left in the FoV, as the model's proposed trajectory probably would be, is because of the drone's momentum from the previous turn. The momentum makes it impossible to change the drone's course sufficiently fast to escape the collision.

To distinguish the different reasons for an episode to end, each episode is marked as either *reset*, *other* or *time out*. "Reset" occurs when the model is almost certainly going to collide with the environment, which in most cases is due to an incoming corner collision. If the best trajectory has an average collision probability of $> 20\%$ the episode is stopped and marked as reset. The reset threshold of 20% is found by trial and error and does not guarantee that a corner collision is happening but catches corner collisions in most cases. "Other" is for every other type of collision, which is mainly caused by pass-by collisions or blind collisions. In some cases, these collisions are also corner collisions but are not marked so because the collision probability is barely below the reset threshold. An episode is marked as "time out" if the episode lasts 125s, which is the upper time limit.

In the following subsections, different evaluation studies of the model will be done. The first study presented is navigation in cluttered environments with different levels of obstacle density. The objects and the environment will be the same as used when generating data for training, meaning the environment and the obstacles are known. Perfect depth image measurements are assumed and no filtering is therefore done. The second study introduces unknown objects that are more "real-life"-like into the familiar corridor environment. Unknown objects refer to objects not used during training. This is done to evaluate how well the model generalizes with obstacles it has never seen before. During this particular study, filtering the depth image will also be used to generalize the obstacles to simpler shapes and smoothing their details. In the third evaluative study, there will be an eight-shaped unknown environment with known obsta-

cles. This study is interesting because it is desired to look at the performance of the model in an environment that allows more yaw-movement than the corridor environment. In the fourth evaluative study, measurement noise will be added to the depth image and filtering will be applied. The model performance with perfectly simulated depth images, noisy depth images and filtered depth images is presented and commented on briefly.

6.2.1 With Different Degrees of Obstacle-Density of Known Obstacles

In this study, the model will be evaluated in the same 10m × 100m corridor environment and with the same obstacles that were used during the model training. Three difficulty levels will be evaluated where the density of the obstacles determines the degree of difficulty. The three difficulty levels are hereby referred to as "easy", "medium" and "hard", where each level has an increased obstacle density. The environments of the three levels can be seen in Appendix C.1. The paths that the drone chose at each episode for the three different environment difficulties can be seen in fig. 6.13-6.15.

The drone is initialized in the corridor every episode at $(x, y, z) = (-50, 0, 2)$ (using the right-hand convention) and with no orientation. The corridor does not have walls at either of the ends. This will affect some of the "time out"-marked episodes, as some of the episodes will spend the last few meters outside of the environment in empty space. However, in most cases, this is only for some few meters as the "time out"-time is 125s, the corridor is 100m long, the forward velocity is $(1 \pm 0.25)\frac{\text{m}}{\text{s}}$ and the drone is not moving in a straight line. The latter reason is especially valid for more cluttered environments, like in fig. 6.15c compared to fig. 6.13c. In fig. 6.14c and fig. 6.15c the drone does a U-turn once and goes outside the corridor. This type of time-outs affects the results more than if the drone escapes the corridor at the other end. The corner collisions are especially easy to notice in fig. 6.14a, where the episodes are marked as "reset" and finished due to the high collision probability.

The metrics for the easy, the medium and the hard environments are shown in Table 6.1. The easy environment has the longest time-of-flight, the longest distance traveled and the most action steps executed, while the hardest environment has the lowest in all metrics. The metrics for the easy environment and the medium environment are similar, but the easy environment has more "timed-out"-episodes than the medium one, with 72% and 48% respectively. This indicates that the medium environment, which had an average time-of-flight of 106.46s±23.73s, usually crashed or was reset not long before the time out would occur. Nearly all of the collisions for the easy environment are either pass-by collisions or blind collisions. Two blind collisions triggered the model to reset for the easy environment, as can be seen in fig. 6.13a. The episodes for the easy and the medium environment also have lower standard deviation than for the hard environment, meaning that the results were more consistent for the easy and the medium environment.

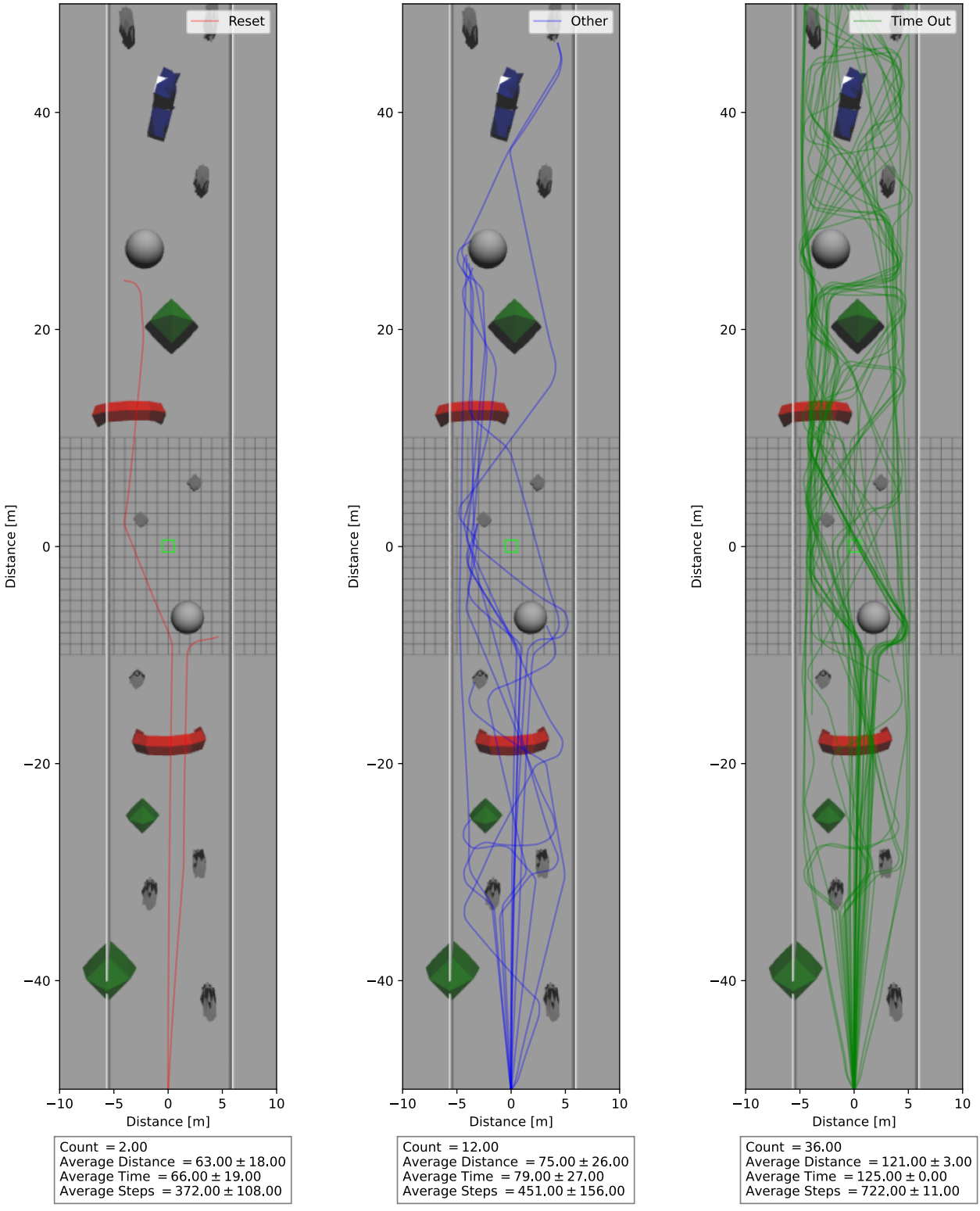
The medium environment had eight episodes that triggered a reset, where all were caused by corner collisions. The majority of the episodes in the medium environment were still marked as "time-out". The collision under the "other"-category were mainly pass-by collisions or blind collisions.

The hard environment had the lowest time-of-flight, distance traveled and action steps executed and with the highest standard deviation, which was as expected as the environment was severely cluttered. However, the drone managed to time out six times, where the episode never reached outside of the environment boundaries due to the heavily cluttered environment. The drone had the most collisions quite early on in this environment. This affected the high standard deviation, as it often either managed to pass the first cluttered area in the beginning and then continued for a while or crashed in this area. The model also triggered reset eight times in this environment, where most of the cases were due to corner collisions.

Table 6.1: Average time-of-flight, distance traveled and action steps executed with standard deviation (denoted std) and the collision type distribution for the easy, the medium and the hard environments.

Environment	Time ± std [s]	Distance ± std [m]	Steps ± std	Reset	Other	Time Out
Easy	111.71 ± 25.50	107.86 ± 25.44	643.12 ± 150.67	4%	24%	72%
Medium	106.46 ± 23.73	102.99 ± 23.60	624.40 ± 143.07	16%	36%	48%
Hard	51.59 ± 40.83	49.46 ± 41.50	305.92 ± 249.08	16%	72%	12%

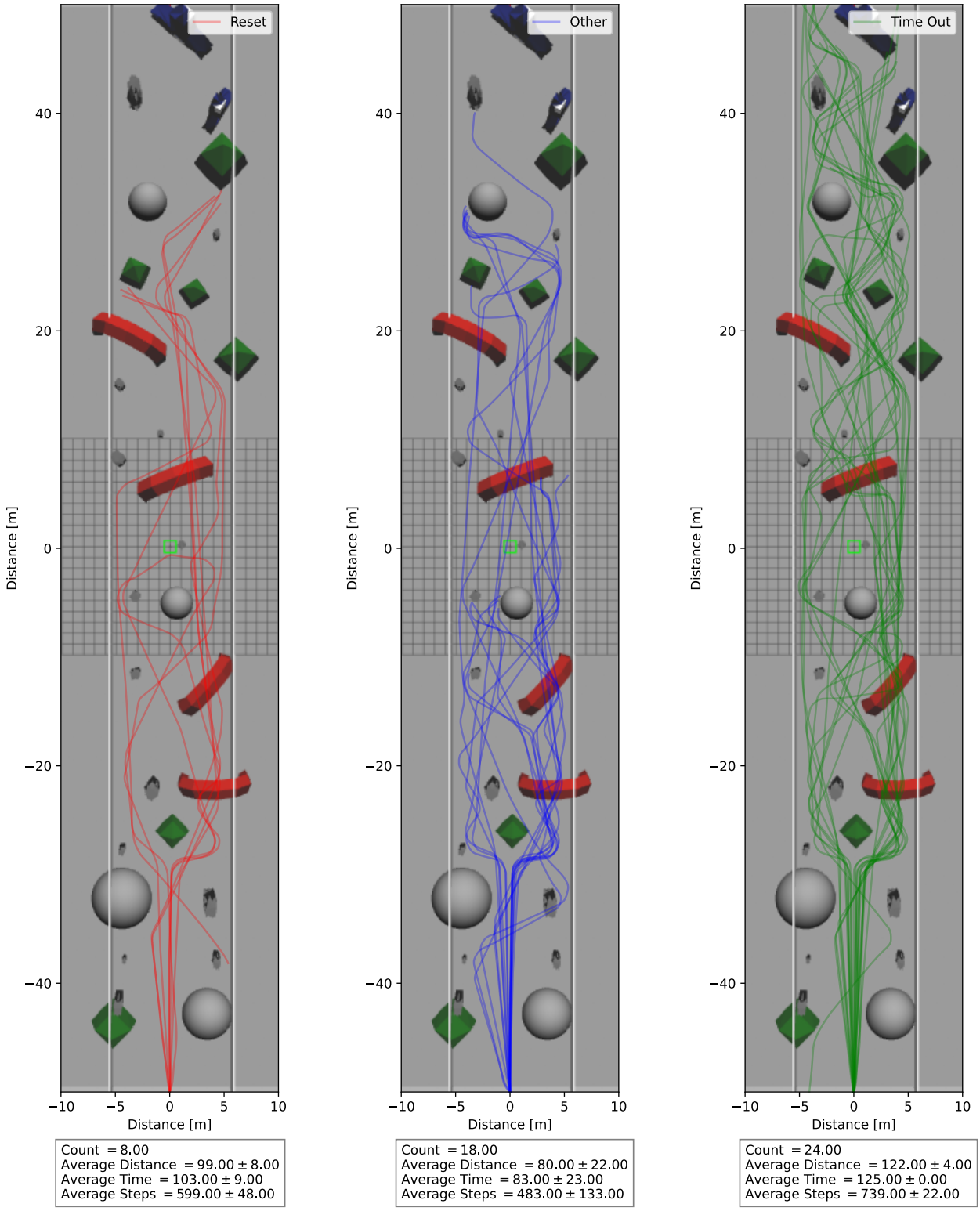
Easy



(a) Reset. (b) Other. (c) Time out.

Figure 6.13: 50 episodes with easy difficulty

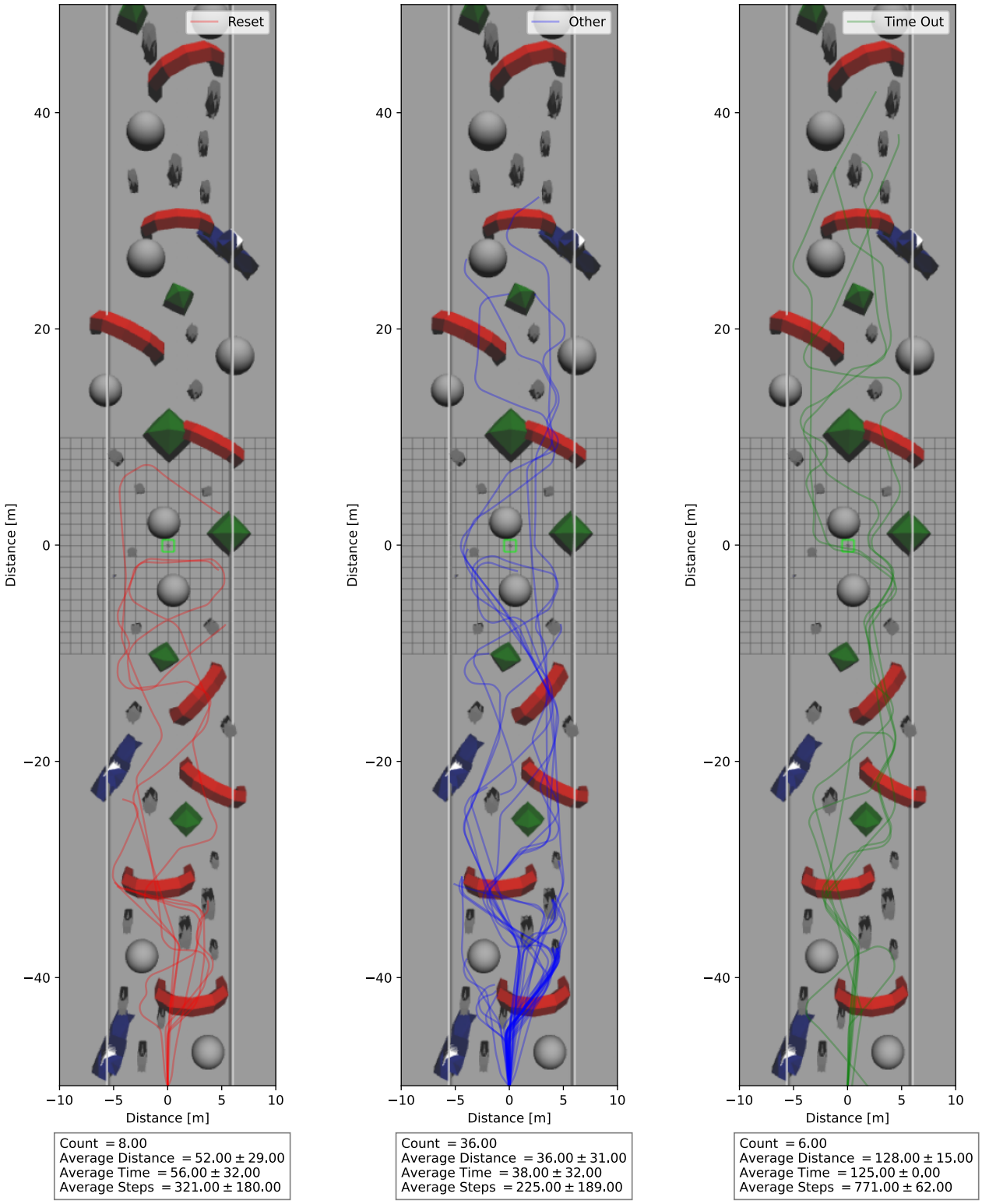
Medium



(a) Reset. (b) Other. (c) Time out.

Figure 6.14: 50 episodes with medium difficulty.

Hard

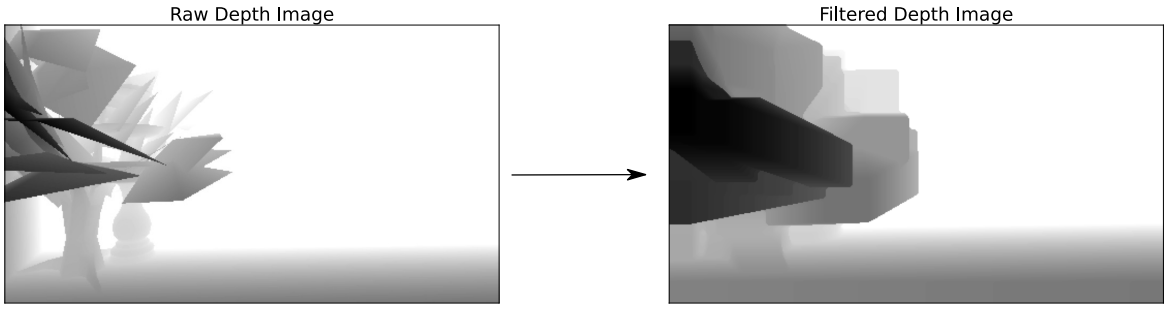


(a) Reset. (b) Other. (c) Time out.

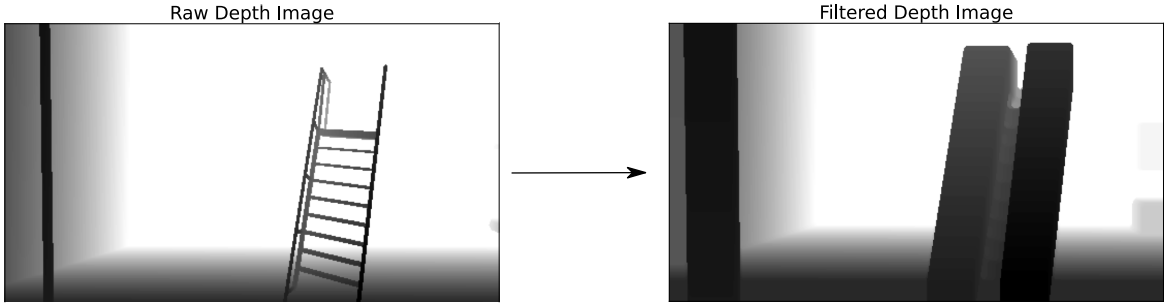
Figure 6.15: 50 episodes with hard difficulty.

6.2.2 With Unknown Obstacles

In this evaluative study, the familiar 10m × 100m corridor environment will be used, but with obstacles that are unknown to the model. The obstacles in this environment are closer to obstacles found in real life and with more details than the known obstacles. The unknown obstacles used in this study are trees, water fountains, mailboxes, ambulance cars, fire hydrants, dumpsters, trashcans, ladders, lamp posts and jersey barriers. The objects have different sizes and orientations. All objects are taken from `http://models.gazebosim.org/`. The environment can be seen in Appendix C.2. The environment is evaluated twice, once with raw depth image input and once with a filtered depth image input. The filtering applied is similar to the work presented by Ku et al. [77] and consists of dilation, morphology and blurring, but the full-kernel's size used in the dilation is increased to (42×42). The reason for introducing the filtering with increased dilation is to simplify the shapes of the obstacles. Typical shapes in this environment that the model especially struggles with are leaves on the trees and thin objects like ladders, lamp posts and so on. The depth images before and after applied filtering are shown in fig. 6.16. The specific CPU-time (using Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz) of the filtering depends on the depth image but averaged 5.46ms with a standard deviation of 6.58ms, suggesting there is a strong variation in the CPU-time required.



(a)



(b)

Figure 6.16: Filtering applied to depth images with unfamiliar obstacles. (a) shows filtering with large dilation of a tree structure and (b) shows filtering with large dilation of a ladder and a pole.

The average time-of-flight, distance traveled and action steps executed with standard devia-

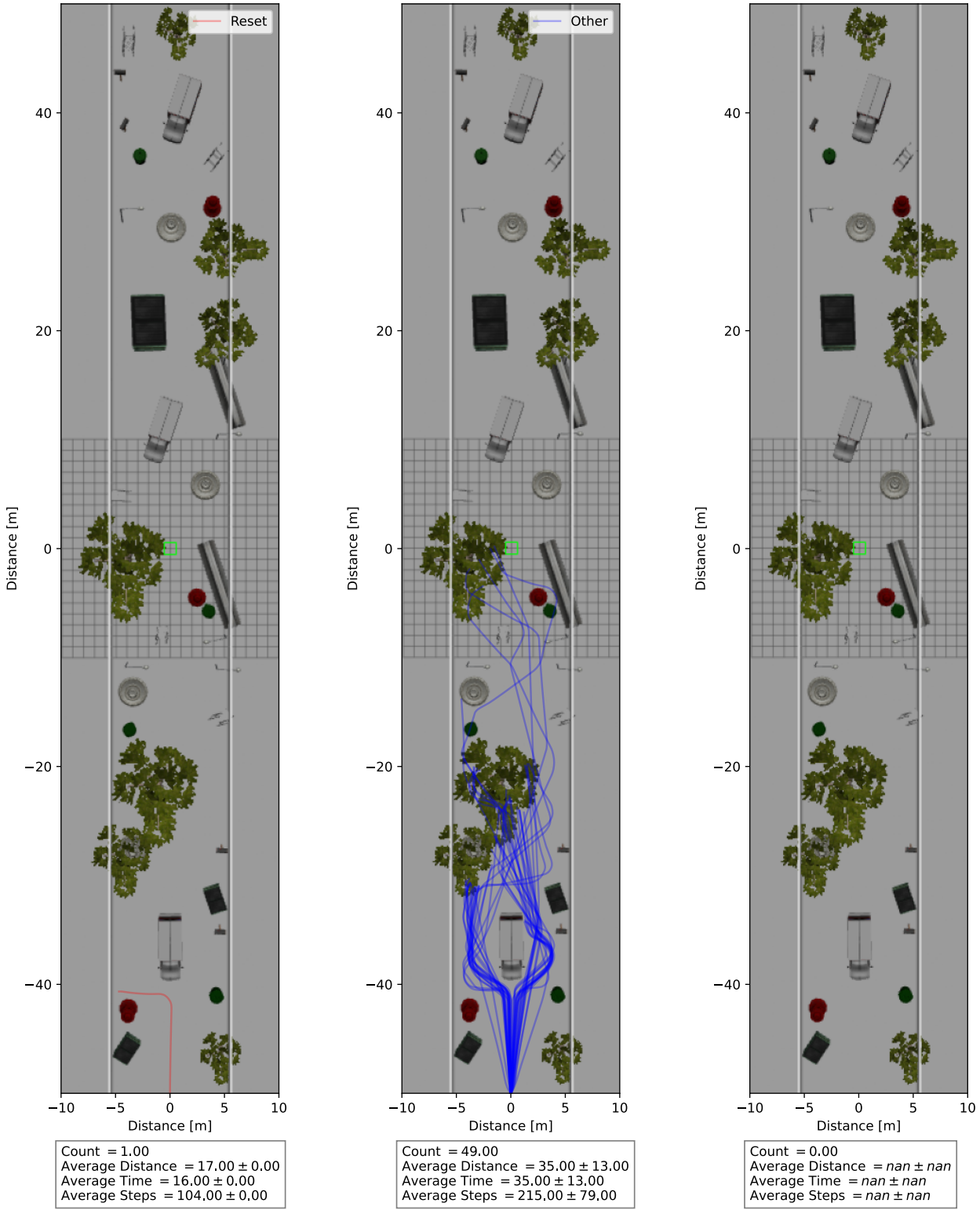
tion for the 50 episodes with unknown obstacles in a known environment, as well as the collision type distribution for unfiltered and filtered depth images, can be seen in Table 6.2. The results are better with the filtered depth images, but not by a great margin. It is necessary to inspect the paths in the episodes to see what caused the collisions to better understand the model's behavior. The results of the 50 episodes with unfiltered depth images can be seen in fig. 6.17 and the results of the 50 episodes with filtered depth images can be seen in fig. 6.18. As expected, the model struggles with tree structures and specifically the tree's leaves and branches. Nearly all collisions with the raw depth images are either caused by a collision with the first or the second tree, as can be seen in fig. 6.17b. The first tree has a high collision rate, meaning if the drone is on a collision course with this tree, it does not change its course. An interesting observation is that the model did not reset itself due to a high collision probability caused by the trees. In fact, the collision probabilities were low even though a collision with the trees was inevitable. This is because the model does not know how to deal with this type of depth images, as it has never seen anything close to it before. It did not have issues navigating around the ambulance, but that was expected as its shape is not as complicated as the trees'. It had zero time-outs and triggered a reset once due to a likely collision with the wall.

With the filtering is applied, the drone manages to get past the first tree almost every time but struggles with the larger trees, such as the second and the third tree. This is because their leaves and branches reach further, making the filtered depth image more diffuse for the model. When the tree is sufficiently small, it appears more like a "solid block", as displayed in fig. 6.16a. Reset also occurs more frequently and are caused by likely collisions in the filtered tree structures.

Table 6.2: Average time-of-flight, distance traveled and action steps executed with standard deviation (denoted std) and the collision type distribution when including unfamiliar objects in the environments for unfiltered and filtered depth images.

Environment	Time ± std [s]	Distance ± std [m]	Steps ± std	Reset	Other	Time Out
Unknown obstacles	34.47 ± 12.75	34.32 ± 12.98	213.26 ± 79.43	2%	98%	0%
Unknown obstacles (filtered)	45.27 ± 20.87	46.10 ± 21.87	287.46 ± 134.31	26%	74%	0%

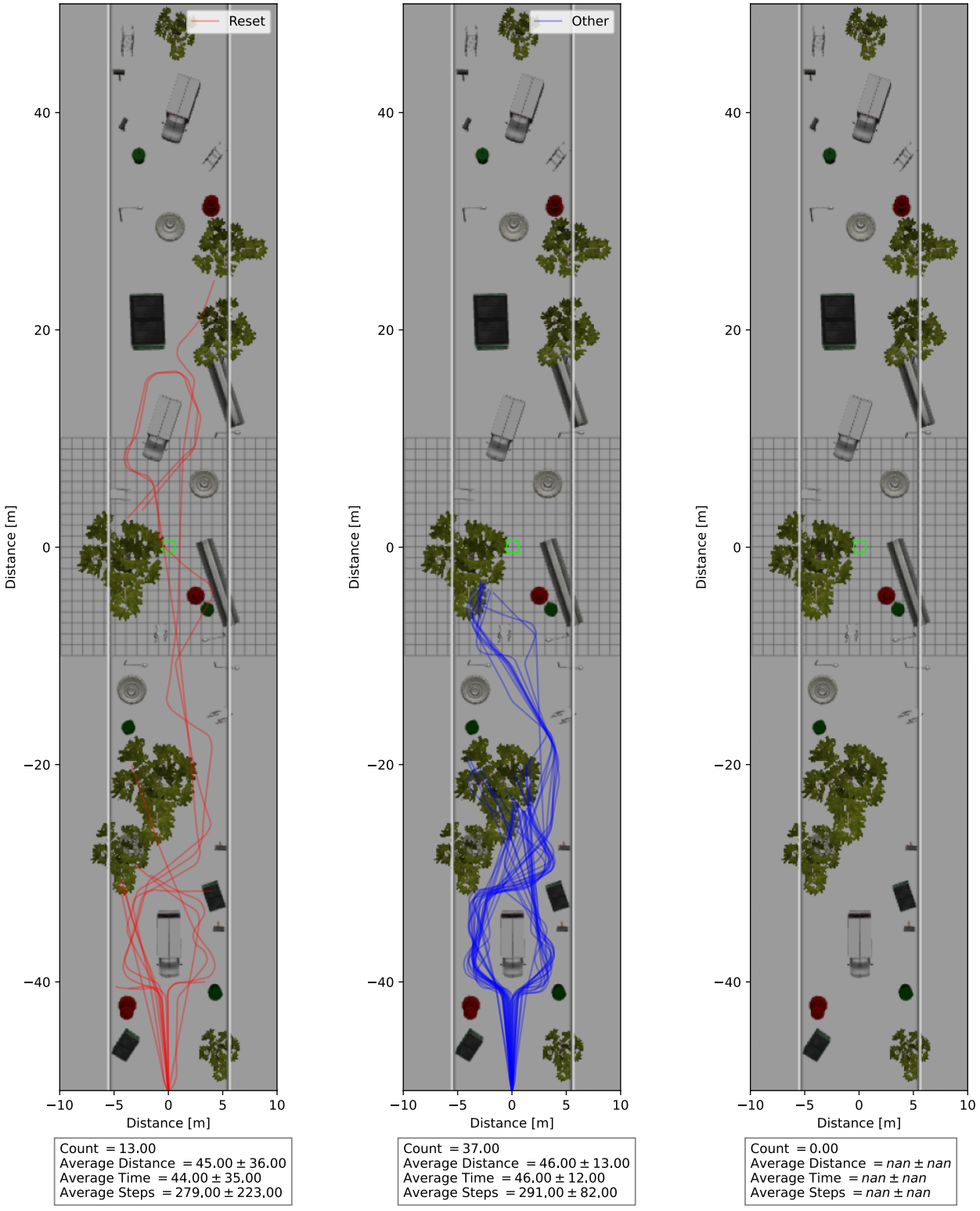
Unknown Obstacles without Filtering



(a) Reset. (b) Other. (c) Time out.

Figure 6.17: 50 episodes with unknown obstacles.

Unknown Obstacles with Filtering



(a) Reset. (b) Other. (c) Time out.

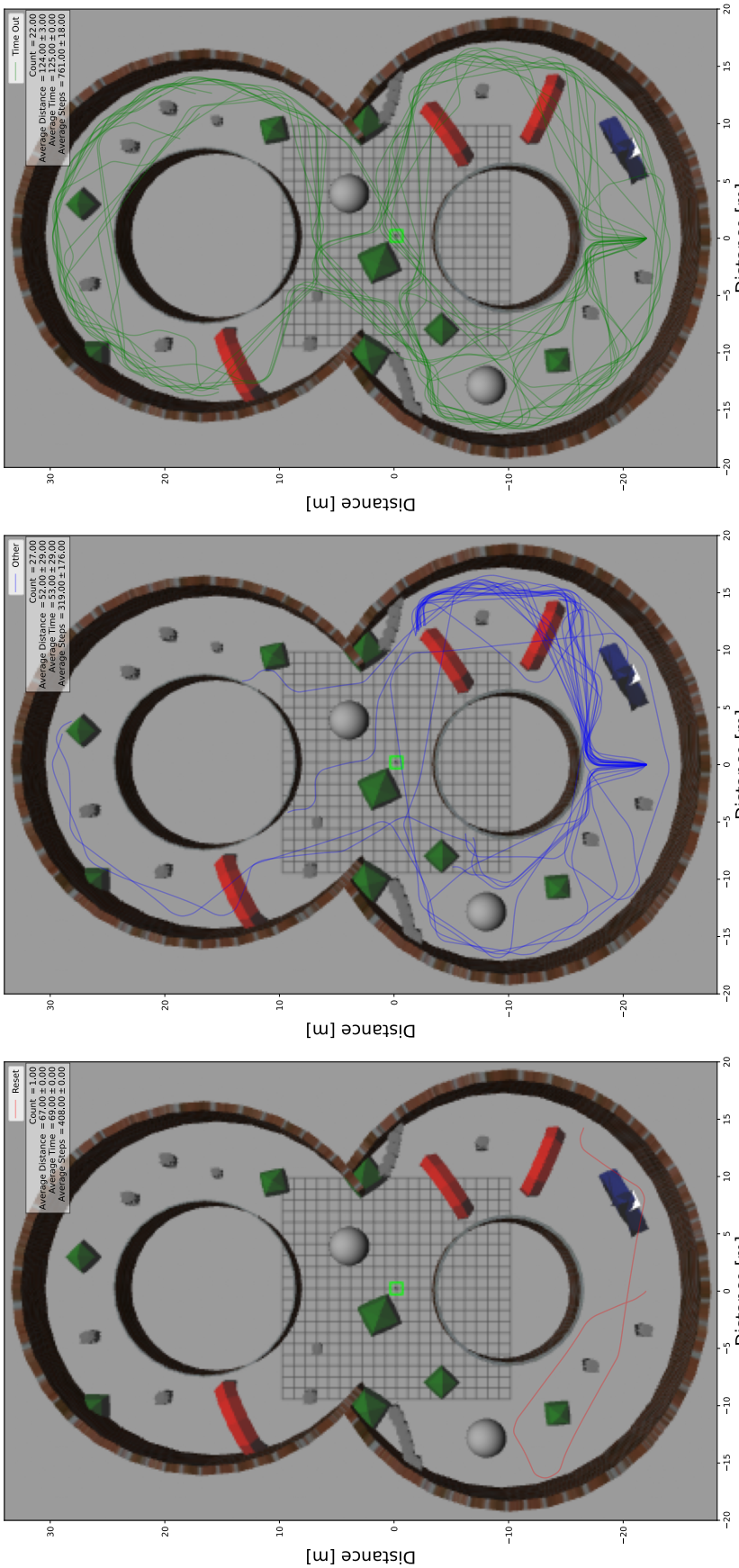
Figure 6.18: 50 episodes with unknown obstacles.

6.2.3 In an Unknown Environment with Known Obstacles

The model is tested in an "eight"-shaped environment to see how well it generalizes from the corridor environment to this new type of environment with curved walls. This type of environment, as can be seen in Appendix C.3, allows more yaw movement than the corridor environment on which the model was trained. The average time-of-flight, distance traveled, action steps executed and the collision type distribution can be seen in Table 6.3 and the results of 50 episodes of flights in fig.6.19. The drone is initialized at $(x, y, z) = (-22, 0, 2)$ every episode. The drone timed out in 44% of the cases. This can be seen in fig. 6.19c. The model generalizes well to the unfamiliar environment, and most of the collisions are blind collisions, especially caused by the U-shaped obstacle. This is a rather narrow passage that is not due to the eight-shaped environment. Guards for the corners are intentionally placed to avoid obvious reset triggers. Reset was only triggered once (2% of the episodes) and was caused by a blind collision, as can be seen in fig. 6.19a.

Table 6.3: Average time-of-flight, distance traveled and action steps executed with standard deviation (denoted std) and the collision type distribution for the eight-shaped environment with known obstacles.

Environment	Time ± std [s]	Distance ± std [m]	Steps ± std	Reset	Other	Time Out
Eight-shaped	85.11 ± 41.42	83.66 ± 41.53	515.60 ± 254.13	2%	54%	44%



(a) Reset. (b) Other. (c) Time out.

Figure 6.19: 50 episodes in an "eight"-shaped structure with known obstacles.

6.2.4 With Added Depth Image Noise

In this study, noise is added to the depth image and filtering of the noisy image is applied. The model's behavior is studied in the medium environment (from Section 6.2.1) and the simulated depth image, the noisy depth image and the filtered depth image are evaluated and compared. Depth images are, as discussed earlier in this thesis, highly affected by a characteristic noise. The measurement noise has been neglected until now as perfect sensory data obtained from the simulation have been used. Therefore, it is desired to add noise to the depth images and evaluate the model's performance with the added noise and with filtering of this noise. The characteristic noise model is inspired by the SimKinect [80] and the filtering is inspired by the image-processing algorithm created by Ku et al. [77] (which was also used during Section 6.2.2). Shortly summarized, the noise model is a combination of the work done by Barron and Malik [81] and Bohg et al. [82] and creates random offsets to shift the pixel locations in the image and bilinearly interpolate the ground truth depth values. The filtering consists of dilation, morphology and median and Gaussian blurring. The dilation has a full kernel with a size of 32×32 such that large, noisy areas are filtered out. A simulated depth image with its corresponding noisy and filtered versions is presented in fig. 6.20. The average CPU time with standard deviation for filtering the depth images during this study was 5.15ms±7.29ms (Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz).

Two important notes to make are that (i) the model used is trained on perfectly simulated depth images and has never witnessed noise before, and (ii) realistic depth image noise modeling is difficult to imitate and is a challenging field of study on its own.

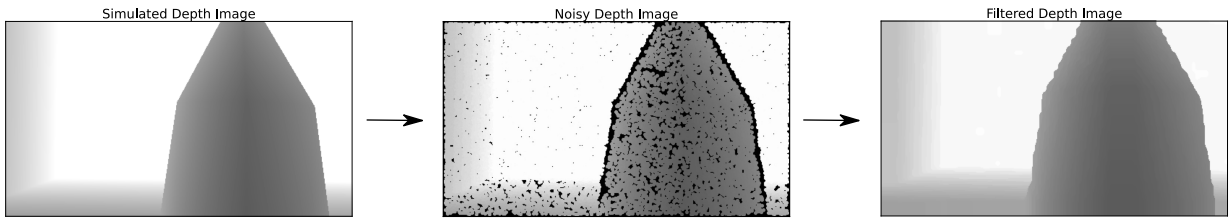


Figure 6.20: The simulated depth image, the simulated depth image with added noise and the noisy depth image with applied filtering. The filtering have a dilation kernel size of 32×32.

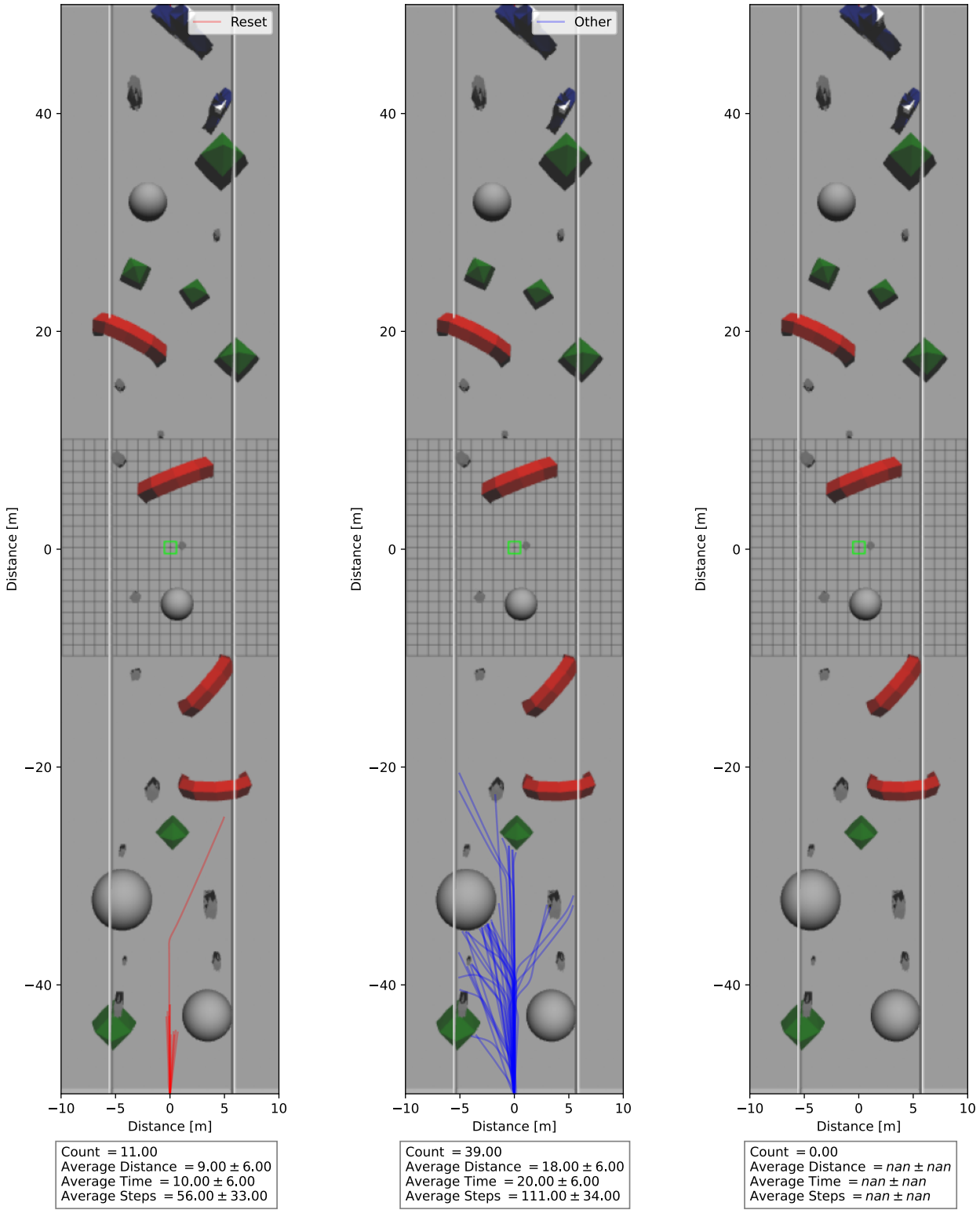
The average time-of-flight, distance traveled, action steps executed and the collision type distribution for the perfectly simulated, the noisy and the filtered depth images can be seen in Table 6.4. The results of 50 episodes for the perfectly simulated are shown in fig. 6.14 (same as in Section 6.2.1). The results of the noisy and the filtered depth images are shown in fig. 6.21 and fig. 6.22. As expected, the model performs poorly with the added noise. As can be seen in fig. 6.21a and fig. 6.21b the model rarely detects obstacles or detects objects that are not there. This results in a low average time-of-flight of 17.61s±7.13s and repetitive collisions in the earlier

obstacles. The behavior is expected because of the aggressive noise added, which the model has never seen before. When introducing filtering of the noisy depth images, the average time-of-flight, distance traveled and action steps executed are close to the metrics when perfectly simulated depth images are used. The average time-of-flight was 99.13s±30.80s for the filtered run and 106.46s±23.73s for the perfectly simulated run. Both also had similar numbers of time-out episodes, 48% and 44% respectively.

Table 6.4: Average time-of-flight, distance traveled and action steps executed with standard deviation (denoted std) and the collision type distribution with simulated depth images, added depth image noise and with applied filtering for the medium environment.

Depth image	Time ± std [s]	Distance ± std [m]	Steps ± std	Reset	Other	Time Out
Simulated	106.46 ± 23.73	102.99 ± 23.60	624.40 ± 143.07	16%	36%	48%
With noise	17.61 ± 7.13	15.92 ± 6.76	98.84 ± 40.95	22%	78%	0%
Filtered	99.13 ± 30.80	90.55 ± 24.47	561.06 ± 175.10	14%	42%	44%

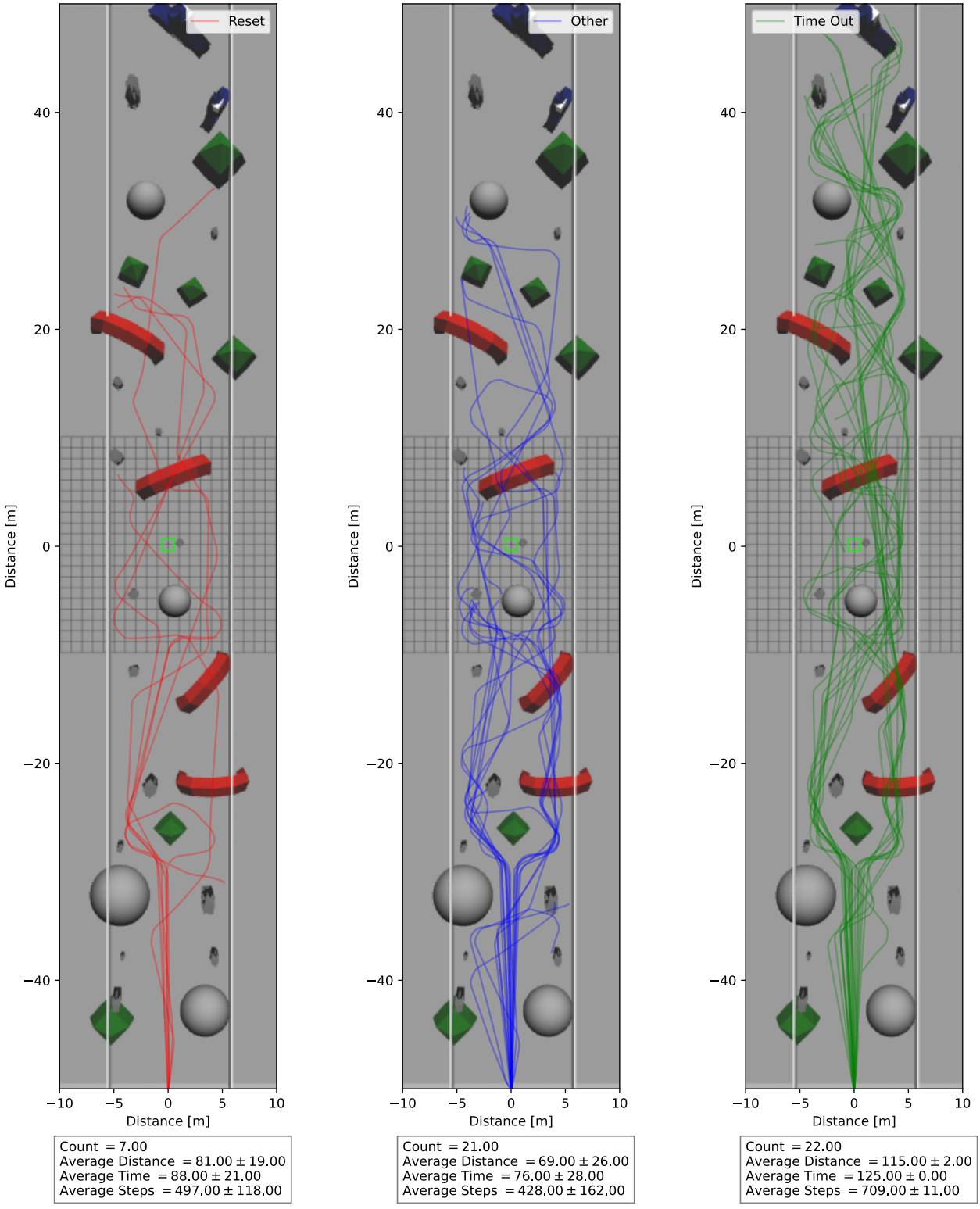
With Added Noise



(a) Reset. (b) Other. (c) Time out.

Figure 6.21: 50 episodes with noisy depth images.

With Filtering of Added Noise

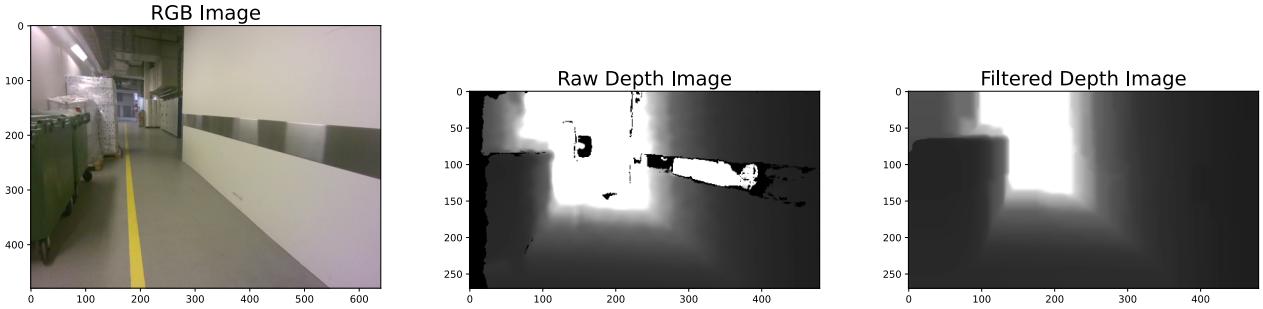


(a) Reset. (b) Other. (c) Time out.

Figure 6.22: 50 episodes with filtering of noisy depth images.

6.3 With Real-Life Depth Images

The model is also tested with real-life depth images obtained from the Intel RealSense Depth Camera D455. It is essential for the real-life flight that the model is able to make sense of real-life depth images. To visualize how the depth camera interprets the environment, an RGB image and its corresponding depth image, as well as the depth image with applied filtering, are shown in fig. 6.23. This section aims to be a proof-of-concept that the model could have successful sim-to-real applications if realized. The filtering done in this section is the same as in Section 6.2.4, but with a full 42×42-kernel used in the dilation. The reason why the kernel size is increased is due to the large reflecting areas in this specific environment, which is clear to see when comparing the reflecting metal plate in fig. 6.23a to the depth image in fig. 6.23b. A type of noise that is present in real-life environments but not accounted for in Section 6.2.4 is the noise at the end of the depth image horizon.



(a) RGB-image. (b) Depth image. (c) Filtered depth image.

Figure 6.23: RGB-image with the corresponding depth image and the filtered depth image.

The heatmaps in fig. 6.25 and fig. 6.24 show the CNN's last layer's activation of the network and are presented with a Grad-CAM visualization in the same manner as in Section 6.1.2. The heatmap with the worst trajectory, shown in fig. 6.24, shows that the combination of the action sequence and the depth image most likely is going to collide with the obstacle to the right. By comparing this heatmap with the "best possible" heatmap in fig. 6.25, it can be seen that the CNN's last layer's activation is highly dependant on the given action sequence, as the heatmaps are dissimilar and the heatmap with the worst action sequence has a collision area which corresponds to the action sequence.

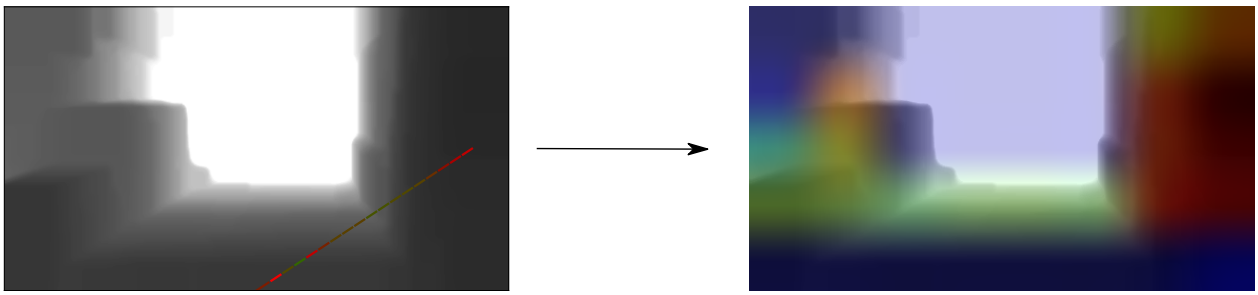


Figure 6.24: Grad-CAM visualization of the worst trajectory with a real depth image.

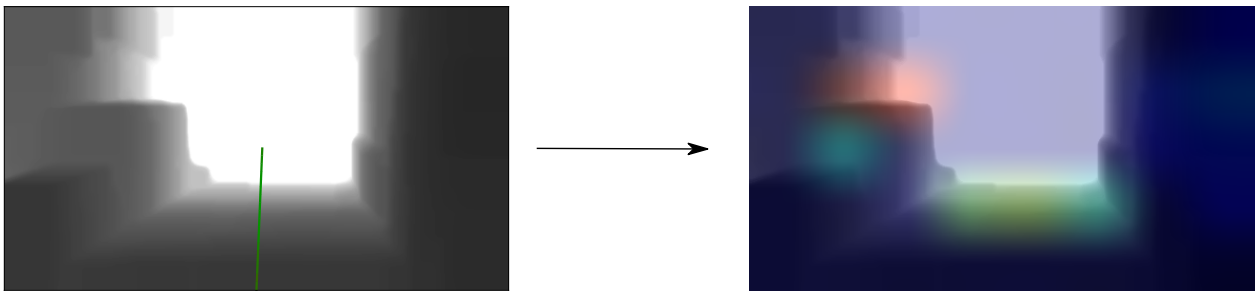
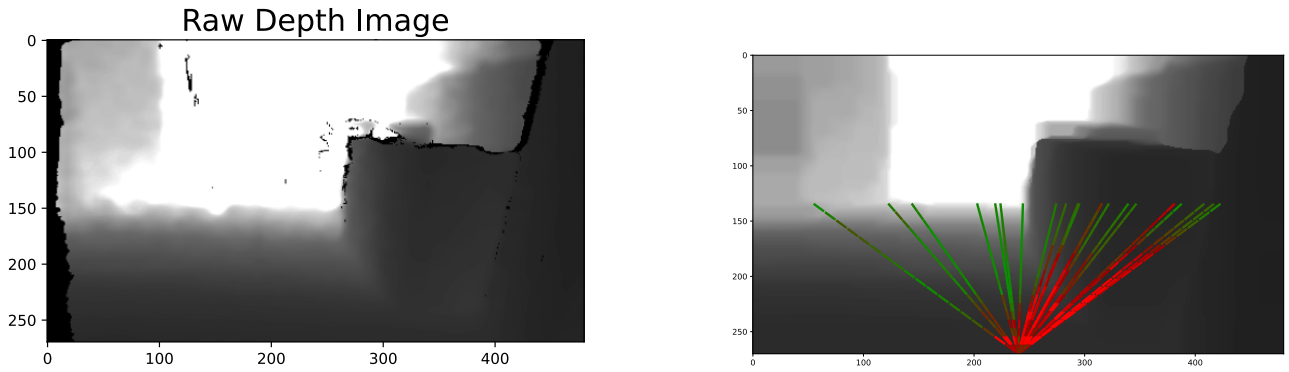


Figure 6.25: Grad-CAM visualization of the best trajectory with a real depth image.

Fig. 6.26b shows the the model's evaluation of 20 random trajectories over a filtered version of the depth image shown in fig. 6.26a. The action sequences that correspond to the trajectories that are going into the obstacle to the right in the depth image have a high collision probability, and the action sequences that correspond to the trajectories going away from the obstacles have a low collision probability. Interestingly, the first action steps have a high collision probability regardless of the trajectory's angle and velocity. This might be due to the depth camera not being aligned with the ground when the image was taken. As mentioned earlier, the model assumes a constant height. It can also be due to the depth camera being held closer to the ground than what the model is used to.



(a) (b)

Figure 6.26: Local motion planning of a real depth image. (a) shows the raw depth image and (b) shows the model's evaluation of 20 random trajectories over the filtered depth image, showing each trajectory's collision probability.

Chapter 7

Discussion

What to include as inputs to the network was tested several times during the development of the model. In earlier stages of the thesis, the robot's states (position, orientation and velocity) were concatenated with the action sequence input to form the input to the LSTM. This was done as an attempt to improve the model, as states would influence the chosen trajectory based on relevant measurements of the robot itself. However, it was found that the inclusion of the states made the model perform worse than without them. Hence, it was removed from the network. A possible reason it performed worse with the state input is due to the lack of variation for several of the states during the data generation. In the corridor environment, the robot has a velocity in the robot's x-direction of $\left(1 \pm 0.25\right)\frac{\mathrm{m}}{\mathrm{s}}$, but little variation in the y-direction and no variation in the z-direction. The same applies to the orientation, as the pitch angle is only non-zero when the robot is accelerating or decelerating, which rarely induces great pitch angle variation. The yaw angle has more variations than the other angles, but the corridor environment is not really suitable for great variations as it is rather narrow and long, forcing the robot to move in a certain direction. The roll angle also has little to no variation. In future work, the forward velocity and current yaw rate of the robot, which have large variation in reality, should be fed to the network to improve the prediction accuracy.

An approach that has been tested during this thesis regarding the action sequence generation was to calculate the UAV's motion primitives based on the approach presented by Mueller et al. [83]. With this method, the UAV's proposed motion was calculated by the drone's initial state, the desired motion duration, and any combination of components of the drone's position, velocity, and acceleration at the motion's end. Meaning that the trajectory would be waypoint-based instead of directional-based as implemented in this thesis. The UAV would follow a trajectory to a randomly selected goal destination that was within the depth sensor's FoV. The goal destination was updated after a certain amount of steps in the trajectory were executed to achieve a smooth trajectory for the UAV. The number of steps in the trajectory was determined by the goal distance and the average velocity. An issue with this approach was that the trajectory

sometimes would have steps outside the keyframe's FoV, which is not ideal. This typically happened if the UAV had a high velocity in the robot's y-direction and the computed trajectory was curved outside the FoV. Because of this issue, the method was not preferable given the depth camera's limited FoV of 86°. When this method was tested, the drone's yaw angle movement was disabled and the drone moved sideways by adjusting its roll angle. The model that was created with this motion library worked, but results were significantly better when the yaw angle was used instead of the roll angle.

There are three typical causes for collisions: (i) corner collisions, (ii) pass-by collisions and (iii) blind collisions, where each type of collision is reasonable given the drone's limitations and the trained model. The different types of collisions are illustrated in fig. 6.12. Blind collisions are the hardest to deal with as the drone does not have sufficient information about the environment but is forced to act to avoid an inevitable collision. The result of avoiding the collision is another unavoidable collision with the environment that it did not see when making the decision. A possible way of minimizing these collisions is to decrease the number of action steps executed before the trajectory replanning. The drone executes the first 8 steps of the 72-long action sequence and then replans. If several of the planned trajectories have a large yaw angle because of an oncoming obstacle, the drone will find itself with a rapid change of scenery, as the FoV is quite limited. By decreasing the number of steps executed, the drone will replan more often and get a better understanding of the current environment. This is because, at every replanning, the model also receives the current depth image (the keyframe). However, by decreasing the number of steps executed, the pass-by collisions could become more abundant. This is because when successfully navigating around an obstacle, the model would lose track of the obstacle it is passing and potentially suggest a trajectory that scratches the tip of the obstacle as it passes by. This is typical for the sphere obstacle that has been used in this thesis. Another element that could be included to tackle the blind collisions is to allow the UAV to stop its forward movement and turn around to inspect more of the environment before deciding where to go. This would allow the UAV to get out of typical blind collision scenes.

A possible way of tackling pass-by collision is to introduce a depth image sequence into the network instead of the current depth image only. The model would have a better understanding of where it currently is if this was the case. However, this would increase the input space of the network drastically, making it computationally expensive. As the height of the drone is assumed fixed, a possible solution could be only to use the most relevant parts of the depth images. Some information would be lost by doing this, but the model's understanding of its current placement in the environment would increase.

The last type of collisions, corner collisions, are more challenging to deal with as they often are caused by the depth camera's limited range. The model is not aware of whether the end of a corner-like structure would have a safe path or an inevitable collision. This was not something

that was especially taken into consideration when generating the training data. Therefore, corner collisions are probably strongly underrepresented in the training data. This can be stated because, although corners still exist in the training data, the drone does not "seek" them as it is moving randomly. In contrast, the trained model will seek the corners, as they often are found at the end of an otherwise feasible path. A 3D image of a corner collision is shown in fig. 7.1. Here, it can be seen that the trajectory deemed the best possible by the model (colored green) leads straight into a corner. The trajectory is sufficiently short, so the model does not evaluate it as a likely collision with the corner. It can also be seen from the visualization that the drone has enough space to maneuver out of the situation.

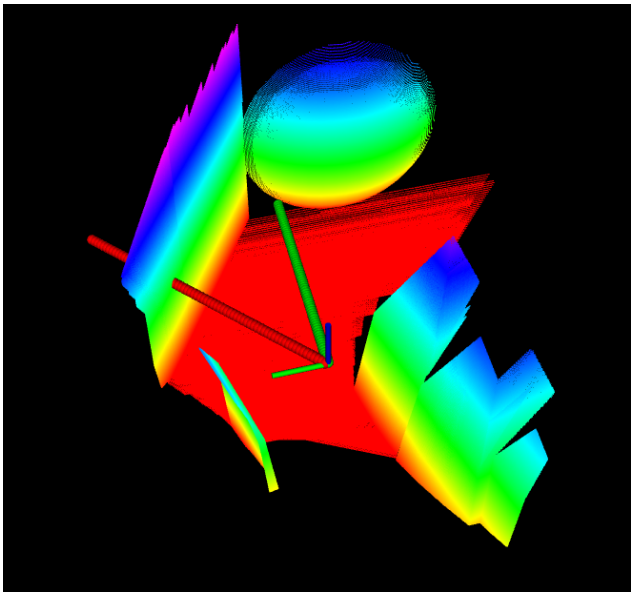


Figure 7.1: Visualization of an oncoming corner collision. The green trajectory has the lowest average collision probability and the red trajectory has the highest.

Another parameter that could be changed for the local motion planning is the velocity, which could be decreased or increased. If the velocity is decreased, it could potentially avoid more obstacles, be more stable and have fewer blind collisions as it would have more time to travel the same distance. It would also be able to adjust its course faster, as its momentum would be lower. By decreasing the velocity, it would also travel a shorter distance in the same amount of time and discover less of the environment. This trade-off must be chosen based on the application as the endurance of a UAV is limited.

As can be seen from the evaluation studies in the simulated environments presented in Section 6.2, the model will nearly every time choose a trajectory that does not alter its course except if it is necessary to avoid an obstacle. Due to this, the model will often move in straight lines. This is not enforced or prioritized but is a result of the training of the model. The model finds that the path with the lowest probability of colliding is the one that does not apply a change to

its yaw angle. This could possibly be a defense mechanism for blind collisions, as the model is conservative to changing the depth image scenery. The model does not want to build momentum for a turn as it does not want to move in unseen environments. A related observation is that the model does not really seem to care whether a trajectory is barely avoiding an obstacle (avoiding it by less than a meter distance) or by a great margin as long as the proposed trajectory does not collide. Therefore, the model is not conservatively risk-averse and does not want to pass the obstacle with a "safety distance". This is one of the main reasons for the pass-by collisions. The behavior is related to the model's desire to move in a direction that does not induce a change in its course. A possible way of handling this potential issue is to give different scores to the collisions during the data generation, where "barely passing the obstacle" would have a lower score than colliding with the obstacle but higher than avoiding it with a great margin. This would transform the problem from a binary classification problem to a multiclass classification problem. As mentioned earlier, including a depth image sequence in the network would provide the model with better situational awareness and could be a better solution for handling the pass-by collisions. A possible "quick fix" to this issue could be to introduce large dilation or another type of image processing of the depth image before feeding it into the network. By doing this, the model would believe that an obstacle is larger than it actually is and would choose a safer path around it. This is similar to what happened in Section 6.2.2 with the unknown obstacles and in Section 6.2.4 with the filtering of the added noise. The unknown obstacles had more details and finer structures that the model had never seen before. By introducing filtering of the depth image prior to the model's interpretation, the unknown objects appeared larger and with fewer details than without the filtering. This is why after the filtering was done, it was able to fly past the first tree structure, while it could not do so without the filtering done. Much of the same applies in the case with the filtering of the added noise. The objects appear larger than what they actually are, making the drone move with a "safer" distance around them. A drawback with this method is that it performs worse in tight areas where the filtering removes possible feasible paths.

Chapter 8

Conclusion and Further Work

This thesis introduces a method and a framework for motion planning using a deep collision prediction network with a depth image and an action sequence as inputs. The thesis also presents the relevant literature in motion planning and learning-based navigation, as well as the current state-of-the-art technology in these fields. A presentation of the theoretical background of this thesis is also presented in detail. A review of the experimental setup is presented, and a thorough evaluation study of the model's performance in different simulated settings and when using real-life depth images is done.

The framework presented in this thesis is divided into three components: (i) a self-labeling data generation method in a simulated environment, (ii) the creation and the training of the collision prediction network and (iii) a local planner which uses the model to navigate in cluttered environments while being exclusively motivated by intrinsic objectives. The model is tested with perfectly simulated depth images, noisy depth images and filtered depth images, as well as with real-life filtered depth images. The model is also studied in different types of environments, including different levels of difficulty, with unfamiliar obstacles and in unfamiliar environments. The results from the study are encouraging and present a deep collision prediction network as a novel method of performing motion planning for aerial robots.

In future work, the model should be tested in different real-life environments using a physical drone. There was a hope to do so at the end of the thesis, but it was postponed as the physical drone materials required to build the drone were still on the way to ship to NTNU. The depth camera (Intel RealSense Depth Camera D455) and the SOM (NVIDIA Jetson Xavier NX) were shipped in time and the development with running the model on the SOM using NVIDIA TensorRT[1] was progressing as the thesis was finalized. In this thesis, the states and the model parameters have been assumed to be perfect measurements. This is not the case in real-life systems and the model should be tested with uncertainty in the drone's states and model parameters. Online out-of-distribution detection should be implemented to study when the model is

[1] https://developer.nvidia.com/tensorrt

unable to generalize to new inputs, either due to covariate shifts or anomalous data.

Dynamic obstacles should also be introduced into the environment, and the model should be tested with different types of dynamic obstacles. However, creating a well-generalized model with dynamic data gathered from simulation could be challenging, due to the increased complexity of the system, especially for sim-to-real applications. The motion planner should evaluate the different trajectories in parallel to reduce the computational time required when running the system. This is beneficial when applying the model on a physical drone and would enable the planner to replan more often, which could be vital in dynamic environments. The parallel computation advantage of the proposed learning-based planner would be that the computational time required would not scale linearly with the number of trajectories evaluated.

Instead of the current depth image only, a depth image sequence should also be implemented as it would provide the model with greater situational awareness. For this to be realistic, in regards to the increased input space of the network, the depth images should be processed and only the most essential parts of the depth images should be fed into the network.

A motion planning algorithm with extrinsic objectives, such as waypoint navigation, should also be implemented and compared to other traditional sample-based path planning algorithms such as PRM*, RRT* or similar. This will allow a comparison between the model-based path planner with the optimal path in an environment. Another area that should be investigated is to use of the collision prediction model as a cost function for a reinforcement learning scheme. This could allow a planner to learn to navigate the environment, possibly to some waypoint or similar, based on the collision-aware model. Lastly, the model could be used as a core function for more advanced motion planning missions, such as autonomous exploration and similar.

Bibliography

[1] Sondre Holm Fyhn. Reinforcement learning for waypoint navigation of a flying robot in obstacle-free space. *Project report in TTK4550. Department of Engineering Cybernetics, NTNU - The Norwegian University of Science and Technology*, December 2020.

[2] Ben Grocholsky, James Keller, Vijay Kumar, and George Pappas. Cooperative air and ground surveillance. *Robotics Automation Magazine, IEEE*, 13:16 – 25, 10 2006. doi: 10.1109/MRA.2006.1678135.

[3] Teodor Tomić, Korbinian Schmid, Philipp Lutz, Andreas Dömel, Michael Kassecker, Elmar Mair, Iris Grixa, Felix Ruess, Michael Suppa, and Darius Burschka. Toward a fully autonomous uav: Research platform for indoor and outdoor urban search and rescue. *Robotics Automation Magazine, IEEE*, 19:46 –56, 09 2012. doi: 10.1109/MRA.2012.2206473.

[4] Matteo Fumagalli, Roberto Naldi, Alessandro Macchelli, Raffaella Carloni, Stefano Stramigioli, and Lorenzo Marconi. Modeling and control of a flying robot for contact inspection. In *IROS*, pages 3532–3537. IEEE, 2012. ISBN 978-1-4673-1737-5. URL http://dblp.uni-trier.de/db/conf/iros/iros2012.html#FumagalliNMCSM12.

[5] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006. doi: 10.1109/MRA.2006.1638022.

[6] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *IEEE Robotics Automation Magazine*, 13(3):108–117, 2006. doi: 10.1109/MRA.2006.1678144.

[7] 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)*, pages 4974–4981, 2014. doi: 10.1109/ICRA.2014.6907588.

[8] R. Reinhart, T. Dang, E. Hand, C. Papachristos, and K. Alexis. Learning-based path planning for autonomous exploration of subterranean environments. In *2020 IEEE Inter-*

national Conference on Robotics and Automation (ICRA), pages 1215–1221, 2020. doi: 10.1109/ICRA40945.2020.9196662.

[9] Dave Ferguson, Michael Darms, Chris Urmson, and Sascha Kolski. Detection, prediction, and avoidance of dynamic obstacles in urban environments. In *2008 IEEE Intelligent Vehicles Symposium*, pages 1149–1154, 2008. doi: 10.1109/IVS.2008.4621214.

[10] David Hoeller, Lorenz Wellhausen, Farbod Farshidian, and Marco Hutter. Learning a state representation and navigation in cluttered and dynamic environments. *IEEE Robotics and Automation Letters*, PP:1–1, 03 2021. doi: 10.1109/LRA.2021.3068639.

[11] Avishek Chatterjee and Venu Madhav Govindu. Noise in structured-light stereo depth cameras: Modeling and its applications. *ArXiv*, 2015.

[12] Alexander C. Woods, Hung M. Lay, and Quang P. Ha. A novel extended potential field controller for use on aerial robots. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 286–291, 2016. doi: 10.1109/COASE.2016.7743420.

[13] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):1179–1187, 1989. doi: 10.1109/21.44033.

[14] Thomas Eppenberger, Gianluca Cesari, Marcin Dymczyk, Roland Siegwart, and Renaud Dubé. Leveraging stereo-camera data for real-time dynamic obstacle detection and tracking. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10528–10535, 2020. doi: 10.1109/IROS45743.2020.9340699.

[15] Sergey Levine and Fereshteh Sadeghi. Cad2rl: Real single-image flight without a single real image. 2017. doi: 10.15607/rss.2017.xiii.034.

[16] 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. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017. doi: 10.1109/icra.2017.7989182. URL http://dx.doi.org/10.1109/ICRA.2017.7989182.

[17] P. Long, Tingxiang Fan, X. Liao, Wenxi Liu, H. Zhang, and Jia Pan. Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6252–6259, 2018.

[18] Lei Tai, Jingwei Zhang, Ming Liu, and Wolfram Burgard. Socially compliant navigation through raw depth inputs with generative adversarial imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1111–1117, 2018. doi: 10.1109/ICRA.2018.8460968.

[19] Simon Lynen, Bernhard Zeisl, Dror Aiger, Michael Bosse, Joel Hesch, Marc Pollefeys, Roland Siegwart, and Torsten Sattler. Large-scale, real-time visual–inertial localization revisited. *The International Journal of Robotics Research*, 39(9):1061–1084, 2020. doi: 10.1177/0278364920931151. URL https://doi.org/10.1177/0278364920931151.

[20] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotic Research - IJRR*, 30:846–894, 06 2011. doi: 10.1177/0278364911406761.

[21] Sikang Liu. Motion planning for micro aerial vehicles. *Publicly Accessible Penn Dissertations*, 2018. URL https://repository.upenn.edu/edissertations/3146.

[22] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996. doi: 10.1109/70.508439.

[23] S. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[24] Wennie Tabib, Kshitij Goel, John Yao, Mosam Dabhi, Curtis Boirum, and N. Michael. Real-time information-theoretic exploration with gaussian mixture model maps. In *Robotics: Science and Systems*, 2019.

[25] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Auton. Robots*, 34(3):189–206, 2013. URL http://dblp.uni-trier.de/db/journals/arobots/arobots34.html#HornungWBSB13.

[26] Helen Oleynikova, Zachary Taylor, Marius Fehr, Roland Siegwart, and Juan Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[27] Stephen Nuske, Sanjiban Choudhury, Sezal Jain, Andrew Chambers, Luke Yoder, Sebastian Scherer, Lyle Chamberlain, Hugh Cover, and Sanjiv Singh. Autonomous exploration and motion planning for an unmanned aerial vehicle navigating rivers: Autonomous exploration and motion planning for a uav navigating rivers. *Journal of Field Robotics*, 32, 06 2015. doi: 10.1002/rob.21596.

[28] 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*, pages 2820–2825, 2013. doi: 10.1109/ICRA.2013.6630967.

[29] Andreas Bircher, Mina Kamel, Kostas Alexis, Helen Oleynikova, and Roland Siegwart. Receding horizon "next-best-view" planner for 3d exploration. In *2016 IEEE International Conference on Robotics and Automation (ICRA 2016)*, pages 1462 – 1468, Piscataway, NJ, 2016. IEEE. ISBN 978-1-4673-8026-3. doi: 10.1109/ICRA.2016.7487281. IEEE International Conference on Robotics and Automation (ICRA 2017); Conference Location: Stockholm, Sweden; Conference Date: May 16-21, 2016.

[30] Tung Dang, Marco Tranzatto, Shehryar Khattak, Frank Mascarich, Kostas Alexis, and Marco Hutter. Graph-based subterranean exploration path planning using aerial and legged robots. *Journal of Field Robotics*, 37(8):1363–1388, 2020. doi: https://doi.org/10.1002/rob.21993. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21993.

[31] Mihir Dharmadhikari, Tung Dang, Lukas Solanka, Johannes Loje, Huan Nguyen, Nikhil Khedekar, and Kostas Alexis. Motion primitives-based path planning for fast and agile exploration using aerial robots. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 179–185, 2020. doi: 10.1109/ICRA40945.2020.9196964.

[32] Lukas Schmid, Michael Pantic, Raghav Khanna, Lionel Ott, Roland Siegwart, and Juan Nieto. An efficient sampling-based method for online informative path planning in unknown environments. *IEEE Robotics and Automation Letters*, 5:1–1, 01 2020. doi: 10.1109/LRA.2020.2969191.

[33] Russell Reinhart, Tung Dang, Emily Hand, Christos Papachristos, and Kostas Alexis. Learning-based path planning for autonomous exploration of subterranean environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1215–1221, 2020. doi: 10.1109/ICRA40945.2020.9196662.

[34] Markus Achtelik, Simon Lynen, Stephan Weiss, Margarita Chli, and Roland Siegwart. Motion- and uncertainty-aware path planning for micro aerial vehicles. *Journal of Field Robotics*, 31, 07 2014. doi: 10.1002/rob.21522.

[35] Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. Uncertainty-aware reinforcement learning for collision avoidance. 02 2017.

[36] Gregory Kahn, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Plato: Policy learning using adaptive trajectory optimization. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3342–3349, 2017. doi: 10.1109/ICRA.2017.7989379.

[37] Gregory Kahn, Pieter Abbeel, and Sergey Levine. Badgr: An autonomous self-supervised learning-based navigation system. *IEEE Robotics and Automation Letters*, 6(2):1312–1319, 2021. doi: 10.1109/LRA.2021.3057023.

[38] Antonio Loquercio, Ana I. Maqueda, Carlos R. del Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 2018. doi: 10.1109/LRA.2018.2795643.

[39] Péter Fankhauser and Marco Hutter. Anymal: A unique quadruped robot conquering harsh environments. *Research Features*, 126:54 – 57, 05 2018. doi: 10.3929/ethz-b-000262484.

[40] Elia Kaufmann, Antonio Loquercio, Rene Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Deep drone racing: Learning agile flight in dynamic environments. In Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto, editors, *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 133–145. PMLR, 29–31 Oct 2018. URL http://proceedings.mlr.press/v87/kaufmann18a.html.

[41] Alessandro Giusti, Jérôme Guzzi, Dan C. Cireşan, Fang-Lin He, Juan P. Rodríguez, Flavio Fontana, Matthias Faessler, Christian Forster, Jürgen Schmidhuber, Gianni Di Caro, Davide Scaramuzza, and Luca M. Gambardella. A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, 2016. doi: 10.1109/LRA.2015.2509024.

[42] Linhai Xie, Sen Wang, Andrew Markham, and Niki Trigoni. Towards monocular vision based obstacle avoidance through deep reinforcement learning. In *RSS 2017 workshop on New Frontiers for Deep Learning in Robotics*, 2017.

[43] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[44] Aj Piergiovanni, Alan Wu, and Michael S. Ryoo. Learning real-world robot policies by dreaming. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7680–7687, 2019. doi: 10.1109/IROS40897.2019.8967559.

[45] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. *Supervised Learning*, pages 21–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[46] Bing Liu. *Supervised Learning*, pages 63–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19460-3. doi: 10.1007/978-3-642-19460-3_3. URL https://doi.org/10.1007/978-3-642-19460-3_3.

[47] Froduald Kabanza, Julien Filion, Abder Benaskeur, and Hengameh Irandoust. Controlling the hypothesis space in probabilistic plan recognition. pages 2306–2312, 08 2013.

[48] Chin Ho and Panos Parpas. Empirical risk minimization: probabilistic complexity and stepsize strategy. *Computational Optimization and Applications*, 73, 06 2019. doi: 10.1007/s10589-019-00080-2.

[49] WilliamH.Jefferys. On the method of least squares. *The Astronomical Journal*, 85(2), 1980.

[50] L. Le Cam. Maximum likelihood: An introduction. *International Statistical Review / Revue Internationale de Statistique*, 58(2):153–171, 1990. ISSN 03067734, 17515823. URL http://www.jstor.org/stable/1403464.

[51] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014. ISBN 978-1-10-705713-5.

[52] Pascal Massart and Élodie Nédélec. Risk bounds for statistical learning. *The Annals of Statistics*, 34(5), Oct 2006. ISSN 0090-5364. doi: 10.1214/009053606000000786. URL http://dx.doi.org/10.1214/009053606000000786.

[53] Enzo Grossi and Massimo Buscema. Introduction to artificial neural networks. *European journal of gastroenterology hepatology*, 19:1046–54, 01 2008. doi: 10.1097/MEG.0b013e3282f198a0.

[54] Lofti A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965. URL http://www-bisc.cs.berkeley.edu/Zadeh-1965.pdf.

[55] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519. URL http://dx.doi.org/10.1037/h0042519.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. Mcclelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[57] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, volume 30, page 3, 2013.

[58] L. Trottier, P. Giguere, and B. Chaib-draa. Parametric exponential linear unit for deep convolutional neural networks. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 207–214, 2017. doi: 10.1109/ICMLA.2017.00038.

[59] R. Avenash and P. Viswanath. Semantic segmentation of satellite images using a modified cnn with hard-swish activation function. In Alain Trémeau, Giovanni Maria Farinella, and José Braz, editors, *VISIGRAPP (4: VISAPP)*, pages 413–420. SciTePress, 2019. ISBN 978-989-758-354-4. URL http://dblp.uni-trier.de/db/conf/visapp/visapp2019-1.html#AvenashV19.

[60] Russell D. Reed and Robert J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, 1999.

[61] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 9780262035613. URL https://books.google.co.in/books?id=Np9SDQAAQBAJ.

[62] P. De Boer, Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. In *Annals of Operations Research*, volume 134, pages 19–67, 2004.

[63] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, New York, USA, 20–22 Jun 2016. PMLR. URL http://proceedings.mlr.press/v48/gal16.html.

[64] David H. Hubel and Torsten N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195:215–243, 1968.

[65] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL http://dblp.uni-trier.de/db/journals/corr/corr1308.html#Graves13.

[66] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

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

[68] Russell Smith. Open dynamics engine, 2008. URL http://www.ode.org/.

[69] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. ISBN 0321552628.

[70] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, chapter RotorS—A Modular Gazebo MAV Simulator Framework, pages 595–625. Springer International Publishing, Cham, 2016. 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.

[71] Stanford Artificial Intelligence Laboratory et al. Robotic operating system (ros), May .

[72] Martín 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, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[73] Francois Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras.

[74] Paolo De Petris, Huan Nguyen, Tung Dang, Frank Mascarich, and Kostas Alexis. Collision-tolerant autonomous navigation through manhole-sized confined environments. In *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 84–89, 2020. doi: 10.1109/SSRR50563.2020.9292583.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[76] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[77] Jason Ku, Ali Harakeh, and Steven L. Waslander. In defense of classical image processing: Fast depth completion on the cpu. In *2018 15th Conference on Computer and Robot Vision (CRV)*, pages 16–22, 2018. doi: 10.1109/CRV.2018.00013.

[78] HyeongRyeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. Rviz: a toolkit for real domain data visualization. *Telecommunication Systems*, 60:1–9, 10 2015. doi: 10.1007/s11235-015-0034-5.

[79] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, 2017. doi: 10.1109/ICCV.2017.74.

[80] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. *ICRA*, 2014.

[81] Jonathan T. Barron and Jitendra Malik. Intrinsic scene properties from a single rgb-d image. *CVPR*, 2013.

[82] Jeannette Bohg, Javier Romero, Alexander Herzog, and Stefan Schaal. Robot arm pose estimation through pixel-wise part classification. *ICRA*, 2014.

[83] M. W. Mueller, M. Hehn, and R. D'Andrea. A computationally efficient motion primitive for quadrocopter trajectory generation. *IEEE Transactions on Robotics*, 31:1294–1310, 2015.

Appendix A

Acronyms

ANN Artificial neural network

CNN Convolutional neural network

DNN Deep neural network

ERM Empirical risk minimization

FoV Field of view

GNSS Global navigation Satellite Systems

MAV Micro aerial vehicle

ML Machine learning

MLE Maximum likelihood estimation

MPC Model predictive control

PRM Probabilistic roadmaps

LSTM Long short-term memory

RRT Rapidly-exploring random trees

ROS Robot operating system

RNN Recurrent neural network

RMF Resilient micro flyer

SLAM Simultaneous localization and mapping

SRM Structural risk minimization

UAV Unmanned aerial vehicle

Appendix B

ROS-graph

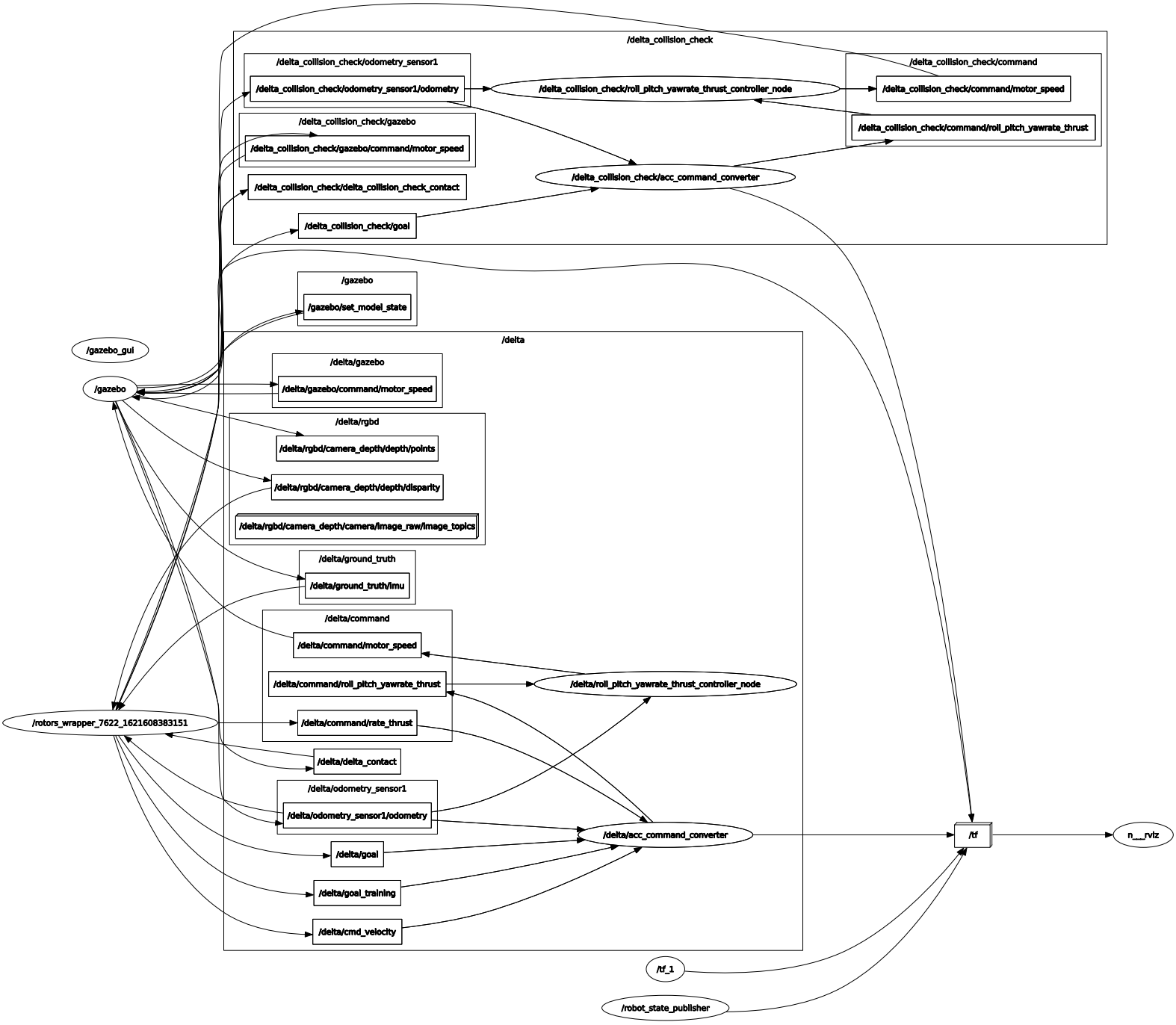


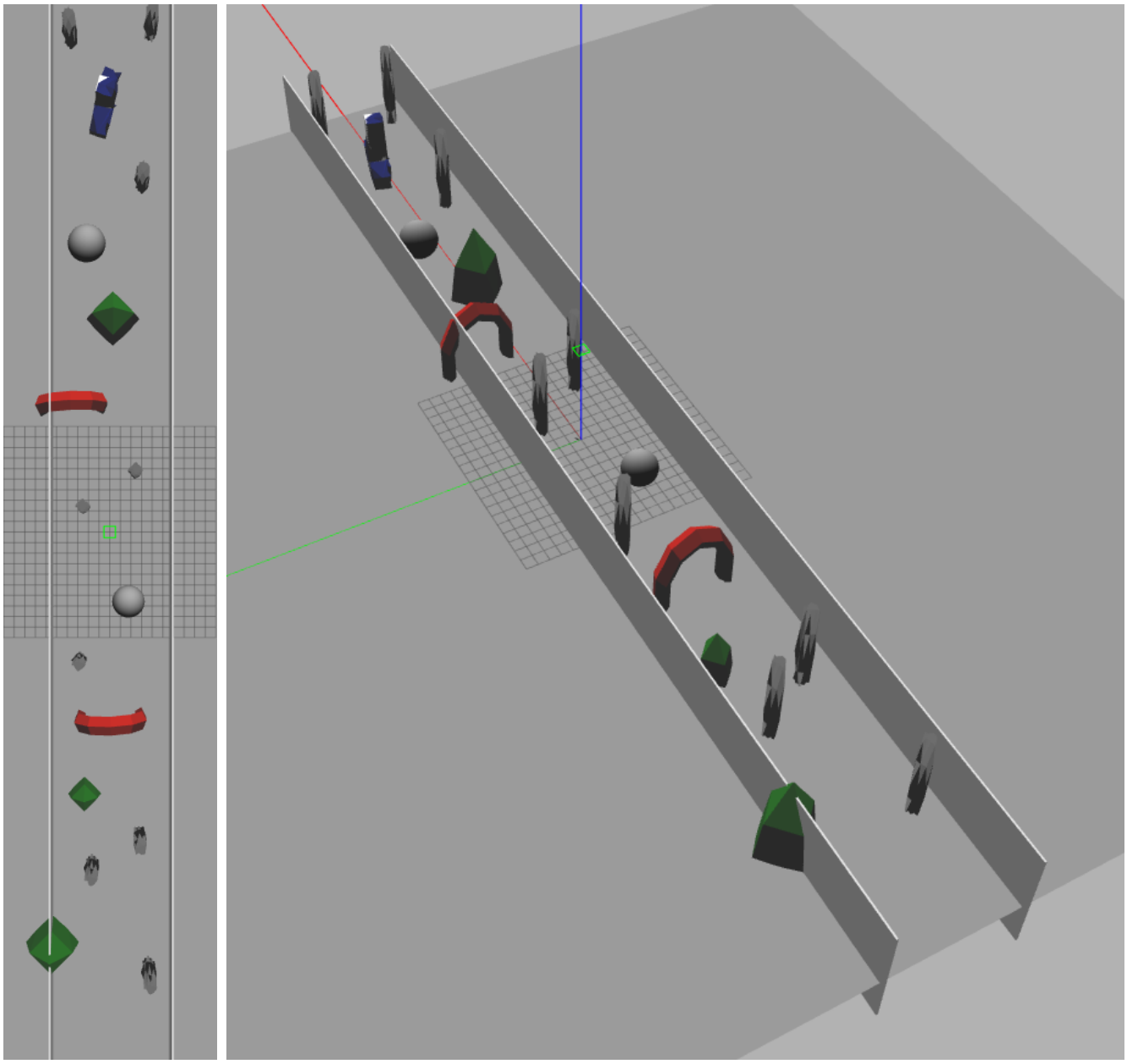
Figure B.1: A complete graph over the ROS-topics and the connections between them in the simulation environment. The RMF-drone is called "delta".

Appendix C

Environments

C.1 The Environments with Increasing Difficulty

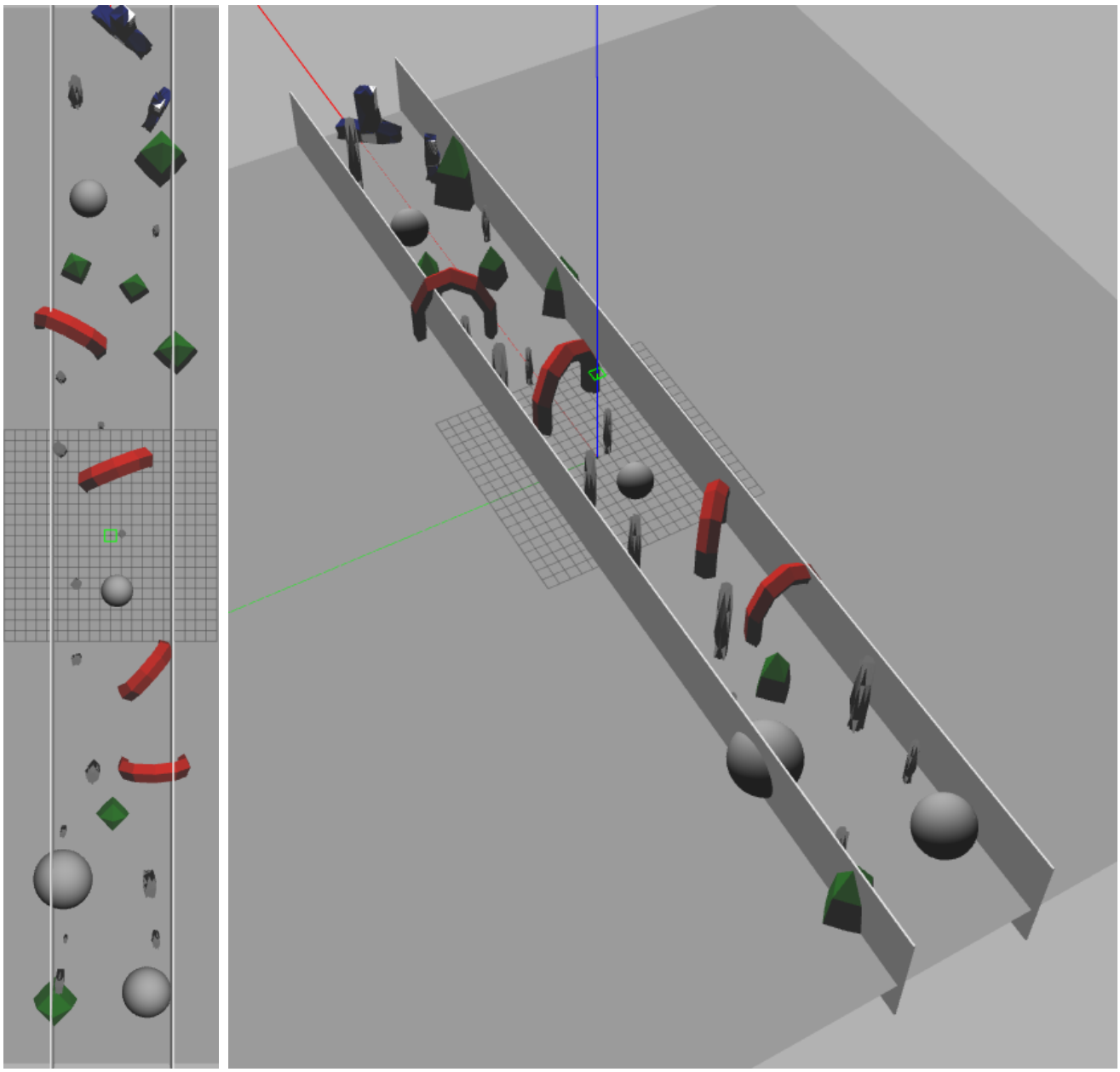
C.1.1 Easy



(a) (b)

Figure C.1: The environment with easy difficulty.

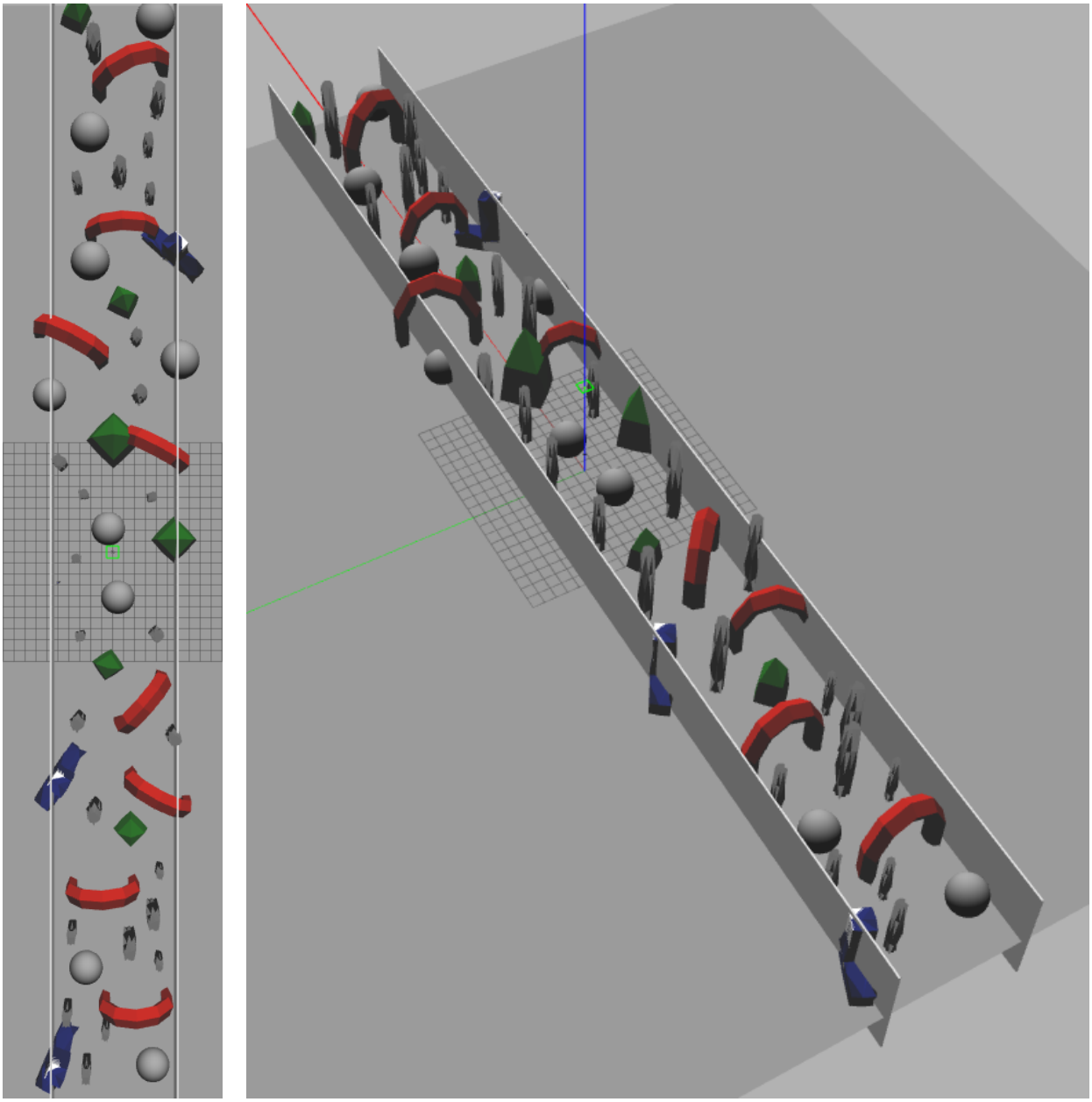
C.1.2 Medium



(a) (b)

Figure C.2: The environment with medium difficulty.

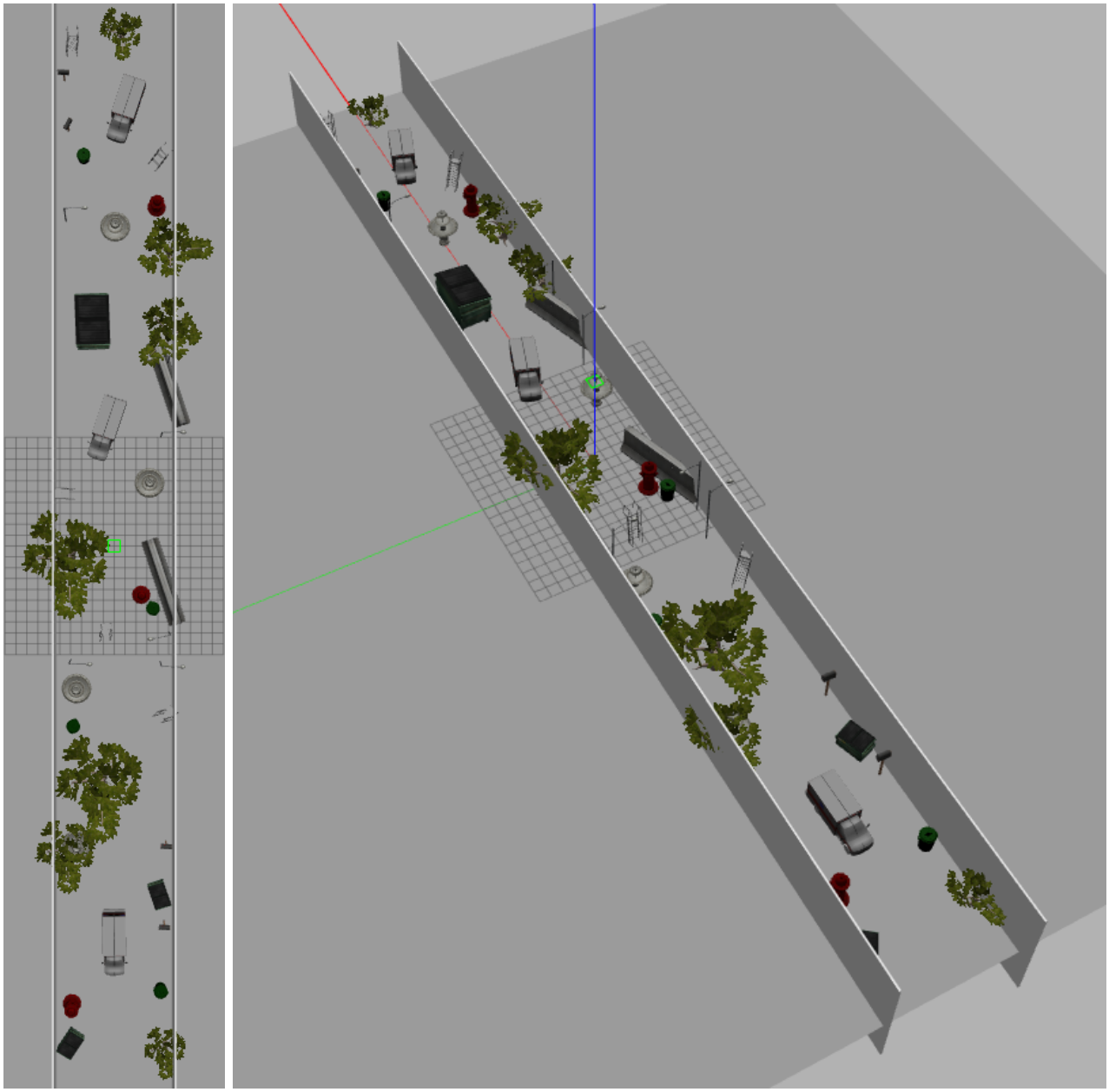
C.1.3 Hard



(a) (b)

Figure C.3: The environment with hard difficulty.

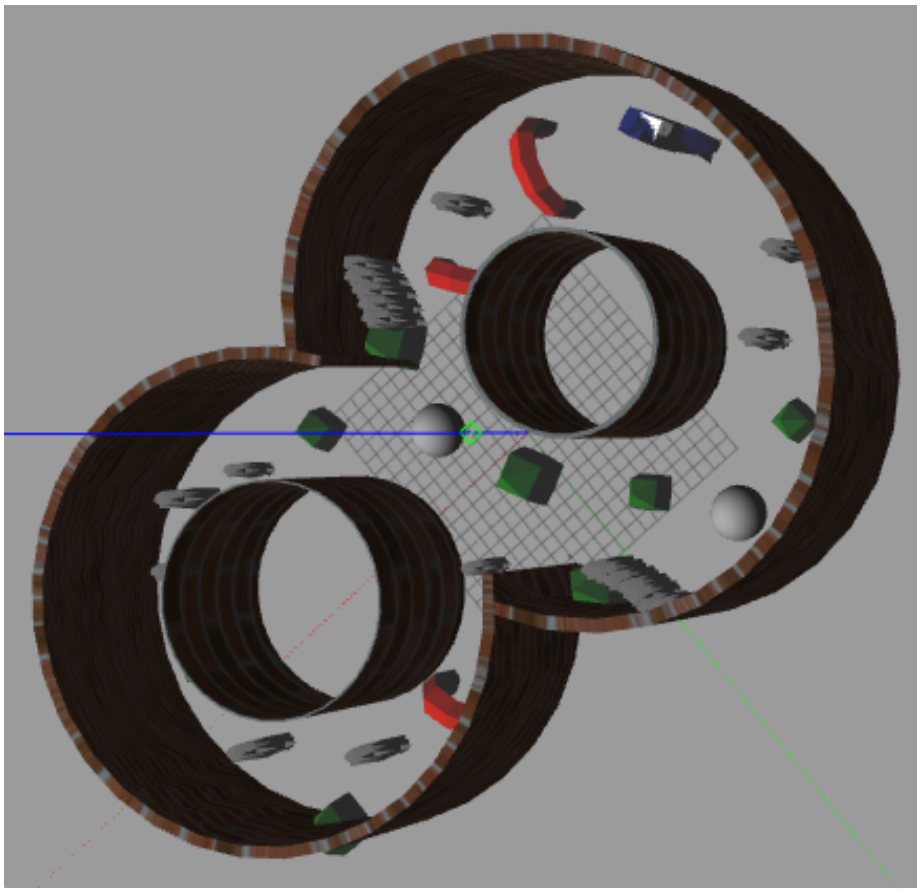
C.2 The Environment With Unknown Obstacles



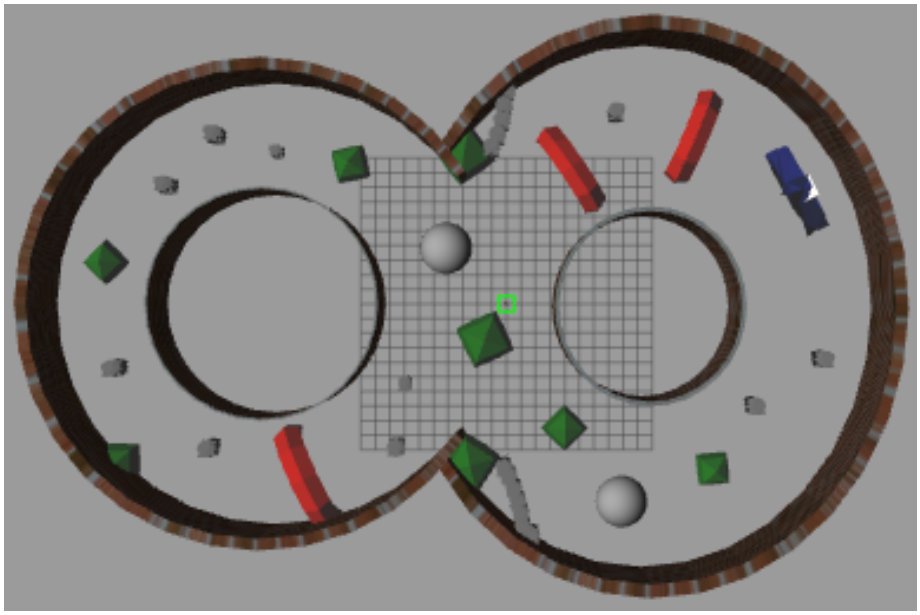
(a) (b)

Figure C.4: The environment with unknown obstacles.

C.3 The Eight-shaped Environment with Known Obstacles



(b)



(a)

Figure C.5: The "eight"-shaped environment.

Sondre Holm Fyhn

Learning-Based Navigation in Cluttered Environments Using A Deep Collision Prediction Network

NTNU
Norwegian University of
Science and Technology