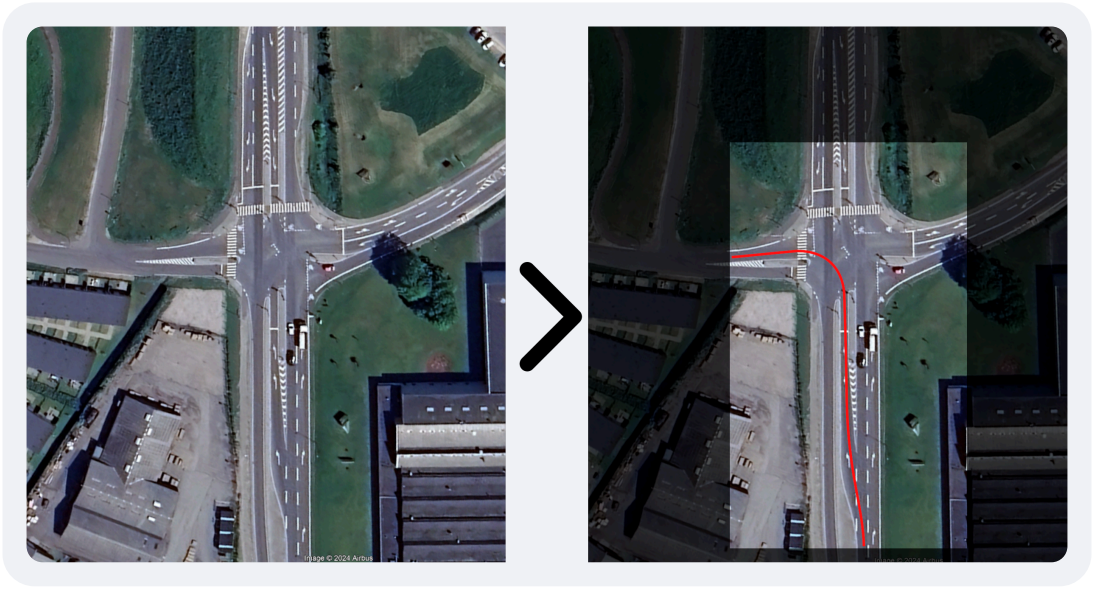
Pre-planning Intersection Traversal for Autonomous Vehicles

Master's Thesis in Computer Engineering

Ian Dahl Oliver

Department of Electrical and Computer Engineering
Aarhus University
Aarhus, Denmark
ian.oliver@post.au.dk

08-04-2025



Supervisor: Lukas Esterle
lukas.esterle@ece.au.dk

AARHUS UNIVERSITY

Preface

This master thesis is titled *"Pre-planning Intersection Traversal for Autonomous Vehicles"* and is devised by Ian Dahl Oliver. The author is a student at Aarhus University, Department of Electrical and Computer Engineering, enrolled in the Computer Engineering Master's programme. The author has completed a Bachelor's degree in Computer Engineering under the same conditions.

The thesis has been conducted in the period from 27-01-2025 to 05-06-2025, and supervised by Associate Professor Lukas Esterle. I would like to express my gratitudes to my supervisor for his support and advice throughout the project.

An additional thanks goes to Associate Professor at AU, Kaare Mikkelsen for his guidance in the early stages of this project.

Enjoy reading,
Ian Dahl Oliver

Abstract

hello Robot Operating System 2 (ROS2)

Nomenclature

Some terminology and type setting used in this thesis may not be familiar to the reader, and are explained here for clarity.

`monospace`	Inline monospace text is used for code function names, variables, or parameters.
`a.b`	In inline monospace text, a period `.` is used to denote a method or property of an object. Can also be used outside of monospace text.
`listing:<int>`	A reference to a specific listing, where `<int>` represents a line number.
`listing:<int>-<int>`	Reference to a range of lines within a listing.
`file.ext`	A reference to a specific file of given file type.
`file.ext:<func>`	A reference to a specific function within a file.
Stack	Also called a suite, a stack is a collection of related functions.

Acronyms Index

Acronym	Definition
AI	Artificial Intelligence
AIM	Autonomous Intersection Management
ANN	Artificial Neural Network
APF	Artificial Potential Field
API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
AV	Autonomous Vehicle
BCE	Binary Cross-Entropy
BEV	Bird's Eye View
CE	Cross-Entropy
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DL	Deep Learning
DOF	Degrees of Freedom
EV	Electric Vehicle
FCN	Fully Convolutional Network
FL	Fuzzy Logic
GA	Genetic Algorithm
GAN	Generative Adversarial Network
GIMP	GNU Image Manipulation Program
GNN	Graph Neural Network
GUI	Graphical User Interface
ID	Identifier
JSON	JavaScript Object Notation
MLP	Multilayer Perceptron
NN	Neural Network
PNG	Portable Network Graphics

Acronym	Definition
PRNG	Pseudo-Random Number Generator
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNG	Random Number Generator
ROS2	Robot Operating System 2
RRT	Rapidly-exploring Random Tree
TOML	Tom's Obvious Minimal Language
UI	User Interface
UX	User Experience
V2X	Vehicle-to-Everything
ViT	Vision Transformer
YAML	YAML Ain't Markup Language

Contents

Preface ii

Abstract iii

Nomenclature iv

Acronyms Index v

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problem Statement 2
- 1.3 Research Questions 3

2 Background 4
- 2.1 Autonomous Vehicles 4
- 2.2 Deep Learning 10
- 2.3 Satellite Imagery 26

3 Related Work 28
- 3.1 Path-planning 28
- 3.2 Intersection Management 29

4 Methodology 30
- 4.1 Satellite Imagery 30
- 4.2 Loss Function Design 33
- 4.3 Dataset Creation 48
- 4.4 The Models 64

5 Results 66

6 Discussion 67
- 6.1 Integration with existing systems 67
- 6.2 Shortcomings 67
- 6.3 Future Work 67
- 6.4 Other considerations 67

7 Conclusion 68

References 69

Appendix vi

1 Introduction

The introduction to the thesis will be structured as follows. First, Section 1.1 presents the motivation for this thesis with the problem statement following in Section 1.2. With the motivation and problem statement in place, the research questions will be presented in Section 1.3.

1.1 Motivation

Since the dawn of time, humans have made strides in automating any and all systems that surround them. In the days before technology, humans used animals to help them with their daily tasks. As technology advanced and exploded during the Industrial Revolution, humans replaced animals with machinery with the intent of automating production tasks. As this mentality continued to grow and technologies improved at a rapid pace, automation spread to other areas of life, such as transportation. The first Autonomous Vehicles (AVs) were developed in the 1980s by both Americans at Carnegie Mellon University [1], [2] and Europeans at Mercedes-Benz and Bundeswehr University Munich [3]. Since then, the development has spread to the individual car manufacturers instead of universities, making it more of a competitive field than a cooperative one. Still, the development in recent years with the rise of more powerful computers and the invention of machine learning, has led the field to become a fiercely researched area with many companies doing their part to create reliable, efficient, and safe AVs.

Despite this rapid development, AVs still encounter many challenges in their deployment, chief amongst which is their ability to handle intersections [4]. Unlike motorway driving — where lane following and obstacle detection are relatively well-defined tasks with few obstacles — intersection introduce a lot of complexity. Challenges arise from many different factors, such as unpredictable driver behaviour of other drivers, a huge variety of intersection types and configurations, and the state of the intersection with regards to faded or obstructed lane markings. Current solutions rely heavily on on-board sensors for perception and reactive decision-making, which can struggle in certain situations. Other solutions to intersections rely heavily on infrastructure support, such as V2X communication, which is not yet widely deployed.

A potential alternative to purely perception-based or infrastructure-dependent approaches is pre-planned path traversal, where an AV generates an optimal path through an intersection before reaching it. By leveraging Deep Learning (DL) models trained on annotated satellite imagery, AVs can gain a significantly increased amount of understanding

about the intersection it is about to enter. This presents many potential improvements to the capability and efficient of AVs. Firstly, it can reduce the reliance on on-board sensors by taking away the need for real-time perception and decision-making. Secondly, it can reduce the reliance on infrastructure support by allowing the AV to make decisions based on the pre-planned path. Finally, it can increase the safety of the AV by reducing the number of unpredictable situations it may encounter and it give it a fair chance when it then does encounter foreign situations.

Beyond improving the performance of individual AVs, better intersection handling has even broader implications to the public as a whole. Optimized intersection traversal can lead to smoother traffic flow, reduced congestion, and improved urban mobility. By generating more efficient paths, AVs can reduce waiting times, minimize unnecessary stops, and create a more predictable and coordinated traffic environment. As the proportion of autonomous vehicles on the road increases, these optimizations scale exponentially, leading to fewer bottlenecks and a decrease in fuel consumption and emissions.

1.2 Problem Statement

Advancements in AV technologies have been at the forefront of tech innovations in the 21st century. A key challenge in the development of fully autonomous vehicles, is their ability to handle intersections. Intersections pose a wide variety of challenges to AVs: from those posed by complex structures, to those posed by the unpredictability of human drivers, to faded lines that make it difficult for on-board computer vision system to clearly identify lanes or paths. All of these hinder AVs from reaching their full potential and being able to navigate intersections safely and efficiently.

Current existing solutions are very infrastructure-dependent. The Car2X system by Volkswagen, for example, relies on a network of sensors and communication devices installed in the infrastructure to spread information to vehicles on the road [5]. Autonomous Intersection Management (AIM) also relies on infrastructure to provide vehicles with information regarding intersections, with an orchestrator monitoring and managing individual intersections [6], [7], [8], with active development moving towards a more decentralized and distributed approach [9]. Furthermore, reliance on camera-based vision is susceptible to environmental limitations, such as adverse weather, that reduce system reliability.

The challenges posed by intersections cause major problems for AV developers who want to push fully autonomous driving. AVs' inability to properly react to and handle intersections, leads to significant delays in real-world deployment as a consequence of the unreliability experienced by regulators and the general public. If AVs want to enter the market with full self-driving capabilities, full autonomy is a key challenge to be tackled, as it is an essential task experienced when driving.

This projects aims to develop a solution that will help AVs to better handle intersections. With the use of DL and Computer Vision (CV) technologies, trained on and utilizing satellite imagery, this project aims to train a model that can accurately identify the proper path for an AV to travel through an intersection. The system is not meant to replace current

systems deployed in AVs, but rather assist the existing systems make better decisions when in self-driving mode, approaching an arbitrary intersection.

1.3 Research Questions

The following research questions have been formulated to address key challenges in AV path planning at intersections. The questions are designed to explore the effectiveness of different approaches and models in generating accurate and efficient paths for autonomous vehicles. The research questions are as follows:

RQ-1 How can pixel-subset–based deep learning approaches, including Convolutional Neural Networks (CNNs), Fully Convolutional Networks (FCNs), Vision Transformers (ViTs), and diffusion-based models, be optimized to improve accuracy and efficiency in path planning for autonomous vehicles at intersections?

RQ-2 Is it possible to design a loss function that effectively captures the similarity between generated and desired paths for autonomous vehicles without forcing exact matches?

RQ-3 Is it possible to create a dataset that allows for the training of a model, such that the data is not too stringent to a singular path?

2 Background

This section outlines the theory relevant to the thesis. It begins with an introduction to AVs to provide a thorough understanding of the context in which the work is situated. The tool with which this project's problem statement will be tackled is presented in Section 2.2, where the fundamentals of DL are presented. Within this section, the focus is on CV and its applications in AVs, as well as the importance of datasets in training DL models. Section 2.3 presents underlying theory on satellite imagery and its applications in AVs. Each section and subsection includes clear examples to aid comprehension.

2.1 Autonomous Vehicles

Initially released in 2014 by the Society of Automotive Engineers (SAE), the J3016 standard [10] defines six levels of driving automation, ranging from Level 0 to Level 5, with the latest revision released in 2021 [11]. These levels are visualized in Figure 1. These levels are further split into two separate categories based on the environment observer; the first three levels are concerned with the human driver being the environment observer, and the latter three levels are concerned with the vehicle being the environment observer named automatic driving system (ADS) features.

To understand how SAE defines the levels, it is important to gain a high-level overview of how these levels are defined. The document starts by defining the scope of the standard, clearly stating that it "describes [motor] vehicle driving automation systems that perform part or all of the dynamic driving task (DDT) on a sustained basis" [11, p. 4]. This definition excludes any momentary actor systems in place in a car, such as electronic stability control, automatic emergency braking, or lane keeping assistance (LKA). These systems are not considered to be part of the DDT, as they do not perform the driving task on a sustained basis. Finally, three primary actors are defined: the human user, the driving automation system, and other vehicle systems and components.

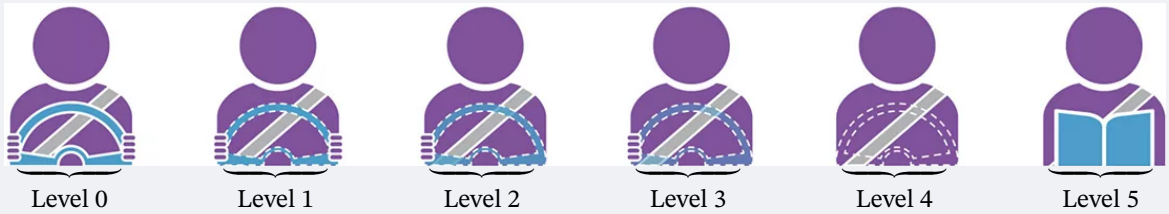


Figure 1. SAE 6 levels of autonomy. Image source: Synopsys [12]

Of course, many different manufacturers are actively developing their own systems, slowly climbing through the levels of automation, with some being further along than others. A comprehensive overview of the levels will be presented the following along with clear examples of each level. The levels are defined as follows:

- **Level 0**: No Automation. *Manual control. The human performs all driving tasks (steering, acceleration, braking, etc)*[1].

Level 0 autonomy may encompass more cars than one would imagine. This is largely due to the fact, that, as stated, momentary systems are not included as giving a vehicle any form of autonomy. Shown with the visualization in Figure 1, this level of autonomy required the user to be in full control at all times during the DDT. That means keeping both hands on the wheel at all times and staying aware of the environment for the duration of the trip. Common systems like emergency braking and lane keeping assistance do not push vehicles with these features any higher than this level. This is due to their unsustained nature.

Today, while exact numbers are unknown, it is believed that the majority of vehicles on the road today are at this level. Statistics by statista [13], [14], hints that very few vehicles sold before 2014 had any form of autonomy. This is shown by their 2015 statistics, showcasing that 51.3% of vehicles sold that year were Level 0. By 2018 this number had dropped to 24% and by 2023 it was down to just 7%, with a massive shift towards Level 1 coming in at 71%.

- **Level 1**: Driver Assistance. *The vehicle features a single automated system (e.g. it monitors speed through cruise control).*

The first commercially available adaptive cruise control (ACC) vehicles came from Chrysler in 1958 following the invention of the SpeedoStat [15]. They named this system the "Auto-Pilot" and vehicles equipped with it, was some of the first vehicles in history to achieve what would later be called Level 1 autonomy. Soon after, Cadillac adopted the technology for their own vehicles and dubbed it "Cruise Control", which since became the default term for the ACC technology, which is still used today, though usually with the term "adaptive" in front of it.

As shown in the visualization in Figure 1, the steering wheel has a dashed out look, and if the pedal were shown, they would be dashed out as well. This is to show that the human driver is not in control of every aspect of the DDT. Per the definition, Level 1 autonomy is defined as an autonomous system working on either the lateral or the longitudinal vehicle motion, which means that it can either control the steering or the acceleration of the vehicle. The human driver is still in control of the other aspect of the DDT. ACC is one of the most common systems elevating vehicles into this level. ACC can in this level not work with other systems, such as automatic steering. For vehicles that do both, end up in Level 2.

- **Level 2**: Partial Automation. *Advanced driver-assistance system (ADAS). The vehicle can perform steering and acceleration. The human still monitors all tasks and can take control at any time.*

Level 2 autonomy is the first level where the vehicle can perform both steering and acceleration. This is done through a combination of systems, such as adaptive cruise control

[1]These italicised definitions are from the infographic shown on the Synopsys blog covering the topic [12].

(ACC) and lane-keeping assistance (LKA). The human driver is still in control of the vehicle and must monitor the environment at all times. This level is the most common level with Electric Vehicles (EVs), as all have ACC and LKA systems to a smaller or greater extent and capability. As visualized, the driver must always have at least one hand on the wheel at all times and be aware of the surrounding environment. This is the level where most of the current systems are at, such as Tesla Autopilot, Ford BlueCruise, and GM Super Cruise. Despite only being Level 2, Ford's BlueCruise has been allowed to establish "blue zones" in the EU, where drivers are allowed to remove the hands from the steering wheel when engaged [16].

At this level, the European version of the Tesla Autopilot belongs, where it's American relative is climbing for level 4. This is despite the fact that Tesla Full Self-Driving (FSD) is legally classified as Level 2 in both regions, but in the US they operate at a higher level due to more permissive testing environments and less restrictive oversight [17]. Thus, laws and regulations create a functional separation at this level, as this is the level currently allowed in EU countries, with the single exception, as of 2022, being the Mercedes S-Class, allowed to operate in Level 3 under strict conditions [18]. This allowance has since been given to other manufacturers.

- **Level 3**: Conditional Automation. *Environmental detection capabilities. The vehicle can perform most driving tasks, but human override is still required.*

Level 3 autonomy is characterised by their ability to detect and act upon the environment by themselves. This means that the vehicle can perform most DDTs, but the human driver must still be able to take control at any time. This level is the first level where the vehicle can operate without human intervention in certain situations, such as highway driving in cases where overtaking a slow moving vehicle is possible. This is also the first level to rely on automated systems to monitor the environment.

An important note as the level keeps rising, is what the SAE calls DDT Fallback. This refers to the action of taking over the DDT as a user, in the event of a DDT performance-relevant system failure or upon operational design domain (ODD) exit. ODD essentially means that the vehicle can only operate in certain environments, small and defined, like specific motorways, or broad, like an entire trip. For instance, the ODD for Ford's BlueCruise is when driving in the predefined "blue zones", meaning that the user must take over when leaving them. While Level 3 autonomy has limited spread, some manufacturers are allowed to operate at this level, with Germany laying out the groundwork for the road to Level 4 autonomy [18].

- **Level 4**: High Automation. *The vehicle performs all driving tasks under specific circumstances. Geofencing is required. Human override is still an option.*

The main difference between Level 3 and Level 4 is the fact that the vehicle can operate without human intervention in certain situations. This means that the vehicle can perform all driving tasks under specific circumstances, such as highway driving or in urban environments. This is the referred to as geofencing, where the vehicle can only operate in certain areas. As visualized in Figure 1, the steering wheel is now only outlined by dashes, meaning that the human driver is not in control of the vehicle at all times, but can take over if needed or required by the vehicle.

There are currently no commercially available vehicles that reach this level of autonomy, but some manufacturers have developed and deployed what is essentially Level 4 systems. For instance, Waymo and Cruise have developed systems that can operate in certain areas, such as San Francisco and Phoenix. These systems are available to customer use, but they are developing vehicle any one can buy. Both Waymo and Cruise act as a taxi service, where users can request a ride through an app. As mentioned, however, these systems are only allowed to operate in specified, geofenced areas, and they are not allowed to operate outside of these areas. This is what is keeping these technologies at Level 4 instead of Level 5. They gather massive amounts of detailed data on the cities they operate in and train their systems on this data, meaning they are great in their respective areas, but they are not able to operate outside of them. Furthermore, Tesla unveiled their own robotaxis at the "We, Robot" event, featuring vehicles without steering wheels or pedals. While not commercially available either, they still present a glimpse of the future of Level 4 autonomy, potentially even Level 5.

- **Level 5**: Full Automation. *The vehicle performs all driving tasks under all conditions. Zero human attention or interaction is required.*

Level 5 is the final level of the taxonomy presented by SAE. At this level, the ODD is unlimited, whereas every earlier level has been limited. At this level, both the DDT and DDT Fallback are fully handled by the vehicle. This means that the human user relinquishes the role of driver, and becomes a passenger, purely. The vehicle can operate in any environment, and the human user does not need to be aware of the environment at all. Not only does this mean that the vehicle can operate in any environment, but it also means that the vehicle can operate in conditions. If at any point during a trip, the system can't figure out how to navigate and for any reason requires human intervention, then it is not Level 5. This is neatly presented in the visualization in Figure 1, where there is no steering wheel in the image and the once-drive-now-passenger is sitting with a book, completely unaware of the environment.

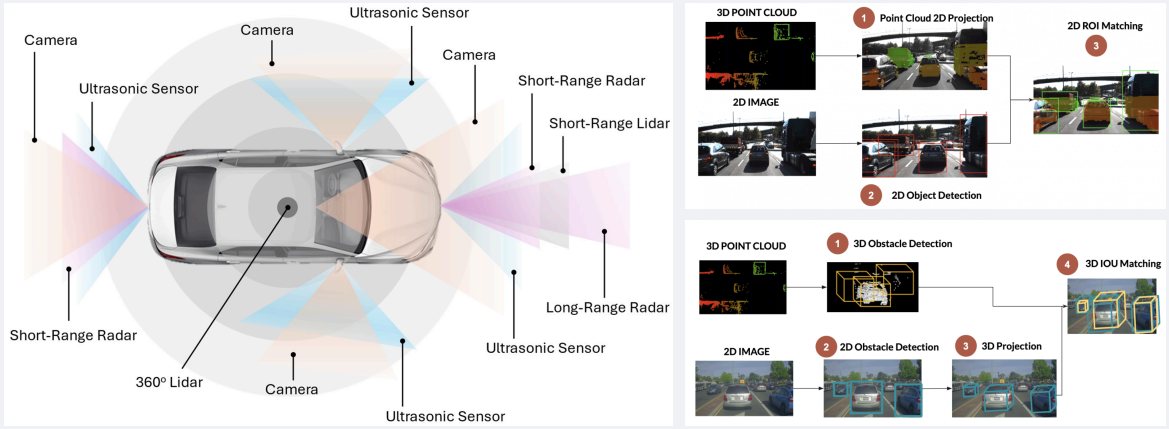
This level is not available anywhere in the world yet, and it is still believed to be at least a decade away before being commercially available to consumers [19]. This is most commonly the level depicted in futuristic sci-fi movies and games. Movies like "Minority Report" (2002), "Total Recall" (1990), and "Knight Rider" (1982) feature Level 5 AVs used for various tasks, such as autonomous driving, autonomous taxi services, and even fighting crime, respectively. Level 5 vehicles with personalities is a common trait in sci-fi, with the Delamain AI cab service from the game "Cyberpunk 2077" (2020) being a prime example.

* * *

To achieve Level 3 autonomy and higher, the vehicle must be able to understand the environment around it. This is done by various means and is done largely the same across the industry, with a few outliers. The most common way to understand the environment is through the use of sensors, such as cameras, radar, and lidar. Cameras are a fairly recent addition to the AV technology stack. They are used to detect and understand the environment around the vehicle, such as detecting pedestrians, cyclists, and other vehicles [20]. This requires a fair amount of computing power and efficiency, as the vehicle must be able to process the data from the cameras in real-time. The technology required is relatively

new, which is why it is only in recent years that cameras have become a mainstay in AVs, at least in the context of autonomous driving. Drivers have had rear-view and Bird's Eye View (BEV) cameras for many years, however, these were only to assist said driver. Before cameras, it was common for vehicles to be equipped with radar and lidar systems. These systems offer a much higher resilience to adverse conditions, such as rain and fog, but they are also much more expensive. Lastly, ultrasonic sensors are used to detect objects in close proximity to the vehicle, such as when parking. These sensors are not used for understanding the broader environment, but they are used for detecting objects around the vehicle. A typical configuration os these sensors is shown in Figure 2a.

These systems are often used in conjunction with one another to create a single coherent picture of the environment. This is what is known as sensor fusion, where the deluge of data from the various sensors is combined. Not only does this create a coherent picture, but it also draws on the strengths and weaknesses of each of the aforementioned technologies. For example, a camera has great spatial resolution and little noise, but is poor at estimating velocity and distance where radar and lidar shine, respectively [21]. There are 3 types of sensor fusion classifications: Low-level, mid-level, and high-level. Low-level is considered early fusion, and the others late fusion.



(a) (b)

Figure 2. (a) shows a typical sensor configuration, showcasing the large amount of different sensors on a single vehicle. (b) shows the difference between Early (top) and Late (bottom) fusion. Image source: (a) Yeong *et al.* [22], (b) Think Autonomous [23].

Low-level fusion is considered early fusion, as it combines the raw data streams, i.e. camera pixels, lidar points, etc., from the sensors before any processing is done. While this method retains the most information possible from the sensors, it is also very computationally heavy [22]. Mid-level fusion is considered late fusion, as it combines the intermediate representations. For instance, the vehicle's camera and radar might recognize a vehicle, then it recognizes those two representations as representing the same object. Finally, high-level fusing is also considered late fusion, as the combine the mid-level fusions with positional tracking. This is often through the use of probabilistic filters.

The most popular of these is the Kalman filter [24]. It is a powerful recursive algorithm used in sensor fusion to estimate the state of a dynamic system by combining measurements from multiple, potentially noisy sensors over time. It consists of two main steps: the predic-

tion step and the update step. In the prediction step, the Kalman filter uses a mathematical model of the system to predict the next state based on the current state and control inputs:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{1}$$

where $\hat{x}$ is the state estimate, F_k is the state transition model, B_k is the control input model, and u_k is the control input. Then, in the update step, the filter combines the predicted state with the new measurement to produce an updated estimate:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\left(z_k - H_k \hat{x}_{k|k-1}\right) \tag{2}$$

where K_k is the Kalman gain, z_k is the measurement, and H_k is the observation model. By iteratively applying these steps, the Kalman filter fuses information from different sensors, accounting for their noise characteristics, to produce a statistically optimal estimate of the system's state. This sensor fusion is typically more accurate than relying on a single sensor, as it combines the strengths of each sensor type. This helps the vehicle understand its immediate environment, and the with help of technologies like Vehicle-to-Everything (V2X), it can further understand the world around it. With all of these technologies in a vehicle's stack, it can make informed decisions and navigate dynamically complex environments effectively.

With this robust and detailed understanding of the world, many different control methodologies have been developed to control vehicles and robots, designated as the control problem. The most common and simple method is the Proportional-Integral-Derivative (PID) controller. Briefly, PID controller always try to minimize the error between the measured state and a desired. This error is then subject to three different operations: proportional, integral, and derivative, each multiplied by some scalar. Each property influences things like speed, overshoot, and stability. A more complex method is the Model Predictive Control (MPC) method. This method uses a model of the system to predict the future states and optimize the control inputs over a finite time horizon. It is particularly useful for systems with constraints, such steering angle, acceleration, and velocity. MPC is often used in conjunction with path planning, where the vehicle must follow a specific path while avoiding obstacles.

While path-planning is covered in Section 3, how AVs achieve this will briefly be covered, for with this robust representation of the environment and the control methodologies, the vehicle can make informed decisions and navigate dynamically complex environments effectively. Today, this is largely done with the use of DL, but more manual methods exist. These methods are less generalizable than DL ones, but are robust in their own ways. One such method is built up around the concept of a state machine, known as rule-based methods. These are scenario specific rules that are applied at predefined objectives, such as slowing down automatically when seeing a red light. For DL approaches, data-driven approaches are common. Behavioural cloning and imitation learning are common when you have massive amounts of driving data. These methods are trained on the data, and learn to mimic the behaviour of the human driver. This is done by training a Neural Network (NN) to predict the control inputs based on the sensor data. DL approaches have

witnessed massive growth in the last decade, and is now the most common methodology for AVs.

2.2 Deep Learning

The groundwork for the modern Artificial Intelligence (AI) we see today, has been under development since the 1940′s. In 1943, Warren McCulloch and Walter Pitts proposed the artificial neuron [25]. Their goal was to create a model that acted like the human brain, in that brains are made up of constant firing neurons. They proposed NNs can be modelled using a logical calculus based on the "all-or-none" firing principle of neurons. They demonstrate that neural activations correspond to logical statements. This provides a systematic way to analyse and predict neural behaviour, but also lays the theoretical groundwork for later advances in computational neuroscience and AI. Already in 1951, Minsky and Edmonds presented the Stochastic Neural Analog Reinforcement Calculator (SNARC), which consisted of 40 artificial neurons [26]. As will become the norm, this network was trained by adjusting the strengths of the connections between neurons based on the outcomes of the previous trials.

Another concept introduced early, was what would come to be known as reinforcement learning. Arthur Samuel, often called the father of machine learning, introduced the Samuel Checkers-Playing program in 1952, which learned the game of checkers via self-learning, improving its skills over time by playing many more games than a human ever could [27]. The term AI was coined in 1956, but it was not until 1958 Frank Rosenblatt developed the first Artificial Neural Network (ANN), which he named the perceptron, which was a single-layer NN that could learn to classify patterns [28]. Building on the early breakthroughs, the following decades saw an expansion of ideas that sought to overcome the limitations of single-layer NNs. Researchers began experimenting with multi-layered architectures and early variants of backpropagation [29]. In general, this period laid important groundwork for the following decades in the research and development of AI and, subsequently, DL.

One of the most important breakthroughs came in the form of the rediscovery and refinement of backpropagation [30]. This sparked the renaissance in NN research, as interest in learning from data was renewed. This breakthrough enabled the development of DL architectures, which could learn complex patterns and representations from large datasets. The following decades saw the integration of statistical learning methods with NN architectures that further enhanced their accuracy and robustness. What is considered the first point of contact with AI, came in 2012 with the release of AlexNet [31]. This model was capable of classifying images with a high degree of accuracy, paving the way for modern learning algorithms and DL architectures.

Looking at the highlights of the previous paragraphs, the most important take-aways are as follows. McCulloch and Pitts stated that "At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse." which, as will be presented, is exactly how modern machines are taught to think and learn. SNARC used a method of

adjusting the strengths of the connections between neurons based on the outcomes of the previous trials. Samuel's Checkers-Playing program used a method of self-learning, which is the basis of modern reinforcement learning. Rosenblatt's perceptron was the first ANN, introducing the concept of layers in a NN, later expanded to the modern Multilayer Perceptron (MLP).

The MLP is the most common and simple type of NN used in DL. It consists of an input layer, one or more hidden layers, and an output layer. Each layer consists of a number of neurons, which are connected to the neurons in the previous and next layers. The connections between the neurons are weighted, and these weights are adjusted during training to minimize the error between the predicted output and the actual output. The training process is done using backpropagation, which is a method for calculating the gradients of the weights with respect to the error. This is done by applying the chain rule of calculus to calculate the gradients of each weight in the network.

This process will now be broken down into its components, with the intent of creating a clear understanding of how MLPs work, and how they are trained. This will act as a springboard for understanding the more complex architectures and methods used in this work.

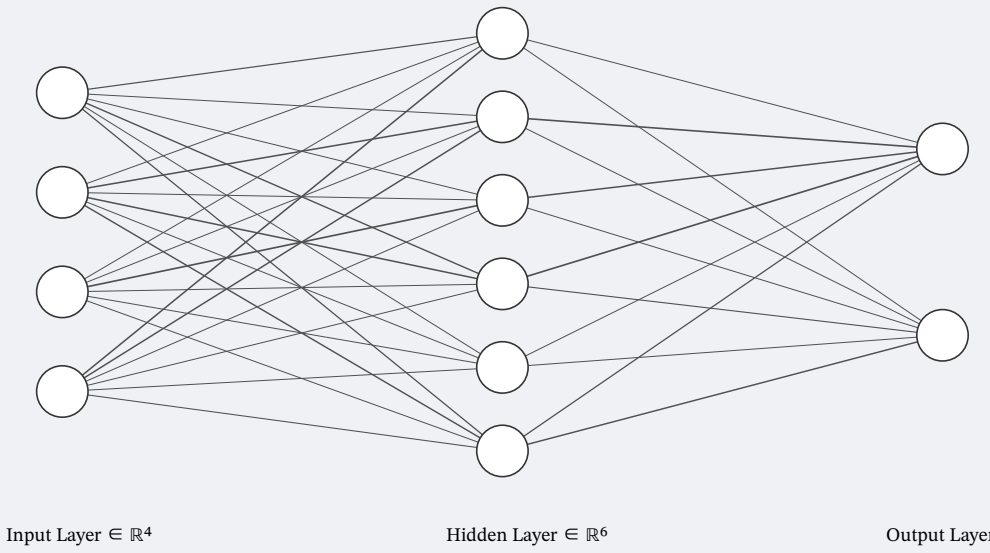


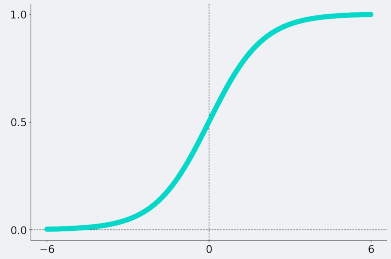
Figure 3. Multi-Layer Perceptron (MLP).

MLPs are composed of a number of layers, each consisting of a number of neurons. Each neuron is connected to the neurons in the previous and next layers. The connections between the neurons are weighted (as shown by the different opacities of the connections in Figure 3). MLPs are often what is called "fully connected", meaning that each neuron in a layer is connected to every neuron in the next layer. This is done to allow for the maximum amount of information to be passed between the layers.

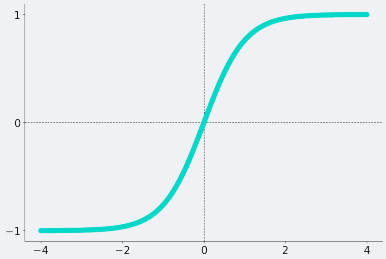
To start from the smallest possible unit, the neuron, it is important to understand how they work. During a forward pass through a NN (giving it some input feature), the value of each neuron is calculated as follows:

$$z = \boldsymbol{w}^T \boldsymbol{x} + b = \sum_i \boldsymbol{w}_i x_i + b \tag{3}$$

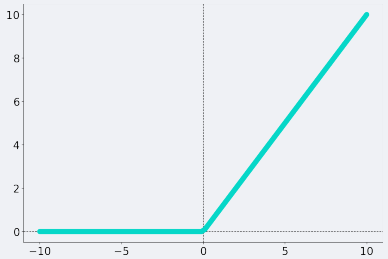
where $\boldsymbol{w}$ is the weight vector, $\boldsymbol{x}$ is the input vector, and b is the bias term. The bias term is a constant that is added to the weighted sum of the inputs. This allows the neuron to learn a threshold value, which is important for learning complex patterns. The output of the neuron a is then calculated using an activation function $a = \varphi(z)$, which introduces non-linearity into the model. Three of the most common activation functions φ are shown below:



$\sigma(z) = \frac{1}{1+e^{-z}}$



$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



$\mathrm{ReLU}(z) = \max(0, z)$

The first of the three is the sigmoid function. It maps the input to a value between 0 and 1, making it particularly useful for binary classification tasks. tanh scales the output to a value between -1 and 1, often leading to faster convergence. The last one, ReLU, is the most common activation function used in DL today. It is defined as $\mathrm{ReLU}(z) = \max(0, z)$, meaning that it outputs the input directly if it is positive, and 0 otherwise. This function is computationally efficient and helps mitigate the vanishing gradient problem, which can occur with sigmoid and tanh functions. The vanishing gradient problem occurs when the gradients of the weights become very small, making it difficult for the model to learn.

Another important activation function is the softmax function, which is used in the output layer of a MLP for multi-class classification tasks. It converts the raw output logits into probabilities by exponentiating each logit and normalizing them:

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4}$$

which ensures that the sum of the probabilities is equal to 1. Now, here is an example of a forward pass through the MLP. First, the input features $\boldsymbol{x}$ is passed through the hidden layer:

$$\begin{aligned} \boldsymbol{z}^{(1)} &= \boldsymbol{W}^{(1)} \boldsymbol{x} + b^{(1)} \\ \boldsymbol{a}^{(1)} &= \varphi\big(\boldsymbol{z}^{(1)}\big) \end{aligned} \tag{5}$$

where $\boldsymbol{W}^{(1)}$ is the weight matrix for the first layer, $\boldsymbol{z}^{(1)}$ is the weighted sum of the inputs, and $\boldsymbol{a}^{(1)}$ is the output of the first layer. The output of the first layer is then passed through the output layer:

$$\begin{aligned} \boldsymbol{z}^{(2)} &= \boldsymbol{W}^{(2)}\boldsymbol{a}^{(1)} + b^{(2)} \\ \hat{y} &= \text{softmax}\big(\boldsymbol{z}^{(2)}\big) \end{aligned} \tag{6}$$

where, in this example, the output $\hat{y}$ is put through a softmax activation to get probabilities on the output. Depending on the task of the NN, different loss functions are used. Loss functions are used to measure the difference between the predicted output and the actual output. For the task of regression (predicting a continuous value), the most common loss function is the Mean Squared Error (MSE) loss, which is defined as:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i}(\hat{y}_i - y_i)^2 \tag{7}$$

where N is the number of samples, $\hat{y}$ is the predicted output, and y is the actual output. For classification tasks, the most common loss function is the Cross-Entropy loss, which is defined as:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i} y_i \log(\hat{y}_i) \tag{8}$$

which calculates the dissimilarity between the predicted distribution $\hat{\boldsymbol{y}}$ and actual distributions $\boldsymbol{y}$. The binary version, called Binary Cross-Entropy (BCE), is also commonly used for binary classification tasks, and is defined as:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \tag{9}$$

where y_i is the actual label, and $\hat{y}_i$ is the predicted probability of the positive class. Once the forward pass is complete and the loss L is calculated using a function like MSE or Cross-Entropy, the goal of training the neural network is to minimize this loss. This done through what is called gradient descent. Gradient descent is a method for mathematical optimization, meaning by doing certain operations, a network's parameters can reach minima, where the error, or loss, is the smallest it can be. This is done by calculating the gradients of the loss with respect to the parameters, and updating the parameters in the opposite direction of the gradients. The goal is to reach what is called a local minima, where the loss is minimized. This minima symbolizes the best (smallest) loss that a network can achieve depending on its weights and biases.

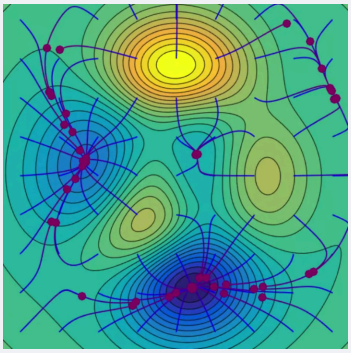


Figure 4. Gradient descent. Image source: WikiPedia [32]

Formally, this is known as backpropagation, which is a method for calculating the gradients of the loss with respect to the parameters. This is done by applying the chain rule of calculus to calculate the gradients of each weight in the network. So, the way to find the gradient for a weight w_{ij} connecting neuron i to neuron j in the next layer is to calculate the partial derivative of the loss with respect to that weight:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \tag{10}$$

where

$$z_j = \sum_i w_{ij}x_i + b_j \quad \Rightarrow \quad \frac{\partial z_j}{\partial w_{ij}} = x_i \tag{11}$$

meaning the gradient is

$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i \tag{12}$$

where δ_j is the error term for neuron j, calculated using one of the aforementioned loss functions. With this, it is now time to optimize the weights. This is done by using one of the optimization algorithms available. Two of the most commonly used optimizers are Stochastic Gradient Descent (SGD) and Adam. SGD is a simple and effective optimization algorithm that updates the weights using the gradients calculated during backpropagation. The update rule for SGD is:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta\frac{\partial L}{\partial w_{ij}} \tag{13}$$

where η is the learning rate, which controls the step size of the update. The learning rate is often a very small value, as to help NNs converge to a local minima slowly. A fair bit more involved and extremely influential is the Adam optimizer. Adam means Adaptive Moment Estimation, and is an adaptive learning rate optimization algorithm. It combines the advantages of two other extensions of SGD: momentum and RMSProp. It maintains two moving averages for each parameter: one for the gradients (first moment) and one for the squared gradients (second moment). It also includes a bias-correction mechanism to counteract the initialization bias of the first moment estimates. The update rules for Adam at time t are as follow:

1. Compute the gradients: $g_t = \frac{\partial L}{\partial w_t}$
2. Update the biased first moment estimate: $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$
3. Update the biased second moment estimate: $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
4. Bias-correct the moment estimates: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$, $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
5. Update the parameters: $w^{(t+1)} = w^{(t)} - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}$

where β_1 and β_2 are the decay rates for the first and second moment estimates, respectively. These are typically very close to 1, commonly set to 0.9 and 0.999. ε is a small constant added to prevent division by zero, and is typically set to 10^{-8}. Both m_0 and v_0 are initialized as 0. While a constant learning rate η is often good enough, too small of a constant η may result in the models being stuck in the closest local minima, which is likely not the best. A higher η is not exactly the way to combat this, as the training may become highly unstable, if the changes in weights are too large. This is where learning rate scheduling comes in. Learning rate scheduling is a technique used to adjust the learning rate during training.

There are several different types of learning rate schedules, such as step decay, exponential decay, and cosine annealing.

Step decay is the simplest, where the learning rate is reduced by a factor every few epochs. Exponential decay is largely the same, but it is constantly decreasing the learning rate each epoch. First, the step decay defines the learning rate as:

$$\eta^{(t)} = \eta_0 \cdot \gamma^{\left\lfloor \frac{t}{\text{step-size}} \right\rfloor} \tag{14}$$

where t is the epoch, η_0 is the initial learning rate, and γ is the decay factor. $\lfloor \cdot \rfloor$ denotes the floor function. Step decay is simple and can help a model settle into a local minima, but it is not very flexible. The non-smooth nature of the floor function used in the step decay can lead to abrupt changes in the learning rate, which can cause instability. Furthermore, it requires careful tuning of the step size and decay factor. Exponential decay combat some of these disadvantages by using a continuous decay function. The learning rate is defined as:

$$\eta^{(t)} = \eta_0 \cdot e^{-k \cdot t} \tag{15}$$

where k is the decay constant. This allows for a smoother and more gradual decrease in the learning rate, which can help the model converge more effectively. However, it still requires careful tuning of the decay constant k. It is, however, still a smooth uniform decay, which means it doesn't help the model potentially escape poor local minima. This is where cosine annealing comes in. Cosine annealing is a more advanced learning rate schedule that uses a cosine function to adjust the learning rate. The learning rate is defined as:

$$\eta^{(t)} = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{\pi t}{T}\right)\right) \tag{16}$$

where $\eta_{\min}$ is the minimum learning rate, $\eta_{\max}$ is the maximum learning rate, and T is the total number of epochs for the annealing cycle. This cosine function provides a periodic transition between high and low learning rates. When coupled with warm restarts (occasionally setting $\eta^{(t)} = \eta_{\max}$), this scheduler helps the optimizer escape local minima and thereby increase performance. The downsides are that it is more complex than the other schedulers, and it requires careful tuning of more parameters than both step and exponential decay.

Finally, some other common concepts w.r.t training are these:

- Regularization: A technique used to prevent overfitting by adding a penalty term to the loss function. The most common form of regularization is L2 regularization, which adds a term to the loss function that is proportional to the square of the weights. This encourages the model to learn smaller weights, which can help prevent overfitting. L2, or weight decay, is defined as: $L_{\text{total}} = L_{\text{data}} + \lambda \left\| w \right\|_2^2$.
- Dropout: A technique used to prevent overfitting by randomly dropping out a fraction of the neurons during training. This forces the model to learn more robust features and prevents it from relying too heavily on any one neuron.
- Batch Normalization: A technique used to normalize the inputs to each layer in the network. First the batch mean and variance are found:

$$\begin{aligned} \mu_B &= \frac{1}{m}\sum_{i=1}^{m} x_i, \\ \sigma_B^2 &= \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \end{aligned} \tag{17}$$

where m is the batch size. Then, the inputs are normalized:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{18}$$

where ε is a small constant added to prevent division by zero. Finally, the normalized inputs are scaled and shifted:

$$y_i = \gamma \hat{x}_i + \beta \tag{19}$$

where γ and β are learnable parameters that allow the network to scale and shift the normalized output.

In summary, the architecture and training process of the MLP has been presented in detail. By starting with creating an understanding of how each neuron worked, to how they are connected, and how the entire MLP is trained, a solid understanding of the MLP has been created, and generally how to train NNs. The forward and backward propagation was presented, as well as the loss functions and optimizers. The most common activation functions were presented, as well as the most common learning rate schedulers. With all of these methods, it is possible to train our MLP at a task we desire. However, the MLP is too simple of an architecture to learn any really complex tasks. This is where the more complex architectures and methods come in.

* * *

Before moving on to the complexities introduced in the subfield of computer vision, it is important to note that DL consists of three main learning paradigms. The first, and most common, of these is supervised learning. Supervised learning is concerned with training models using labelled data. This means that the model is trained on a dataset where the input features are paired with the corresponding output labels, i.e. the dataset consists of pairs (x, y) where x is the input features and y is the corresponding labels, often referred to as the ground truth labels. With each of these pairings, the training process involves minimizing a loss function, as described earlier, between the input features x and the ground truth labels y. Therefore, this learning paradigm is often used for image classification tasks, speech recognition, and natural language processing tasks. In other words, supervised learning is for when you know how you want the outcome to look by coming as close to the ground truth as possible.

Alternatively, you might not know exactly how your output should look. This is where unsupervised learning comes in. Unsupervised learning is concerned with training models using unlabelled data. This means that the model is trained on a dataset where the input features are not paired with any output labels. The goal of unsupervised learning is to learn the underlying structure of the data, such as clustering similar data points together or reducing the dimensionality of the data. This learning paradigm is often used for tasks

such as clustering, anomaly detection, and dimensionality reduction. Furthermore, this paradigm is typically used when training generative models. Generative models are models that learn to generate new data points that are similar to the training data. This is done by learning the underlying distribution of the data, and then sampling from that distribution to generate new data points. This task is most often seen in image generators, such as Generative Adversarial Networks (GANs).

Finally, Reinforcement Learning (RL) is a learning paradigm where a model learns through interactions with the environment. The model learns to take actions in an environment to maximize a reward signal. This is done by learning a policy, which is a mapping from states to actions, and a value function, which is a mapping from states to expected rewards. The goal of reinforcement learning is to learn a policy that maximizes the expected cumulative reward over time. This learning paradigm is often used for tasks such as game playing, robotics, and autonomous driving. Popular examples include AlphaGo [X] beating the world's greatest Go player, and OpenAI Five [X] beating the world champions in Dota 2.

These three paradigms broadly cover the methods used to train different models for different tasks. Each offers their strengths and weaknesses. Labelling data can be a time-consuming task, but create faster more targeted training, where unlabelled data and environmental interaction may reach beyond the performance initials goals. Each paradigm has their own objectives when training, from minimizing some prediction error to discovering hidden patterns to maximizing some cumulative reward. This also means that each paradigm lend themselves to achieve specific goals, such as classification, anomaly detection, and decision making, respectively. Computationally, they also offer different complexities: supervised learning can vary a lot depending on the task, but is typically moderately complex; unsupervised learning is often more complex, as the model must learn the underlying structure of the data; and reinforcement learning is often the most complex, as it requires the model to learn through iterative interactions with the environment.

These paradigms form the backbone of machine learning techniques used across various domains, including computer vision. Specifically, supervised learning has driven significant advancements in tasks like image classification, where labelled datasets guide models to recognize intricate visual patterns. One landmark achievement highlighting the potential of supervised methods in computer vision is AlexNet, which is widely regarded as the starting point of modern artificial intelligence.

2.2.1 Computer Vision

AlexNet is widely considered to be the first point of contact with what we classify as AI today [31]. It was an image classification model that was trained on the ImageNet dataset, and was capable of classifying images with a high degree of accuracy. This was the first NN to clearly pass other machine learning methods, such as kernel regression and support vector machines, in image classification tasks. This was a major breakthrough in the field of computer vision, opening the field to more researchers by proving the capabilities if NNs. Since its release, the interest in computer vision tasks has exploded, and is thus deployed in

a wide range of applications, such as image classification, object detection, image segmentation, pose estimation, and image generation.

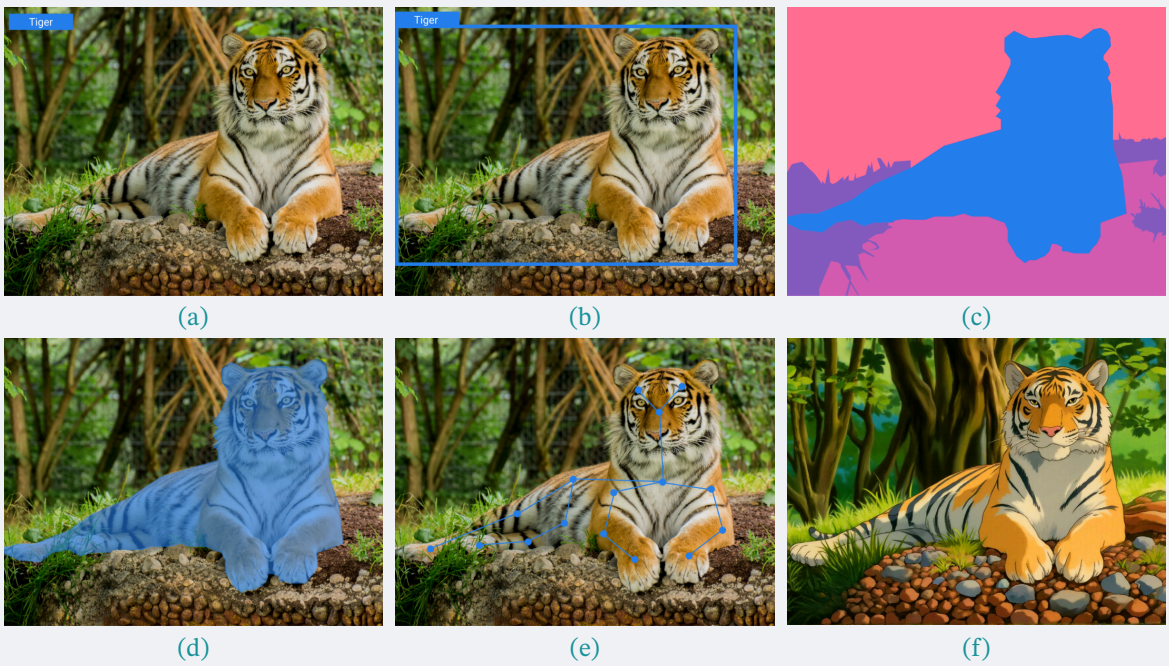


Figure 5. (a) Image classification. (b) Object detection. (c) Semantic segmentation. (d) Instance segmentation. (e) Pose estimation. (f) Image generation. Image source: (a-e) viso.ai [33] (f) Altered with OpenAI 4o [34].

The image classification task is the most simple of the computer vision tasks. It is concerned with classifying an image into one of a number of classes. This task, which was AlexNet's main task, is shown in Figure 5a. As shown, it is concerned with labelling the entire image as one class. It should be noted "class" means the name of a group of similar things, like the example shows the class of the image is "tiger". This ideas of classes, becomes important when moving on to the other CV tasks. The next task is object detection, which is concerned with detecting and localizing objects in an image. This task is shown in Figure 5b. It is concerned with not only classifying the image, but also localizing the objects in the image. This is done by drawing bounding boxes around the objects in the image, and classifying them. Note that there may be multiple instances of the same objects in the image, each requiring their own bounding box for correct classification.

The next task is semantic segmentation, which is concerned with classifying each pixel in the image. This task is shown in Figure 5c. It is concerned with classifying each pixel in the image into one of a number of classes. For this task, the model needs to be trained to not just identify the objects in the image, but also to identify the boundaries of the objects. This is done by creating a class label or class mask in the training data. A class mask is an image where each pixel is assigned a class label, and as mentioned prior, this class mask is the ground truth for said image. Closely related is the task of instance segmentation, which is concerned with classifying each pixel in the image, but also differentiating between different instances of the same object. This task is shown in Figure 5d. The main difference between these two segmentation tasks, is the fact that semantic segmentation does not differentiate between different instances of the same object. This means that if there are two objects of the same class in the image, they will be classified as the same object. This is

not the case for instance segmentation, where each instance of the same object is classified as a different object.

The next task is pose estimation, which is concerned with estimating the pose of an object in the image. It can be split into two separate tasks of its own: estimation of the 3D position of an object within an image and estimating the pose of the isolated object. The latter of these is shown in Figure 5e, where a rigging skeleton is drawn over the object to indicate its pose. This is done by estimating the position of the joints in the image, and drawing lines between them. The former task is concerned with estimating the 3D position of an object in the image, often achieved through a cascade of methods, such as 3D reconstruction, depth estimation, and multi-view stereo techniques.

The final task is image generation, which is concerned with generating new images from a given input. This task is shown in Figure 5f, where the image of the tiger has been altered to mimic a specific art style. This is a very common usage of image generators, but they can also create images from the ground up. In GANs, this is done by training the model on a dataset of images, and then generating new images that are similar to the training data. Other methods like diffusion iterate of noisy images, slowly refining them to create a new image based on some description. This is done by training the model to learn the underlying distribution of the data, and then sampling from that distribution to generate new images.

With these tasks in place, the methods for which either is achieved will now be presented. AlexNet consisted mainly of convolutional layers, with the occasional pooling layer. Convolution is a very important mechanic within the field of computer vision, as it allows for models to gain an understanding of the features within an image.

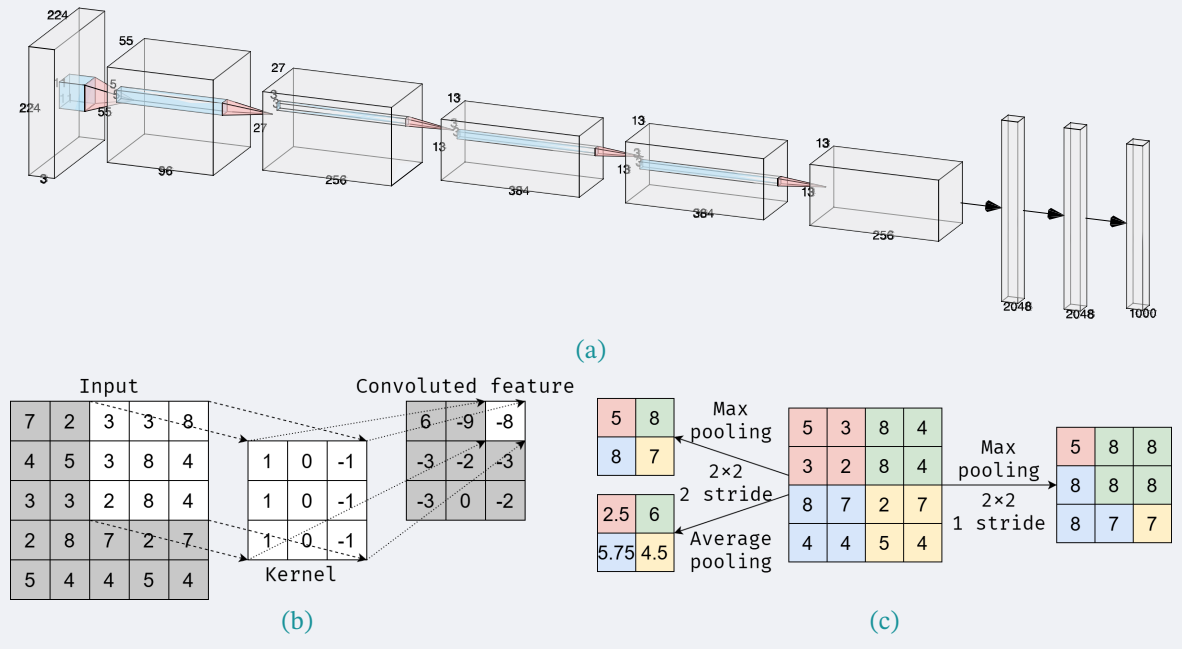


Figure 6. AlexNet architecture.

Convolution is achieved by convolving a kernel over the input image. This is done by sliding the kernel over the image, and at each position, calculating the dot product between the kernel and the image. This is done by multiplying each element in the kernel with the

corresponding element in the image, and then summing the results. The result of this operation is a new image, called a feature map. This map is typically smaller than the original image, as the kernel only stays within the bounds of the image itself. This can be bypassed by padding the input image with zeros, which allows the kernel to convolve along the very outer edge of the image. For input image often containing three channels (RGB), the kernel is convolved with each channel separately, or, rather, the kernel is three dimensional. This outputs one channel, which is why many different kernels are used. As seen in Figure 6a, the first layer of AlexNet after the input layer consists of 96 channel, with the next being 256 channels.

Other than padding, other factors come into the usage of convolution kernels. The stride is the number of pixels the kernel is moved at each step. A stride of 1 means that the kernel is moved one pixel at a time, while a stride of 2 means that the kernel is moved two pixels at a time. This can be used to reduce the size of the feature map, as the kernel will skip some pixels in the image. This is also seen in the figure, where the kernel is a large 11×11 kernel with stride 4. This results in the output being significantly smaller than the input image. Next, the kernel size is the size of the kernel itself, and is typically a small odd number, such as 3, 5, or 7. The kernel size is important, as it determines the size of the receptive field of the model. The receptive field is the area of the input image that the kernel is able to see at any given time. A larger kernel size means a larger receptive field, which allows the model to learn more complex features. However, a larger kernel size also means more parameters to train, which can lead to overfitting.

Padding, stride, and kernel size are also relevant to the other operation incorporated into AlexNet: pooling. Pooling is a downsampling operation that reduces the size of the feature map. This is done by taking the maximum or average value of a small region in the feature map, and using that value as the new value for that region. This is done by sliding a pooling kernel over the feature map, and at each position, calculating the maximum or average value of the region covered by the kernel. The result of this operation is a new feature map, which is smaller than the original feature map.

These concepts of stride and pooling are illustrated in Figure 6c. The size of the pooling kernel is the often the same as the stride, meaning the kernel does not encompass any overlapping pixels. As shown with the right-hand side example, the stride is 1, which results in overlapping regions. This is a rather undesired effect, so a stride equal to the kernel size is often used. This is shown in the left-hand side example, where the stride is 2, and the pooling kernel is 2×2. This results in a non-overlapping pooling operation, which is often desired. The most common pooling operations are max pooling and average pooling. Max pooling takes the maximum value of the region covered by the kernel, while average pooling takes the average value of the region covered by the kernel. Examples of both are shown in Figure 6c on the left-hand side of the input.

Finally, a few more components of the AlexNet architecture are worth mentioning. After each convolutional layer, a ReLU activation function is applied. This is done to introduce non-linearity into the model, which allows it to learn more complex features. The final layer of AlexNet is a fully connected layer, which is used to classify the image into one of the classes. This is done by flattening the feature map into a vector, and then passing it

through a series of fully connected layers. The final output is then passed through a softmax activation function, which converts the output into probabilities for each class. AlexNet utilizes dropout for regularization, as to prevent overfitting. This is done by randomly dropping out a fraction of the neurons during training, which forces the model to learn more robust features. Furthermore, to prevent overfitting, during training, the input images are put through augmentation. This will be covered in greater detail in Section 4.3.2. Finally, AlexNet also uses Local Response Normalization (LRN) layers. It works by normalizing the activations of neurons in a local region across the channel dimension.

Moving beyond the convolutional NN (CNN) architecture that is AlexNet, many other methods and architectures exist to help machines understand images. Closely related by the fact that it is also considered a CNN, is the U-Net architecture, which is widely used for image segmentation tasks [X]. U-Net is a fully convolutional network that consists of an encoder and a decoder. The encoder is responsible for downsampling the input image, while the decoder is responsible for upsampling the feature map back to the original size. This is done by using skip connections between the encoder and decoder, which allows the model to learn both low-level and high-level features. The upsampling, opposed to downsampling/pooling, is done by using transposed convolutions, which are the reverse of normal convolutions. Whereas normal convolution reduce patches to singular values, transposed convolution does the opposite; it take a singular value and expands it to a patch using a kernel. This typically doubles the height and width of the feature map. Convolutions in general see heavy usage in CV tasks, as the kernel they use are learnable, meaning they are affected by backpropagation, meaning they can learn.

* * *

In 2020, ChatGPT hit the ground running, exploding into the public conscience and making AI a mainstream tool that everyone suddenly knew about [X]. This leap in Natural Language Processing (NLP) was made possible by the introduction of the Transformer architecture, which was introduced in 2017. In their landmark paper “Attention is All You Need”, Vaswani *et al.* [X] introduced the Transformer architecture, which revolutionized the field of NLP. The transformer architecture is based on the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence when making predictions. To understand self-attention, imagine reading a sentence. When you read a word, you don’t just look at the word itself, but also at the words around it. In the sentence “The cat chased the mouse because it was fast”, the word “it” refers to “the mouse”. Self-attention allows a model to do the same computationally. The self-attention mechanism works by calculating how relevant a word in a sentence is to every other word in the same sentence. This allows the model to look at other parts of the input sequence and determine which parts are most important for predicting the next word. This mechanism is heavily utilized in the transformer architecture. Transformers can process all words in a sequence in parallel. This is a major advantage over recurrent NNs (RNNs), which process words sequentially. They typically consist of an encoder and a decoder, where the encoder processes the input sequence and the decoder generates the output sequence. Both use multiple layers stacked with self-attention and feed-forward networks. Thus, transformers helped technologies, like the Generative Pre-trained Transformers (GPTs), gain an incredible understanding of human language and explode into the public conscience.

Inspired by the self-attention mechanism and transformers, Dosovitskiy *et al.* [X] introduced the ViT architecture in 2021. This extended the capabilities of transformers into the field of computer vision. To make transformers applicable to images, the input image is divided into fixed-size patches, which are then flattened and linearly embedded into a sequence of tokens. This is very much akin to the way words in a sentence are processed in NLP. This allows the transformer to treat visual data in a similar way to text data, enabling it to learn complex relationships between different parts of the image through the self-attention mechanism. The ViT architecture consists of a series of transformer blocks, each containing a multi-head self-attention layer and a feed-forward network. The output of the final transformer block is then passed through a classification head to produce the final output. So, the vision architecture typically consists of three main steps:

Patch embedding: An input image is divided into equally-sized non-overlapping patches, typically 16x16 pixels each. These patches are flattened into vectors and transformed into embeddings through a learned linear projection, producing fixed-dimensional patch embeddings. **Transformer encoder**: The embeddings go through multiple stacked layers of transformer encoder blocks, each comprising two sub-layers: a multi-headed self-attention mechanism and a feed-forward neural network. Multi-headed self-attention computes attention scores to weigh the relevance of each patch relative to all others simultaneously. To retain spatial information, positional embeddings are added to the patch embeddings before being input into the transformer layers. **Classification head**: The processed embeddings from the transformer layers include a special classification token (`[class]`), prepended to the sequence of patch embeddings. The `[class]` embedding captures a global representation of the image. This global embedding is then passed through a fully-connected neural network to produce class predictions for image classification tasks.

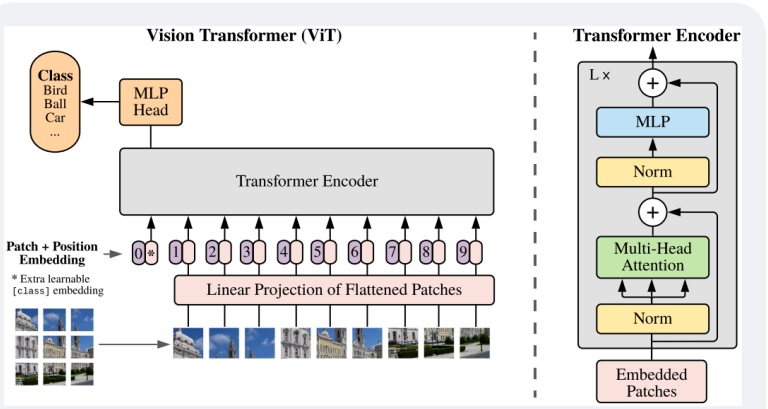


Figure 7. Vision Transformer architecture.
Image source: Dosovitskiy *et al.* [X]

Closely related to the ViT is the Swin Transformer, introduced by Liu *et al.* [X] in 2021. The Swin Transformer is a hierarchical transformer that uses a shifted windowing scheme to reduce the computational cost of self-attention. This is done by dividing the input image into non-overlapping windows, and then applying self-attention within each window. These windows are merged in the deeper layers of the network, achieving great highly accurate segmentation masks.

Now, a term I have not explained yet is “overfitting”. Overfitting is a common problem in machine learning, where the model learns the training data too well, and is unable to generalize to new data. This phenomenon is seen in the training and evaluation (or test or validation) graphs of a model’s training process. Early in the training process, the model’s performance on both the training and test sets improve as the model learns the task. However, as training continues, the model’s performance on the training set continues to improve, while the performance on the test set starts to degrade. This is a sign that the model is overfitting to the training data. This is visualized in Figure 8, where the training loss continues to decrease, while the validation loss starts to increase after training iteration 75.

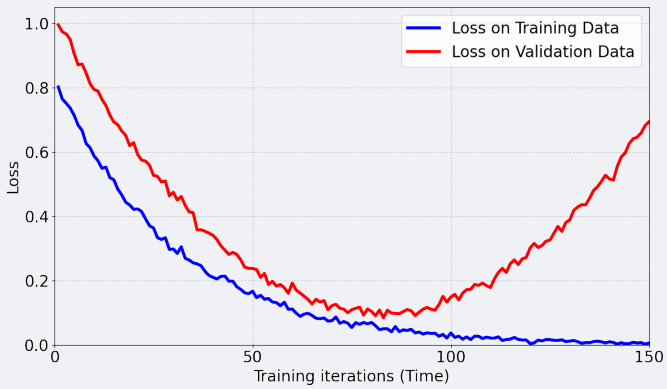


Figure 8. Training and evaluation loss plot.

To combat overfitting, several techniques are used. As presented already, dropout is commonly used to combat overfitting. This is done by randomly dropping out a fraction of the neurons during training, which forces the model to learn more robust features. Another common technique is early stopping, where the training process is stopped when the performance on the validation set starts to degrade. This is done by monitoring the validation loss during training, and stopping the training process when the validation loss starts to increase too significantly. As shown, the validation loss comes across as noisy, so before employing early stopping, it is important to be certain that the validation loss is getting consistently worse. Alternatives to early stopping is simply by using a checkpoint system, where the weights of the model, or its current state, are saved at regular intervals. Regularization is also a common method to combat overfitting. This is done by adding a penalty term to the loss function, which encourages the model to learn smaller weights. The most common form of regularization is L2 regularization, which adds a term to the loss function that is proportional to the square of the weights. This encourages the model to learn smaller weights, which can help prevent overfitting. L1 regularization is also used, which adds a term to the loss function that is proportional to the absolute value of the weights.

Finally, data augmentation comes in many shapes and sizes, literally. Data augmentation is a technique used to artificially increase the size of the training dataset by applying various transformations to the input data. A selection of these is shown in Figure 9. These data augmentation techniques are used to create new training samples by applying various transformations to the original image. This is done to increase the diversity of the training data, which can help improve the performance of the model. The shown examples are very common in computer vision tasks. Finally, a technique called cross-validation is often used. This methods splits the dataset into multiple subsets, or folds. During training, one of the folds is used for validation, while the others are used for training. This is done to ensure that the model is not overfitting to a specific subset of the data.

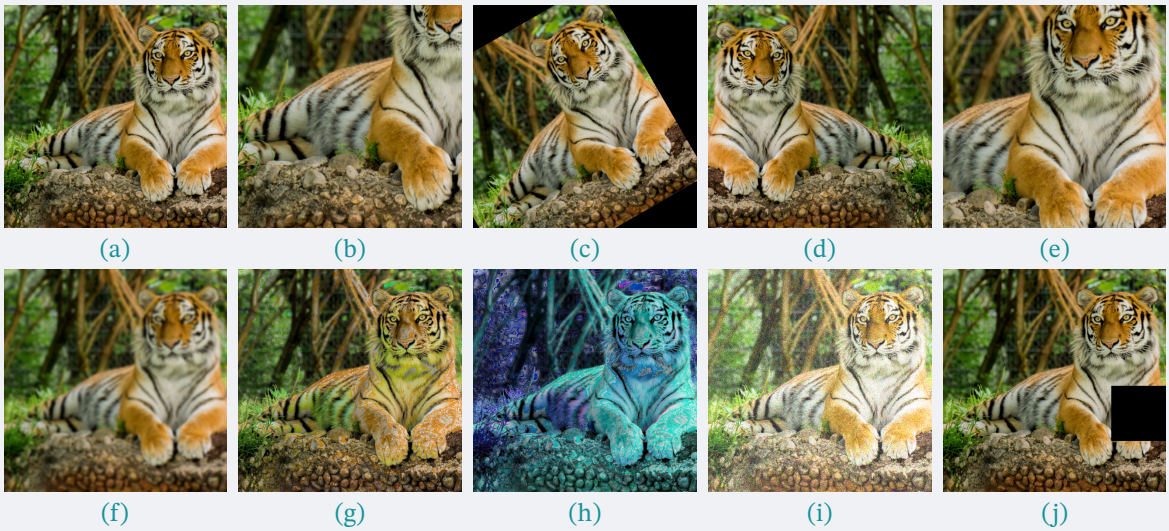


Figure 9. Data augmentation techniques. (a) Original image. Applied augmentation are (b) cropping, (c) rotation, (d) flipping, (e) zooming, (f) blurring, (g) saturation adjustment, (h) hue adjustment, (i) noise addition, and (j) occlusion. Image source: viso.ai [33].

The architectures, methods, and techniques present in modern deep learning and computer vision are vast and complex. However, these methods are only as effective as the data that fuels them. Selecting, generating, or collecting the right data for any given task is as crucial for the success of the model as the model itself.

2.2.2 Datasets

Datasets are the blood of any machine learning model. Entries flow through a model's structure, helping it learn to perform a specific task. In datasets, both the quantity and quality of the data are important. The more data a model has to learn from, the better it will perform. However, the quality and variety are also tantamount to the model's performance. If the data is not representative of the task, the model will not be able to learn to perform the task. If too little variety is present, it will not learn to generalize and only be able to perform the task on scenarios closely related to that which makes up its dataset.

There exists many different datasets for very different purposes. As presented earlier, the ImageNet dataset was key to the success of AlexNet. This dataset consists of millions of images with many thousands of classes. For the task of image classification, this dataset was very sufficient in providing the model with enough data to learn and generalize from. However, the base ImageNet dataset is insufficient for related CV tasks, such as object detection and segmentation. For these tasks, the dataset needs to be annotated with bounding boxes and class labels for each object in the image. This is a very time-consuming task to do from the ground up, which is why it is often to search for datasets that are already annotated and fit the desired task.

Creating datasets of any significant size can be a rather difficult task, where some tasks are simpler than others. For image classification, each dataset entry only requires a single image and a class label. This is a fairly simple task to do, as the most difficult part comes

from collecting the images, with labelling taking very little time per entry. For other tasks like object detection and segmentation, the task of labelling becomes much more difficult. This is because each entry requires a bounding box or segmentation mask for each object in the image. This is a very time-consuming thing to do, and the quality of the labels may be of worse quality than desired, meaning there also needs to be some form of quality control. This means that the creation will take even longer, as labels that are poorly made, might heavily affect the model's performance.

To mitigate errors in the labelling process, some move to simply make the dataset with synthetic data. This is done by using a simulator of some kind, ranging from a simple game engine to a full-fledged simulator. This is done by creating a virtual world, where the objects in the world are simulated. This allows for the creation of a dataset with perfect labels, as the simulator knows exactly where each object is in the world. Furthermore, making synthetic data can help increase the diversity in a dataset for areas where it might be difficult to obtain through real-world means. The most common problem with generating synthetic data, especially imagery of real-world scenarios, is the fact that the generated data does not look like real-world data. This can cause problems for models trained on the data, as they might not be able to generalize to real-world data as textures and shadows are less likely to look realistic.

For AVs, the datasets are often more complex than just images. They often consist of multiple sensors, such as LiDAR, radar, and cameras. This balloons the size of the dataset, resulting in singular entries being multiple gigabytes in size. The Waymo Open Dataset [X] is a prime example of this. It consists of 1950 entries, each consisting of 20 seconds of video at 10Hz. Each contains the data from a mid-range LiDAR, 4 short-range LiDARs, and 5 cameras. The entries contain labelled data for 4 object classes: vehicles, pedestrians, cyclists, and signs. Figure 10e shows an example of some LiDAR readings overlaid on the corresponding camera image. Waymo is a very detailed dataset, with a lot of data to learn from. However, it is also a very large dataset, which makes it difficult to work with on consumer-level computers. Competitors do exist, such as the nuScenes [X] dataset and the Argoverse [X] dataset.

Closely related are the lane-level datasets. These datasets are concerned with detecting and classifying lanes in images. This is a very important task for AVs, as it allows them to understand the road topology and navigate accordingly. The TuSimple dataset example shown in Figure 10d illustrates this task well, providing annotated lane markings for training and evaluation. Systems like AIM that manage the vehicle going through an intersection, datasets like the INTERACTION [X] dataset are needed. This dataset consists of overhead views of intersections, with annotated vehicles moving through them. This dataset is very useful for training models to understand the interactions between vehicles at intersections.

Finally, some satellite image-based datasets also exits for different purposes. The SpaceNet [X] dataset is concerned with the labelling of buildings and roads etc. in satellite images. This dataset consists of high-resolution satellite images and different annotations for the images, such as individual houses, both through bounding boxes and segmentation masks. For a bit wider scale, the DeepGlobe [X] dataset offers semantic masks of roads, building, and land cover.

(a) SpaceNet [X] (b) INTERACTION [X]

(c) DeepGlobe [X] (d) TuSimple [X] (e) Waymo [X]

Figure 10. Examples from datasets for AVs.

Common for all datasets, is the fact that they require a dataloader in order to be used in the program training the model. A dataloader is a class that is responsible for loading the data from the dataset and preparing it for training. This means that the dataset needs to be structured in a consistent way, such that the dataloader can work as seamlessly with it as possible. A dataset typically consists of a number of entries, each with their own attributes, such as ground truth labels, images, LiDAR and radar data, and other sensor data. Maintaining a consistent structure across all entries is important, otherwise a dataloader might get needlessly complex with unnecessary fail-safes.

Dataloaders often come with the ability to split a dataset into training and validation sets. This is done by randomly splitting the dataset into two parts, where one part is used for training and the other part is used for validation. This is commonly done by splitting it with a certain percentage going to either set. Alternatively, the dataset can be split beforehand, meaning it consists of predefined training and validation sets. This can be a good choice when the model could become prone to overfitting, which is likely to happen when the dataset entries are similar looking, such as with intersections. If techniques like cross-validation are used, the model is at some point trained on every entry, which for long training sessions will eventually lead to overfitting. This can, however, be mitigated with the use of data augmentation.

2.3 Satellite Imagery

Datasets can consist of many different views. Many datasets for AVs consist of images taken from a car driving around, typically consisting of many different cameras and sensors. Others, like is the focus in this project, consist of satellite images. Satellite imagery provides

a bird's eye view of the world. This is a very useful perspective for many tasks, such as road extraction and lane detection. Many different sources of satellite imagery exist, such as Google Maps, Azure Maps, and Sentinel, each offering their own capabilities through Application Programming Interfaces (APIs). The most important among these is the ability to get high-resolution static images of a specific location, preferably as close in dates as possible. This not only increases the likelihood of images being of a usable resolution, but also reflecting of the current state of locations.

The resolution of satellite images refers to the size of the pixels in the image. The less area a pixel covers, the more information and detail about the location is captured. Google Maps Static API offers a sub-meter spatial resolution, meaning each pixel represents about 30-50cm on the ground [35]. At this level of detail, things like road markings and lane boundaries can be detected. In contrast, if the resolution is at a meter-level spatial resolution, then these fine details are at best blurred and at worst completely lost since they represent potentially a fraction of the land making up said pixel. At this level, only the general structure may be inferred from the resulting satellite image. Thus, having a decently high-resolution image is very important for giving NNs a chance at understanding intersections. For example, it needs to be able to see road-marking that make up the lanes, especially important when certain lanes are for going certain ways, often marked by arrows; you would not want a vehicle go straight through an intersection when it has placed itself in a lane that is only for turning right.

Satellite images provide strong advantages to AVs. Firstly, they are not reliant on live images, meaning they can be used even when it is cloudy. This means that they can provide a consistent and reliable source of information about the environment, regardless of weather conditions. Secondly, they can help AVs understand the road topology of any intersection before arriving at it. This means that a vehicle can place itself in the appropriate lane beforehand, delivering a smoother experience to the driver and passengers. This is especially important for AVs, as they need to be able to navigate complex intersections without human intervention. Finally, satellite images can be used to create high-definition maps of the environment. These maps can be used to help AVs navigate and understand their surroundings, potentially even help them find paths that offer smoother rides or other desired traits of travelling to some destination.

In summary, this chapter has thoroughly established the theoretical foundation necessary for the work presented in this thesis. By exploring the taxonomy of autonomous vehicles and the technologies enabling their development, it has provided context for the broader application landscape. The in-depth overview of deep learning — from its historical roots to modern architectures — lays the groundwork for understanding the computational methods used in this project. Furthermore, the exploration of datasets and the unique role of satellite imagery highlights the importance of data quality and perspective in training effective models. With this background in place, the thesis now transitions into more closely related works before moving onto the more specific methodologies and experiments that form the core of this work.

3 Related Work

3.1 Path-planning

Path-planning is the task of having a certain amount of knowledge about the environment and finding a path from a starting point to a goal point. This task is one of the most fundamental tasks in the field of robotics and autonomous navigation, and has thus has a long history of improvement and evolution.

One of the first algorithms to be used for path-planning is the Dijkstra algorithm from 1959 [36], which is a graph search algorithm that finds the shortest path between nodes in a graph. The A* algorithm is another popular algorithm from 1968 that is used for path-planning, and is a combination of Dijkstra's algorithm and a heuristic function that estimates the cost of the cheapest path from a node to the goal node [37]. Some years later, the D* algorithm was introduced in 1994, which is an incremental search algorithm that finds the shortest path between nodes in a graph, and is an improvement over the A* algorithm [38]. D* has since become a very popular algorithm for path-planning in robotics, with improved alternatives like Focused D* the year after [39] and D* Lite from 2005 [40] proving use in real-world applications.

The concept of Artificial Potential Fields (APFs) was introduced in 1986 [41]. It assumes some repulsive field around obstacles to avoid and a pulling force towards the goal, resulting in autonomous robots navigating towards the goal while avoiding obstacles. This method is particularly effective in dynamic environments, where the obstacles are moving. It is, however, very prone to local minima and situations where it might get trapped [42], significantly reducing its effectiveness in complex environments, such as a tight hallway where the robot might fit but the calculated repulsive force is too great or if it encounters a dead-end or U-shaped obstacle, leading it to loop infinitely. Combined with other global path-planning algorithms, it has shown considerable success, especially in the field of swarm robotics [43], [44].

The Rapidly-exploring Random Tree (RRT) algorithm was introduced in 1998 [45], and is a popular algorithm for path-planning in robotics. It is a randomized algorithm that builds a tree of possible paths from the starting point to the goal point, and is particularly useful in high-dimensional spaces. A node is randomly chosen from the initial point. The intermediate node is then determined based on the movement direction and maximum section length. If obstacles are detected, the route in that direction is ignored. Otherwise, a new random point is selected. The RRT* algorithm was introduced in 2011 [46], improving

on the original with two small but significant modifications: a cost function that takes into account the distance between nodes and a re-wiring step that allows the tree to be restructured to find a better path. It has shown great usage in real-world applications regarding Autonomous Underwater Vehicles (AUVs) [47], [48], despite challenges regarding the need for information about large areas [49].

Other areas of research in path-planning include Genetic Algorithms (GAs) and Fuzzy Logic (FL). GA [50] is inspired by the process of natural selection where only the fittest organisms survive. Generally the algorithm works by generating a random population of solutions, and then selecting the most efficient ones by using some cost function. Then these selected solutions go through the crossover process where they are combined and mutated to generate new solutions. FL is another old method from 1965 used for path-planning [51], [52]. It depends on functions used in fuzzification, inference, and defuzzification. These functions are based on a descriptive classification of the input data, such as low, medium, or high collision risk. Based on the defuzzification process, the robot decides on the best path to take.

NNs are also finding their usage in the field. NNs are made to imitate the human brain's innate ability to learn. They are trained on data and learn how to react to it. They are used in the field, not necessarily for path-planning explicitly, but more in conjunction with other algorithms that use their output as input. I.e. a NN might be able to tell the controller where some obstacle is, meaning it is giving a helping hand to algorithms like APF. Akin to APF, RL models are taught to react to their surroundings, driving towards a goal and being rewarded and penalized for the actions that it takes, like how APF is moving towards a goal and avoiding obstacles due to the repulsive forces.

In summary, the evolution of path-planning – from early graph search methods like Dijkstra and A* to more adaptive techniques such as D*, RRT, and learning-based models – illustrates a steady push toward efficiency and robustness. Approaches like APFs, GAs, and FL add further flexibility, each with its own trade-offs. Together, these methods highlight the ongoing effort to balance computational efficiency with real-world challenges.

3.2 Intersection Management

4 Methodology

This section covers the methodology and work produced as part of this thesis. The first part to be detailed is the retrieval of satellite images. This includes the API usage and method used for signing URLs.

4.1 Satellite Imagery

ADD BIT ABOUT ZOOM LEVEL. INCLUDE COMPARISON IMAGES

Satellite imagery is a key component of this thesis project. The imagery will be used for both training and testing the DL models, by creating a dataset detailed in Section 4.3, and as input to said model during inference. This section covers the acquisition of satellite imagery, the process of signing URLs as required by the API, and the code created for these purposes.

This project utilizes Google Maps Static API as provided by Google Cloud Platform. The API allows for the retrieval of static map images at a given resolution and zoom level. This API was chosen due to its ease of use, the quality of the retrieved images, and the fact that it is free to use for a limited number of requests. The API is used to retrieve satellite imagery of a given location.

4.1.1 Image Acquisition

Google Maps Static API can retrieve images by forming requests with specific parameters that define the center, zoom level, size, and additional options for the map. For this project, images of type `satellite` are used, as they provide the highest level of detail for each retrieved image. Other types like `roadmap` or `terrain` do not provide enough detail to create a path that would realistically help navigate any kind of intersection as things like line markings are abstracted away.

To request an image, a URL is generated dynamically for the API, incorporating the required parameters. The parameters of the API request are as follows:

- `center`: The latitude and longitude of the center of the map (e.g. `41.30392`, `-81.90169`).
- `zoom`: The zoom level of the map. 1 is the lowest zoom level, showing the entire Earth, and 21 is the highest zoom level, showing individual buildings.
- `size`: The dimensions of the image to be retrieved, specified in pixels (e.g., `400x400`).

- `maptype`: Specifies the type of map to be retrieved. Options include `roadmap`, `satellite`, `terrain`, and `hybrid`.
- `key`: The API key used to authenticate the request.
- `signature`: Secret signing signature given by Google Cloud Platform through their interface.

Furthermore, the API allows for markers to be placed on the map, which can be used to highlight specific points of interest. This is, however, not relevant to this project.

4.1.1.1 URL Signing

While requests to the API can be made using only the API key, the usage is severely limited without URL signing. URL signing is a security measure that ensures that requests to the API are made by the intended user. The signature is generated using the API key and a secret key provided by Google Cloud Platform. The URL signing algorithm is shown in Algorithm 1 and is provided by Google [53].

Algorithm 1: URL Signing Algorithm (`sign_url`)

Input: URL, secret

url ← urlparse(URL)
secret_decoded ← base64_decode(secret)
signature ← HMAC_SHA1(secret_decoded, url.path + '?' + url.query)
signature ← base64_encode(signature)
URL_signed ← URL + '&signature=' + signature

Output: URL_signed

As input is the URL with filled parameters and the secret key. The algorithm generates a signature using the HMAC-SHA1 algorithm with the secret key and the URL to be signed. The signature is then base64 encoded and appended to the URL as a query parameter. The signed URL can then be used to make requests to the API.

4.1.2 Implementation

The main functionality of satellite imagery retrieval can be seen in Listing 1. An example of the output of the functionality can be seen in Figure 11.

```python
def get_sat_image(lat: float, lon: float, zoom: int = 18, secret: str = None, print_url: bool = False) -> requests.Response:
  if not secret:
    raise Exception("Secret is required")
  if not lat or not lon:
    raise Exception("Both lat and lon are required")

  req_url = f"https://maps.googleapis.com/maps/api/staticmap?center={lat},{lon}&zoom={zoom}&size=400x400&maptype=satellite&key={API_key}"
  signed_url = sign_url(req_url, secret)
  if print_url:
    print(signed_url)

  response = requests.get(signed_url)
  return response


def save_sat_image(response: requests.Response, filename: str = "map.png") -> None:
  if response.status_code != 200:
    raise Exception(f"Failed to get image, got status code {response.status_code}")

  with open(filename, "wb") as f:
    f.write(response.content)
```

Listing 1. Python functions used to retrieve and save satellite imagery (`get_sat_image` and `save_sat_image`)

Listing 1 shows two functions, `get_sat_image` and `save_sat_image`, that are used to retrieve and save satellite imagery, respectively. The `get_sat_image` function first checks if a `secret` and coordinates are provided as they are required. It then constructs a URL for the Google Maps Static API request and signs it using the `sign_url` function detailed in Algorithm 1. The signed URL is then used to make a request to the API, and the response containing the image is returned. This response can then be passed to the `save_sat_image` function, which saves the image to a file with the specified filename.

Listing 2 shows the Python implementation of Algorithm 1 with details following beneath.

```python
def sign_url(input_url: str = None, secret: str = None) -> str:
  if not input_url or not secret:
      raise Exception("Both input_url and secret are required")

  url = urlparse.urlparse(input_url)
  url_to_sign = url.path + "?" + url.query
  decoded_key = base64.urlsafe_b64decode(secret)
  signature = hmac.new(decoded_key, str.encode(url_to_sign), hashlib.sha1)
  encoded_signature = base64.urlsafe_b64encode(signature.digest())
  original_url = url.scheme + "://" + url.netloc + url.path + "?" + url.query

  return original_url + "&signature=" + encoded_signature.decode()
```

Listing 2. Python implementation of the URL signing algorithm (`sign_url`)

- `sign_url:1`: Function declaration for the URL signing algorithm. It takes in two parameters, `input_url` and `secret`.
- `sign_url:2-3`: A check is performed to ensure that both `input_url` and `secret` are provided.
- `sign_url:5`: The provided URL is parsed using the `urlparse` function from the `urlparse` library.
- `sign_url:6`: The URL to be signed is extracted from the parsed URL object.
- `sign_url:7`: The secret key is decoded from base64 encoding.

- `sign_url:8`: The signature is generated using the HMAC-SHA1 algorithm with the decoded secret key and the URL to be signed.
- `sign_url:9-10`: The resulting signature is then base64 encoded and the URL is reconstructed.
- `sign_url:12`: The signed URL is returned with the signature appended as a query parameter. An example output could be

  `https://maps.googleapis.com/maps/api/staticmap?center=41.30392,-81.90169&zoom=18&size=400x400&maptype=satellite&key=<api_key>&signature=<signature>`

  with filled `<api_key>` and `<signature>`.

A small `rotate_image` function was also created to rotate the retrieved image by some degrees, as the orientation of the satellite images can vary. The code can be seen in Listing 3. This is meant to help simplify the task performed by the model, as it alleviates the need to handle poorly angled images.

```python
def rotate_image(image_path, angle) -> None:
  image_obj = Image.open(image_path)
  rotated_image = image_obj.rotate(angle)
  rotated_image.save(image_path)
```

Listing 3. Python function to rotate an image by a specified angle (`rotate_image`)

https://maps.googleapis.com/maps/api/staticmap?center=55.780001,9.717275&zoom=18&size=400x400&maptype=satellite&key=wefhuwvjwekrlbvowilerbvkebvlearufhbebwe&signature=aqwhfunojlksdcnipwebfpwebfu=

Figure 11. Example of a signed URL and satellite image retrieved using the Google Maps Static API.

4.2 Loss Function Design

One of the most critical parts of designing a deep learning model, is the creation of the loss function that will guide the training. The loss function is a measure of how well the model is

performing, and it is used to adjust the model's parameters during training. Therefore, the choice of loss function is crucial to the success of the model. In this section, I will discuss the design of the loss functions used to train the models I've set out to create. It will consist of a combination of different loss functions, each designed to capture different aspects of the problem at hand. Firstly, I will cover the development of the novel "cold map"-based part of the loss function, which is supposed to guide the model by penalizing points further away from the true path subject to some threshold. Secondly, I will discuss the use of a commonly used loss function, the BCE loss, which is used for binary segmentation tasks. Finally, the last part of the loss function will deal with the actual topology of the predicted path. It will heavily penalize breaks in the path, branches in paths, and other topological errors, such as not reaching connecting the entry and exit. With these three different loss functions, the goal is to compare different combinations of them to determine which one works best for the task at hand.

4.2.1 Cold Map Loss

The first part of the loss function is the cold map loss. This involves using the predicted path generated by the model, and comparing it to the cold map from the dataset. The creation of the cold maps are detailed in Section 4.3.1. Briefly, the cold maps are grids of the same size as the input image, where the intensity of each cell is a value derived from the distance to the nearest path pixel magnified beyond some threshold.

The main idea behind the cold map loss is to introduce spatial penalty that increases as the distance from the true path increases. Though it is similar to BCE, it differs in some key aspects. It is not pixel-wise, but rather a global loss that is calculated over the entire image. This means that slight deviations from the true path are penalized less than those with a larger discrepancy. This property of the loss function is a desirable trait for path-planning tasks, as minor offsets from the true path are less critical than larger ones. This loss function is defined as follows:

$$\mathcal{L} = \sum_{i=1}^{H} \sum_{j=1}^{W} C_{ij} P_{ij} \tag{20}$$

where C_{ij} is the cold map value at pixel (i, j), and P_{ij} is the predicted path value at pixel (i, j). This version of the loss function is a simple dot product between the cold map and the predicted path. Thus, after flattening the cold map and the predicted path matrices, the loss is calculated as the dot product between the two vectors:

$$\mathcal{L}_{\text{cold}} = C \cdot P \tag{21}$$

where C is the cold map vector and P is the predicted path vector, giving a scalar value, contributing to the total loss. This value does, however, grow extremely quickly, as the dot product, as shown by (20), is simply a sum over the entire image. This means that the loss value will be very high, even for small deviations from the true path, and extremely high for large deviations or noisy images, as expected from the model during early stages of

training. While this rapid growth can be combated by introducing a very low weight to the loss function, doing so would also mean that smaller deviations become irrelevant, which in undesired. Thus, inspired by BCE, I will introduce a mean reduction to the loss function, which will divide the loss by the number of pixels in the image. This will ensure that the loss value is more stable and that the model can learn from smaller deviations. With this, the implementation of a cold map loss, will be based on the following equation:

$$\mathcal{L}_{\text{cold}} = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} C_{ij} P_{ij} \tag{22}$$

where H and W are the height and width of the image, respectively. The implementation of (22) is very straightforward, and is shown in the code listing below:

```python
def cmap_loss(cmap_gt, path_pred, reduction) -> torch.Tensor:
  cmap_f = torch.flatten(cmap_gt)
  path_f = torch.flatten(path_pred)

  loss = torch.dot(cmap_f, path_f)

  return loss if reduction != 'mean' else loss / len(cmap_f)

class CmapLoss(nn.Module):
  def __init__(self, weight: float = 1.0, reduction: str = 'mean'):
    super(CmapLoss, self).__init__()
    self.weight = weight

  def forward(self, cmap_gt: torch.Tensor, path_pred: torch.Tensor):
    loss = cmap_loss_torch(cmap_gt, path_pred, self.reduction)
    return self.weight * loss
```

Listing 4. Implementation of the cold map loss calculation using PyTorch.

To implement this loss function, a class `CmapLoss` is created, which inherits from `torch.nn.Module`. The class has a single parameter, `weight`, which is used to scale the loss value. The `forward` method takes the ground truth cold map `cmap_gt` and the predicted path `path_pred` as inputs. The method then calculates the loss using the `cmap_loss` function and scales it by the `weight` parameter. The function `cmap_loss` takes two PyTorch tensors as inputs: `cmap_gt`, which represents the ground truth cold map, and `path_pred`, which is the predicted path output from the model. The cold map and predicted path are initially matrices with dimensions corresponding to the image's height and width. To compute the loss as a single scalar, both matrices are flattened into one-dimensional vectors. The function then calculates the dot product between these two vectors using `torch.dot`, effectively summing the element-wise products. If the `reduction` parameter is set to `mean`, the loss is divided by the number of elements in the vectors, which is the total number of pixels in the image.

Examples of this loss function in action is shown in Figure 12. The left and center top row plots overlays a complete, single width path on top of the cold map. This highlights the fact that paths that are close to the true path, are penalized less and the further away, the more the penalty explodes in value. The last image in the top row shows how the loss handles a noisy image. As it can be seen, the loss is significantly higher than the rest. This is a desired trait of the loss function, as noise is just about the exact opposite of a continuous path. The bottom row shows alternate paths. The path on the rightmost image is the true

path, which shows that if the path is dead on, then the penalty is none. The leftmost image, shows a path that is not connected. As shown, it still scores a perfect score. This is because the cold map loss only penalizes the distance from the path, not the topology of the path itself, so breaks will only result in a higher score. Lastly, the center image shows the true path, but with several branches. As seen in the loss value, this also incurs very little penalty. The bottom row of images highlight a dire need for a topology-based loss, which will be explored in Section 4.2.3.

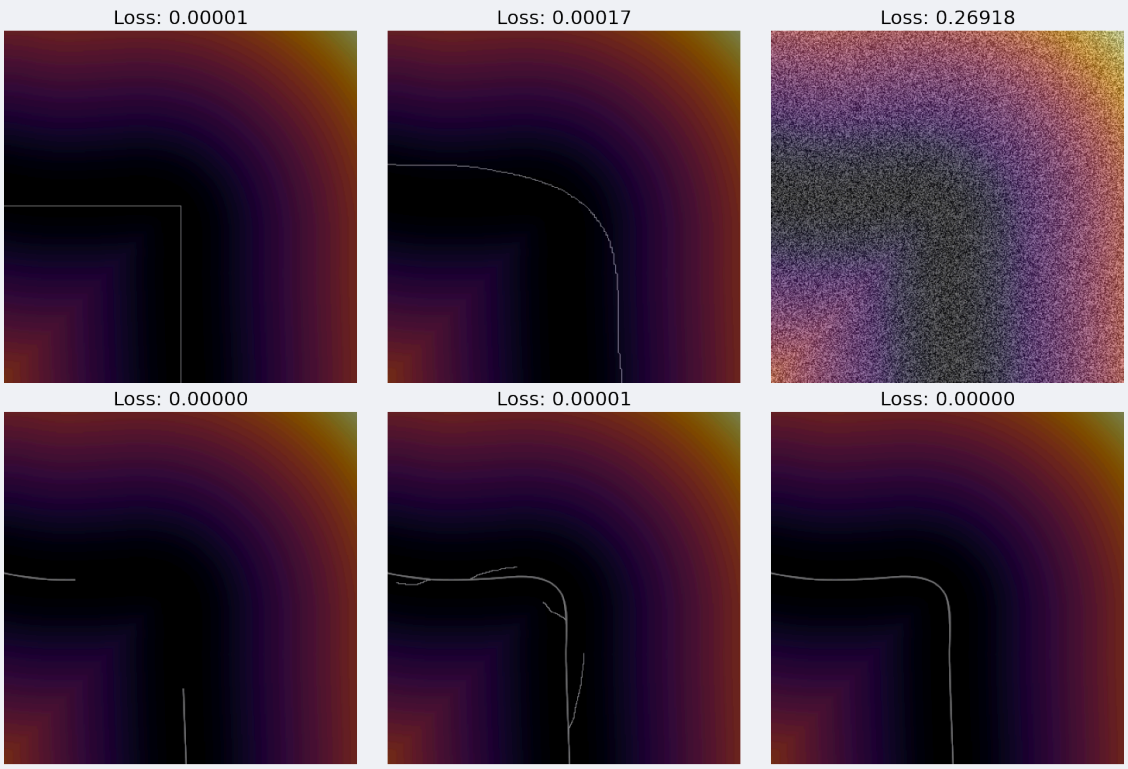


Figure 12. Paths drawn on top of a cold map with their associated loss above calculated using the function from Listing 4. The top row shows fully connected paths, while the bottom row shows paths with breaks and branches, as well as the true path.

4.2.2 Binary Cross-Entropy Loss

The BCE loss is a commonly used loss function for binary segmentation tasks, which is relevant for the task at hand due to the pixel-subset nature of the problem, i.e. pixels can be either 1 or 0. Furthermore, it is well-versed in handling heavily imbalanced data, which is the case when dealing with classification tasks where one class is much more prevalent than the other, like path and non-path pixels in an image. These properties make it ideal for this problem, as the background pixels are much more prevalent than the path pixels. The implementation is using the definition from PyTorch[2], which is defined as follows:

[2]Full implementation details can be found in the official documentation: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

$$\ell(x, y) = L = \{l_1, ..., l_N\}^T \tag{23}$$

with

$$l_n = -w_n[y_n \log x_n + (1 - y_n) \log(1 - x_n)] \tag{24}$$

where w_n is a weight parameter, y_n is the ground truth label, x_n is the predicted label, and ℓ is subject to

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'} \\ \text{sum}(L), & \text{if reduction} = \text{'sum'} \end{cases} \tag{25}$$

depending on the reduction parameter. The left-hand side of (24) is activated when the ground truth label is 1. It evaluates how well the positive class's predicted probability x_n aligns with the ground truth. A smaller loss is achieved when the predicted probability is close to 1. The right-hand side of (24) is activated when the ground truth label is 0. It evaluates how well the negative class's predicted probability x_n aligns with the ground truth. A smaller loss is achieved when the predicted probability is close to 0. The BCE loss is then calculated as the sum or mean of the individual losses, depending on the `reduction` parameter. The weight parameter w_n can be used to scale the output of the loss function. BCE quantifies the dissimilarity between the predicted probabilities and the actual labels, giving a sense of how well the model is performing. For example, calculating the BCE with $y = 1$ and $x = 0.8$ gives

$$-1 \cdot (1 \cdot \log(0.8) + (1 - 1) \cdot \log(1 - 0.8)) = -\log(0.8) = 0.223$$

This value represents the dissimilarity between the predicted probability of 0.8 and the actual label of 1. A lower value indicates that the model's prediction is closer to the ground truth, suggesting a more accurate classification. Alternatively, the value can be near 1, indicating a large discrepancy between the predicted and actual labels. So, a value of 0.223 is a good result, as it indicates that the model is performing well, with some room for improvement. For contrast, if the predicted label is 0.4, but the true label is 1, the dissimilarity would be $-\log(0.4) \approx 0.916$. This higher value reflects a more significant error in prediction. From this, note that the function calculates the dissimilarity for both positive and negative classes. So, in the case of the predicted label being 0.4 and the true label being 0, the loss value would be much better at just $-\log(1 - 0.4) \approx 0.51$.

In the PyTorch implementation, the BCE loss can be either summed or averaged over the batch, depending on the `reduction` parameter. While using the sum method can provide stronger signals on rare but critical pixels, averaging might help maintain stability across batches, especially when dealing with very imbalanced datasets. Thus, the mean method is chosen for this project, as it leads to a more stable training process with smaller fluctuations in the loss values. The BCE loss is calculated using the following code snippet:

```python
def bce_loss_torch(path_gt, path_pred, reduction) -> torch.Tensor:
    bce = torch.nn.BCELoss(reduction=reduction)

    bce_loss = bce(path_pred, path_gt)

    return bce_loss

class BCELoss(nn.Module):
  def __init__(self, weight: float = 1.0):
    super(BCELoss, self).__init__()
    self.weight = weight

  def forward(self, path_gt: torch.Tensor, path_pred: torch.Tensor, reduction: str = 'mean'):
    loss = bce_loss_torch(path_gt, path_pred, reduction)
    return self.weight * loss
```

Listing 5. Implementation of the BCE loss function using PyTorch.

To implement the BCE loss function, a smaller wrapper class for the existing PyTorch implementation is created. The class `BCELoss` inherits from `torch.nn.Module` and has a single parameter, `weight`, which is used to scale the loss value. The `forward` method takes the ground truth path `path_gt`, the predicted path `path_pred`, and the reduction method as inputs. The method then calculates the loss using the `bce_loss_torch` function and scales it by the `weight` parameter. The function `bce_loss` calculates the BCE loss between the ground truth and predicted paths. It takes the paths to the ground truth and predicted images as input, reads them using OpenCV, and converts them to PyTorch tensors after normalization, since the images are stored as 8-bit greyscale images. The function creates a BCE loss `criterion` using `torch.nn.BCELoss` and calculates the loss using the ground truth and predicted tensors. The loss value is then returned as a float. In the tensor version of the function, the paths are already tensors, and the function can be called directly with the tensors as input.

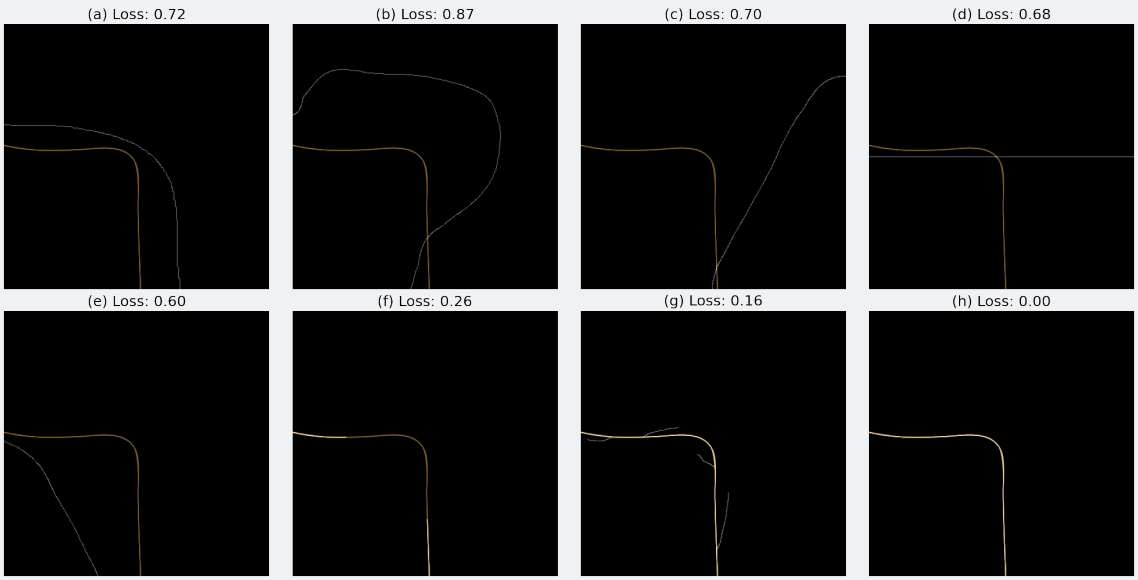


Figure 13. The ground truth ● compared to some drawn path ●. The losses above the plots are the BCE loss using function in Listing 5 using `mean` as the string passed as the reduction method.

Examples of the `mean`-based BCE loss is shown in Figure 13, showing various interesting aspects of the loss function. (b) shows an expectedly high loss value, as the path is far from

the true path. This matches the case for the cold map loss. (a), (c), and (d) show very similar loss values, despite being vastly different, both in terms of closeness to the path, but also where they are going to and from, further highlighting the need for a topology-based loss. Even their sum counterparts show very similar values. Interestingly, the losses seen in (f) and (h) are very different, when the cold map based loss showed them as being equal. This shows that BCE is more sensitive to the topology of the path, but as (g) shows, it still gives a very low loss value when a path has branches. All of this shows that some topology analysis is needed for the loss function to capture realistic paths.

Other considerations for handling imbalanced data include methods like Dice [54] similarity coefficient and Focal loss [55], which can also be effective in certain contexts. The Dice similarity coefficient is a measure of overlap between two samples, and is particularly useful when dealing with imbalanced data. The Focal loss is designed to address the class imbalance problem by focusing on hard examples that are misclassified. These methods can be used in conjunction with the BCE loss to improve the model's performance, especially when dealing with heavily imbalanced datasets. But for the purposes of this project, the BCE loss is expected to be sufficient to handle the class imbalance.

4.2.3 Topology Loss

As the previous sections have highlighted, there is a dire need for a topology-based loss function. The cold map loss and the BCE loss are both excellent at penalizing paths that are far from the true path, but they do not penalize breaks in the path, branches in the path, or paths are only loosely driven towards being connected. This is where the topology loss comes in. The constituent parts of the topology loss are detailed in the following sections. First, the continuity part of the topology loss is discussed, which is designed to ensure that the predicted path is continuous and does not contain any breaks. This is done by aiming for specific Betti number values. Second, eliminating branches in the path is discussed, which is crucial for ensuring that the predicted path is a single connected component. Finally, the entry and exit part of the topology loss is discussed, which is designed to ensure that the predicted path connects the entry and exit points of the intersection.

Considerations of using existing topology existing methods. Dep *et al.* [56] introduced TopoNets and TopoLoss. This loss function revolves around penalizing jagged paths and encouraging smooth, brain-like topographic organization within neural networks by reshaping weight matrices into two-dimensional cortical sheets and maximizing the cosine similarity between these sheets and their blurred versions. Cortical sheets are two-dimensional grids formed by reshaping neural network weight matrices to emulate the brain's spatial organization of neurons, enabling topographic processing. While initially interesting in the context of this project, simple testing showed that the values returned from this loss, did not give a proper presentation of the path's topology, outside of its smoothness. And while smoothness is a part of the topology, this is will largely be handled by the BCE loss.

4.2.3.1 Continuity

The first part of the topology loss is the continuity part. This part of the loss function is crucial for ensuring that the predicted path is continuous and does not contain any breaks. Breaks in a path would be unrealistic for a grounded vehicle to follow. To understand the continuity part of the topology loss, it is essential to understand the concept of Betti numbers as described below:

Betti Numbers

Betti numbers [57] come from algebraic topology, and are used to distinguish topological spaces based on the connectivity of n-dimensional simplicial complexes. The nth Betti number, β_n, counts the number of n-dimensional holes in a topological space. The Betti numbers for the first three dimensions are:

- β_0: The number of connected components.
- β_1: The number of loops.
- β_2: The number of voids.

The logic follows that, in 1D, counting loops are not possible, as it is simply a line. This, if the number is greater than 1, it means it is split into more than component. In 2D, the number of loops is counted, i.e. a circled number of pixels. In 3D, this extends to voids.

With this, for the 2D images used in this project, the Betti numbers are β_0 and β_1. The continuity part of the topology loss is designed to ensure that the predicted path has a single connected component and no loops. This is achieved by aiming for the Betti numbers to be $\beta_0 = 1$ and $\beta_1 = 0$. Higher dimensional Betti numbers are not relevant for this project, as the images are 2D. While Betti numbers are a powerful tool for topology analysis, they are not directly applicable to the loss function as they are discrete values. This means that offer no gradient information, which is essential for training a neural network. Instead, persistent homology is deployed.

Persistent homology is a mathematical tool used to study topological features of data. Homology itself is a branch of algebraic topology concerned with procedures to compute the topological features of objects. Persistent homology extends the basic idea of homology by considering not just a single snapshot of a topological space but a whole family of spaces built at different scales. Instead of calculating Betti numbers for one fixed space a filtration is performed. This filtration is a sequence of spaces, where each space is a subset of the next, i.e. a nested sequence of spaces where each one is built by gradually growing the features by some threshold. As this threshold varies, topological features such as connected components and loops will appear (be born) and eventually merge or vanish (die).

This birth and death of features is recorded in what is known as a persistence diagram or barcode (See Figure 14). In these diagrams, each feature is represented by a bar (or a point in the diagram) whose length indicates how persistent, or significant, the feature is across different scales. Features with longer lifespans are generally considered to be more robust and representative of the underlying structure of the data, whereas those that quickly appear and disappear might be attributed to noise.

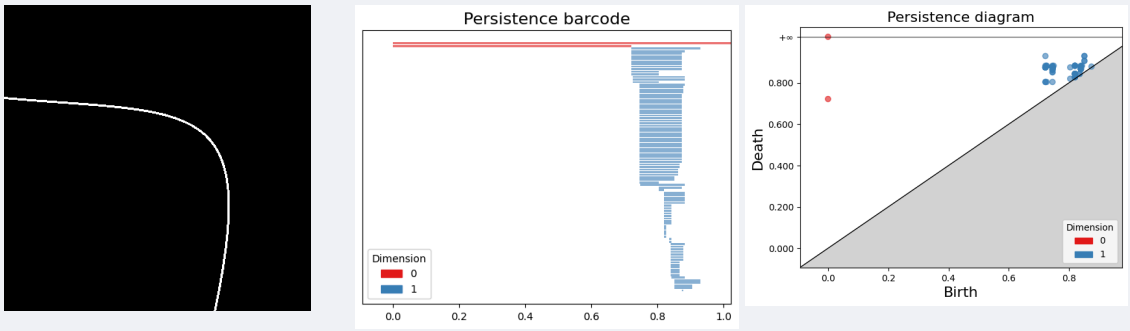


Figure 14. The top row shows a connected path along with its persistence barcode and persistence diagram, while the bottom row shows a disconnected path. The number of lines in the barcode, stems from the fact that the images are rather large in size and thus the number of built spaces are many.

Since sources are very scarce regarding the implementation of the differentiable wasserstein distance, an approximation is made for the backwards pass as shown in Listing 6. The entire implementation is shown in the code listing below:

```python
class PersistentHomologyLossFunction(torch.autograd.Function):
  def forward(ctx, input_tensor, target_betti, threshold):
    array = input_tensor.detach().cpu().numpy()
    array_inv = 1.0 - array
    cc = gd.CubicalComplex(top_dimensional_cells=array_inv)

    computed_betti = {}
    for dim, (birth, death) in cc.persistence():
      if death == float('inf') or death > threshold:
        computed_betti.setdefault(dim, 0)
        computed_betti[dim] += 1

    ctx.computed_betti = computed_betti
    ctx.target_betti = target_betti
    ctx.input_shape = input_tensor.shape

    loss_sq = 0.0
    for dim, target in target_betti.items():
      comp = computed_betti.get(dim, 0)
      diff = comp - target
      loss_sq += diff * diff
    loss_value = np.sqrt(loss_sq)

    ctx.loss_value = loss_value
       ctx.diff_dict = {dim: computed_betti.get(dim, 0) - target for dim, target in
target_betti.items()}

    return torch.tensor(loss_value, dtype=input_tensor.dtype, device=input_tensor.device)

  def backward(ctx, grad_output):
    total_diff = 0.0
    for diff in ctx.diff_dict.values():
      total_diff += diff

    L_val = ctx.loss_value if ctx.loss_value > 1e-8 else 1e-8
    grad_scalar = total_diff / L_val
    grad_input = grad_output * grad_scalar
                 * torch.ones(ctx.input_shape, device=grad_output.device)

    return grad_input, None, None

class PersistentHomologyLoss(nn.Module):
  def __init__(self, target_betti, threshold=0.5):
    super(PersistentHomologyLoss, self).__init__()
    self.target_betti = target_betti
    self.threshold = threshold

  def forward(self, input_tensor):
    return PersistentHomologyLossFunction.apply(input_tensor,
                                                self.target_betti,
                                                self.threshold)
```

Listing 6. Persistent Homology implementation.

The loss function consists of two classes, `PersistentHomologyLoss` is the wrapper class that inherits from `torch.nn.Module`. It takes two parameters, `target_betti` and `threshold`. The `target_betti` parameter is a dictionary containing the target Betti numbers for the input tensor, while the `threshold` parameter is the threshold value used in the persistent homology calculation. The `forward` method of the `PersistentHomologyLoss` class calls the `PersistentHomologyLossFunction` class, which is a custom autograd function that performs the persistent homology calculation. The `PersistentHomologyLossFunction` class inherits from `torch.autograd.Function` and has two methods, `forward` and `backward`.

The `forward` method takes the input tensor, target Betti numbers, and threshold as inputs. First, the tensor is converted to a numpy array so it can be used by the Gudhi function

creating the cubical complex. The cubical complex is created using the `CubicalComplex` class, which constructs a cubical complex from the input tensor. The persistence of the cubical complex is then computed using the `persistence` method, which returns the birth and death values of the topological features. The computed Betti numbers for each dimension are then calculated and stored in the `computed_betti` dictionary. The loss value is calculated by comparing the computed Betti numbers to the target Betti numbers and summing the squared differences. The square root of the sum is then returned as the loss value:

$$\mathcal{L}_{\text{cont}} = \sqrt{\sum_{\text{dim}} \left(\beta^{\text{dim}}_{\text{computed}} - \beta^{\text{dim}}_{\text{target}}\right)^2} \tag{26}$$

The `backward` method calculates the gradient of the loss value with respect to the input tensor. The total difference between the computed and target Betti numbers is calculated, and the gradient is computed as the total difference divided by the loss value. The gradient is then multiplied by the output gradient and returned as the gradient.

Formally, there is no gradient to discrete values, so a surrogate gradient is required. First, the aggregate discrepancy is defined as

$$\Delta = \sum_{\text{dim}} \left(\beta^{\text{dim}}_{\text{computed}} - \beta^{\text{dim}}_{\text{target}}\right) \tag{27}$$

The `backward` method approximates the derivative of the loss with respect to the input by simply computing a scalar gradient:

$$g = \frac{\Delta}{L} \tag{28}$$

where $L = \max(\varepsilon, \mathcal{L}_{\text{cont}})$ with $\varepsilon = 10^{-8}$. This scalar g is then uniformly distributed over all elements of the input tensor. Finally, the gradient with respect to the input is given by

$$\nabla_x L = g \cdot \text{grad\_output} \tag{29}$$

where grad_output is the upstream gradient.

Creating and using the `PersistentHomologyLoss` class is straightforward. The target Betti numbers are defined as a dictionary with the desired Betti numbers for each dimension. The threshold parameter is also defined in a variable and passed during initialization:

```
target_betti = {0: 1, 1: 0}
topo_loss_fn = PersistentHomologyLoss(target_betti, threshold=0.75)
loss = topo_loss_fn(p_tensor)
```

Listing 7. Creating and using the `PersistentHomologyLoss` class.

Examples of the persistent homology loss is shown in Figure 15. Remember, the desired Betti numbers are $\beta_0 = 1$ and $\beta_1 = 0$, meaning that any more than 1 component and any loops should be penalized. As expected, the ground truth in (a) has a loss value of 0. (b) showcases the fact that if parts of the true path are missing, the loss value will swiftly increase. (c) shows this even better, as it consists of 4 different component, thus achieving

a difference of 3. (d) shows that the loss function also penalizes loops, as the loss value is 1.

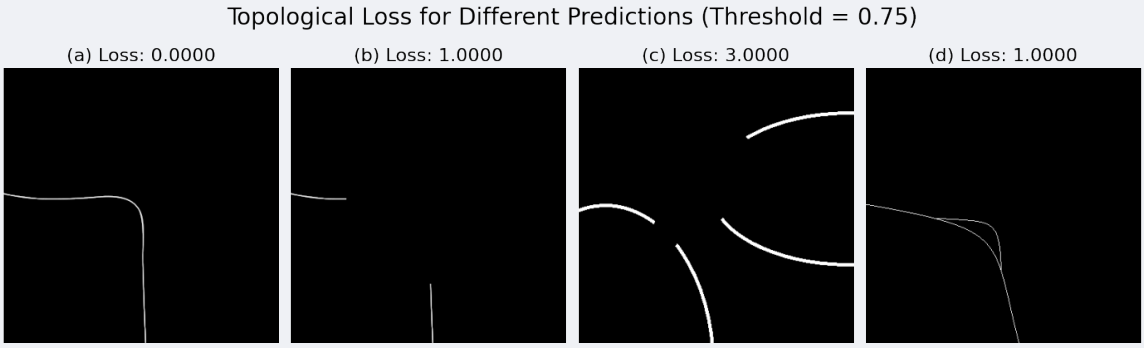


Figure 15. Results of using the continuity part of the topology loss.

4.2.3.2 Branching

The second part of the topology loss is the branching part. Branching refers to the presence of dead-ends in the predicted path. Dead-ends are points in the path that stop abruptly. These are undesired as a vehicle will only need one path to follow through an intersection and dead-end branches might stop in the middle of the intersection or somewhere completely irrelevant. This section of the loss function is another crucial part that goes along nicely with the continuity part because, while the continuity parts ensures that only one component is present, a branching path will still only total one component. So, the continuity part handles component count and loops, while the branching part handles undesired dead-ends. The only desired dead-ends are the entry and exit points of the intersection. How these are handled is discussed in Section 4.2.3.3.

This part of the loss function will focus on counting the number of endpoints of the predicted path. The goal is to penalize paths that have more than two endpoints, as this indicates that the path has branches. This will be achieved by using an altered kind of the 8-neighbour grid. In this grid, the value at each pixel is decided by how many neighbours is has as defined by some kernel. In the 8-neighbour grid, the kernel is defined as

$$k = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{30}$$

resulting in each grid point having a value stored equal to that of the number of occupied pixels surrounding it, naturally ignoring itself, after convolution. In the context of detecting branches, the desired value for all pixels is 2, other than the entry and exit points, which are allowed to have 1 as the only points. A value of 2 is allowed since this means that the pixel is on the path, i.e. each pixel has their own entry and exit neighbour points. This definition does, however, become troublesome when images are not binary, i.e. the path may be subject to some kind of smoothing or anti-aliasing. For example, if the pixels near the exit point have some value, they would be counted as neighbours, meaning that the exit

point would not be considered an endpoint. Therefore, in the implementation in Listing 8, a soft threshold is employed instead of a hard one.

Also to be considered for this, is the fact that it should be differentiable. Thankfully, a soft threshold is differentiable, as it is simply a sigmoid function. The input tensor is put through the following equation:

$$\text{input\_tensor} = \sigma\left(\alpha\left(\frac{\text{input\_tensor}}{255} - t\right)\right) \tag{31}$$

where t is the threshold, α is the steepness of the sigmoid, and σ is the sigmoid function. This approach gives way for a differentiable approximation to a binary threshold, and a high value for α will make the threshold very sharp, while a low value will make it very soft. Tests of various values for α and t were conducted early and can be found in Appendix A. From this, the values for chosen to be $\alpha = 70.0$ and $t = 0.85$. This is also reflected in the code. These high values were chosen to ensure that the threshold is very sharp, and points that occur as a result of anti-aliasing are not counted as pixels in the actual end-points count.

Next, the neighbour sum is calculated using the kernel defined in (30). This is done by convolving the input tensor with the kernel using the `conv2d` function from PyTorch, resulting in a tensor where each pixel contains the number of occupied pixels in its 8-neighbourhood:

$$\text{neighbour\_sum}_{x,y} = \sum_{i=-a}^{a} \sum_{j=-b}^{b} k_{i,j} f_{x-i,y-j} \tag{32}$$

where k is the kernel, f is the input tensor, and a and b describe the size of the kernel by $-a \leq i \leq a$ and $-b \leq j \leq b$. In this case, $a = b = 1$ as the kernel is 3x3. `padding = 1` ensures equal stride.

However, instead of directly counting the number of endpoints, the number of endpoints is calculated using a Gaussian function. This is done to ensure that the loss function is differentiable. The indicator function is defined as

$$\text{indicator} = \exp\left(-\frac{(\text{neighbour\_sum} - 1)^2}{2\sigma^2}\right) \tag{33}$$

where σ is some defined value, here set to $\sigma = 0.1$ as it then sharply peaks around 1. This further works as a smooth approximation compared to a strong threshold of 1. Finally, the loss calculated by finding the number of endpoints with

$$\text{endpoints} = \text{input\_tensor} \cdot \text{indicator} \tag{34}$$

and squaring the difference between the number of endpoints and the target number of endpoints:

$$\mathcal{L}_{\text{branch}} = (E - T)^2 \tag{35}$$

where $E = \sum \text{endpoints}$ and T is the target number of endpoints. The loss is then returned as a tensor.

```python
class EndpointsLossFunction(torch.autograd.Function):
  def forward(ctx, input_tensor, target_ee, alpha = 70.0, threshold = 0.85):
    input_tensor = input_tensor / 255.0
    input_tensor = torch.sigmoid(alpha * ((input_tensor) - threshold))

    p = input_tensor.unsqueeze(0).unsqueeze(0)

    kernel = torch.tensor([[1, 1, 1],
                           [1, 0, 1],
                           [1, 1, 1]],
                          dtype=torch.float32).unsqueeze(0).unsqueeze(0)

    neighbour_sum = F.conv2d(p, kernel, padding=1).squeeze(0).squeeze(0)

    sigma = 0.1
    desired_endpoints = 1
    indicator = torch.exp(-((neighbour_sum - desired_endpoints) ** 2) / (2 * sigma ** 2))

    endpoints_map = input_tensor * indicator
    measured_endpoints = torch.sum(endpoints_map)

    loss_value = (measured_endpoints - target_ee) ** 2

    ... save values to ctx ...

    return loss_value

  def backward(ctx, grad_output):

    ... load ctx values ...

    E = (input_tensor * indicator).sum()

    dL_dE = 2 * (E - target_ee)

    grad_direct = indicator

    factor = -(neighbour_sum - 1) / (sigma ** 2)

    conv_input = (input_tensor * indicator * factor).unsqueeze(0).unsqueeze(0)
    grad_indirect = F.conv2d(conv_input, kernel, padding=1).squeeze(0).squeeze(0)

    grad_total = grad_direct + grad_indirect

    soft_thresholded = torch.sigmoid(alpha * (input_tensor - threshold))
    sigmoid_deriv = (alpha / 255.0) * soft_thresholded * (1 - soft_thresholded)

    grad_input = grad_output * dL_dE * grad_total * sigmoid_deriv

    return grad_input, None

class EndpointsLoss(nn.Module):
  def __init__(self, target_ee, alpha = 70.0, threshold = 0.8):
    super(EndpointsLoss, self).__init__()
    self.target_ee = target_ee
    self.alpha = alpha
    self.threshold = threshold

  def forward(self, input_tensor):
    return EndpointsLossFunction.apply(input_tensor, self.target_ee,
                                       self.alpha, self.threshold)
```

Listing 8. Branching loss implementation.

To explain the backpropagation of this part of the loss function, I will re-introduce and define the following quantities:

Soft-threshold

$$p = \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right) \tag{36}$$

where
- p is the soft-thresholded pixel value,
- x is the input,
- t is the threshold,
- σ is the sigmoid function,
- α is the steepness of the sigmoid.

Indicator

$$I = \exp\left(-\frac{(S-1)^2}{2\sigma^2}\right) \tag{37}$$

where
- I is the indicator function,
- S is the neighbour sum obtained by convolution with k from (30),
- σ is a scalar value.

Pixel endpoint contribution

$$f(p) = p \cdot I \tag{38}$$

where
- p comes from (36),
- I is the indicator from (37).

Loss function definition

$$\mathcal{L}_{\text{branch}} = (E - T)^2 \tag{39}$$

where
- E is the number of endpoints,
- T is the target number of endpoints.

Several calculation contribute to the total derivative. Firstly, the derivative of the loss in (39) is simply

$$\frac{\partial \mathcal{L}_{\text{branch}}}{\partial E} = 2(E - T) \tag{40}$$

Secondly, the direct contribution from the indicator function is found by taking the derivative of (37) with respect to the pixel value p:

$$\frac{\partial}{\partial p} p \cdot I = I \tag{41}$$

directly representing the sensitivity of (38). Thirdly, since the indicator I is a function of the neighbour sum S, which in turn depends on p via convolution, there is an indirect contribution as well. Given (37), its derivative with respect to S is

$$\frac{\partial I}{\partial S} = -\frac{S-1}{\sigma^2} \cdot I \tag{42}$$

To then find the contributions from neighbouring pixels, the product $p \cdot$ (42) is convolved with the kernel k from (30). Lastly, the derivative of (36) is found by taking the derivative of p with respect to x. First, the derivative of the sigmoid function is found:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{43}$$

So the derivative of p with respect to input x is

$$\frac{\partial}{\partial x} p = \frac{\alpha}{255} \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right)\left(1 - \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right)\right) \tag{44}$$

Now, to find the total gradient, the chain rule will be applied to find the gradient of the loss with respect to the input x:

$$\frac{\partial}{\partial x}\mathcal{L}_{\text{branch}} = \frac{\partial \mathcal{L}_{\text{branch}}}{\partial E} \cdot \frac{\partial E}{\partial p} \cdot \frac{\partial p}{\partial x} \tag{45}$$

- $\frac{\partial \mathcal{L}_{\text{branch}}}{\partial E}$ was found in (40),
- $\frac{\partial E}{\partial p}$ is the direct and indirect contributions from the indicator function, and
- $\frac{\partial p}{\partial x}$ is (44) rewritten as $\frac{\alpha}{255}p(1-p)$

yielding the complete gradient

$$\frac{\partial}{\partial x}\mathcal{L}_{\text{branch}} = 2(E-T) \cdot \left[I + \text{Conv}\left(p \cdot \left(-\frac{S-1}{\sigma^2}\right)\right)\right] \cdot \frac{\alpha}{255}p(1-p) \tag{46}$$

as seen in Listing 8 line 48. Some results can be seen in Figure 16.

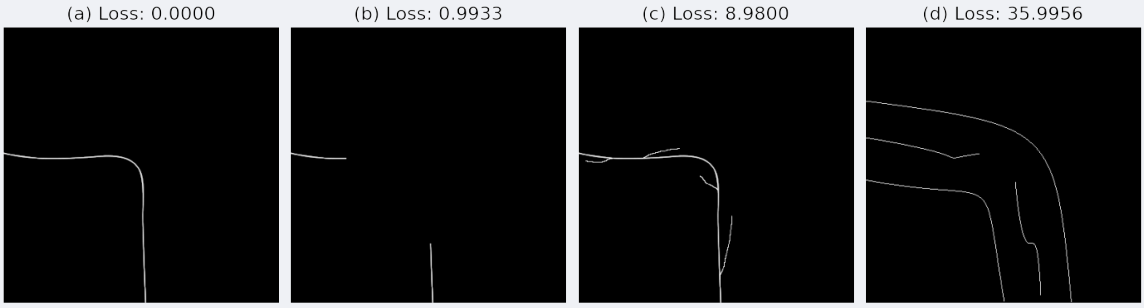


Figure 16. Results of using the branching part of the topology loss.

4.2.3.3 Entry/Exit

4.3 Dataset Creation

The creation of a proper dataset is crucial for making sure the model learns the task desired for it to perform. The dataset will have to work hand-in-hand with the model architecture and the loss function to ensure that the model learns the task effectively. Many aspects are to be considered when creating a dataset for a task as specific as this project sets out to create:

- It should be large enough to capture the complexity of the task. Size can be artificially increased through data augmentation.
- It should be diverse enough to capture the variety of scenarios that can occur at an intersection.
- It should allow for some leniency when it comes to generating paths, as the model should not be too stringent to a singular path.
- For the purposes of this project, its creation should seek to answer Research Question **RQ-3** by providing a dataset that allows for the training of a model that can generate paths that are not too stringent to a singular path.

4.3.1 Cold maps

The deduced method for training the model, as detailed in Section 4.2, includes the use of a cold map. A cold map representation of the desired path was chosen for a small simplification in the loss function. It penalizes points that are further from the desired path, and does not do this for points that are on the path. Creating this cold map was done in several steps. First, a grid of the same size as the input image is created. The input image is the path drawn in white on a black background, as shown in centre Figure 19. This means that the only occupied pixels are those taken up by the path. In this grid, the coordinates of the closest non-zero pixel is found by iterating over the entire input image containing the path. The complexity of this operation will be covered in the following sections. Next, the distance between the current pixel and the closest non-zero pixel is calculated. This distance is then compared to a threshold value to determine its value. If it is further away, the resulting penalty from the loss function should be higher. Different values for the threshold and the exponent of the distance calculation were tested to find the best combination. Lastly, the cold map is saved in a structured folder format for later use in training. Later, the created data is put through augmentation to inflate the size of the dataset and increase its diversity.

4.3.1.1 Finding the distance to the desired path

The algorithm for finding the distance to the closest point on the desired path is shown in Listing 9.

```
occupied = []
for i in range(binary.shape[0]):
  for j in range(binary.shape[1]):
    if binary[i, j] != 0:
      occupied.append((i, j))

h, w = binary.shape
nearest_coords = np.zeros((h, w, 2), dtype=int)

for i in range(binary.shape[0]):
  for j in range(binary.shape[1]):
    if binary[i, j] == 0:
      min_dist = float('inf')
      nearest_coord = (i, j)
      for x, y in occupied:
        d = hypot(i - x, j - y)
        if d < min_dist:
          min_dist = d
          nearest_coord = (x, y)
      nearest_coords[i, j] = nearest_coord
    else:
      nearest_coords[i, j] = (i, j)
```

Listing 9. Non-parallelized code for finding the nearest point on the path.

The algorithm in Listing 9 starts by creating an array of coordinates based on the `binary` map created with the `threshold` function from the OpenCV library. This `binary` map contains every non-black pixel in the input image, which in this case is the path drawn on a black background. With these occupied pixels stored in an array, the algorithm then iterates over every grid point of the `nearest_coords` grid, created to be the same size as the input image. For every point in the grid, the algorithm checks if the point is on the path.

If it is, the algorithm assigns the current point's coordinates to the `nearest_coords` grid. If the point is not on the path, the algorithm iterates over every occupied pixel and calculates the distance between the current point and the occupied pixel. If the distance is less than the current minimum distance, the minimum distance is updated and the coordinates of the closest point are saved. This is repeated for every occupied pixel, and the coordinates of the closest point are saved in the `nearest_coords` grid. This process is repeated for every point in the grid until every point has been assigned the coordinates of the closest point on the path. This grid will later be used under the name `coords`.

The shown algorithm is not parallelized and has a complexity of $\mathcal{O}(n^2)$, where n is the size of the input image. This is due to the nested `for`-loops used in the algorithm. While not a great complexity, it is a vast improvement over its earlier iteration which was $\mathcal{O}(n^4)$ [3]. The actual implementation of this algorithm is parallelized, but the non-parallelized form is shown here. The first iteration of the algorithm took 73 minutes to complete on a 400×400 image, while the parallelized version took 8 minutes on an 8-core CPU. This non-parallelized version takes roughly 30 seconds to complete on the same image, with the parallelized version taking just a few seconds on a full 400×400 image. Further improvements are likely possible to be made both to the complexity of the implementation and parallelization could be distributed to a GPU or the cloud for even faster computation, but this remains future work.

4.3.1.2 Creating the cold map

To start the creation of the cold map, a distance grid is created using Pythagoras' theorem between the coordinates of the point of the grid and the coordinates saved within, retrieved from the aforementioned `coords` variable. A masking grid is then created by comparing the distance grid to a threshold value. This results in each grid point being calculated using:

$$d_{ij} = \sqrt{\left(i - c_{ij0}\right)^2 + \left(j - c_{ij1}\right)^2} \tag{47}$$

$$dt_{ij} = \begin{cases} d_{ij} & \text{if } d_{ij} < t \\ t + \left(d_{ij} - t\right)^e & \text{otherwise} \end{cases} \tag{48}$$

where $c =$ `coords`, $c_{ij0} =$ `coords[i, j][0]`, t is the threshold value, and e is the exponent value. All three of these can be seen as function parameters in the function declaration in Listing 10. The distance grid is then normalized to a range of 0 to 255 to minimize space usage such that it fits within a byte, i.e. an unsigned 8-bit integer. This is done by subtracting the minimum value and dividing by the range of the values. Alternatively, the `normalize` parameter can be set to another value, as usage within a loss function would prefer a value between 0 and 1 (as detailed in Section 4.2). The resulting grid is then saved as a cold map. The resulting cold map can be seen in the rightmost image in Figure 19.

[3] The original implementation can be seen in `dataset/lib.py:process_rows` in the GitLab repository.

```python
def coords_to_coldmap(coords, threshold: float, exponent: float, normalize: int = 1):
  rows, cols = coords.shape[0], coords.shape[1]

  distances = np.zeros((rows, cols), dtype=np.float32)
  for i in range(rows):
    for j in range(cols):
      distances[i, j] = hypot(i - coords[i, j][0], j - coords[i, j][1])

  distances_c = distances.copy()
  mask = distances > threshold
  distances_c[mask] = threshold + (distances[mask] - threshold) ** exponent

  distances_c_normalized = normalize * (distances_c - distances_c.min()) / (distances_c.max()
- distances_c.min())

  return_type = np.uint8 if normalize == 255 else np.float32

  return distances_c_normalized.astype(return_type)
```

Listing 10. Non-parallelized code for finding the nearest point on the path.

To figure out the optimal values for the threshold and exponent, a grid search was performed. The grid search was done by iterating over a range of values for both the threshold and the exponent. The resulting cold maps were then evaluated by a human to determine which combination of values resulted in the most visually appealing cold map. For a 400×400 image, the optimal values were found to be $t = 20$ and $e = 1.25$. The grid of results can be seen in figure Figure 17. In testing, the value of $e = 1$ was excluded as it had no effect on the gradient produced in the cold map, meaning all values of t produced the same map.

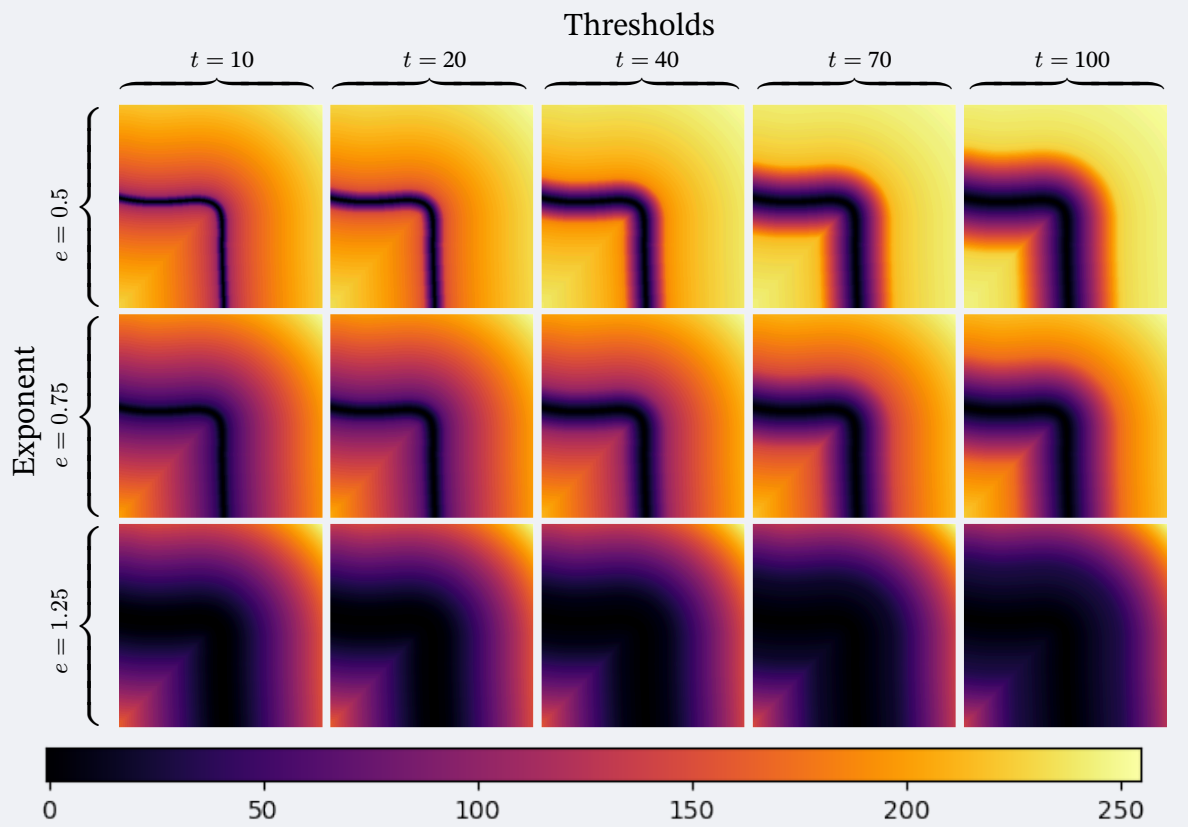


Figure 17. Results of testing threshold values $t \in \{10, 20, 40, 70, 100\}$ and exponent values $e \in \{0.5, 0.75, 1.25\}$. The colour map is shown beneath the results[4].

While pretty, these coldmaps can be difficult to understand. Therefore, Figure 18 shows the 3D plots of the generated cold maps with exponent values $e \in \{0.50, 0.75, 1.25\}$. While the 2D plots with $e < 1$ seem to the eye to be the plots that more greatly penalizes larger distances, the 3D plots reveal that while the points that are further away are more penalized, the gradient is not as steep as the 2D plots would suggest. While heavily penalized, the slope does a poor job of pointing the gradient in the right direction. Then if a point is close, it will experience rapid change that forces it closer to the true path. This is not the desired effect of this loss function, as it should be more lenient to points that are close to the true path. Thus, when $e > 1$, the slope is much more gentle closer to the true path, and steeper further away, which is the desired effect. So, despite the opposite being the intuitive point to take away from glancing at the 2D plots, the 3D plots reveal that the exponent value e should be greater than 1, and thus the value of $e = 1.25$ was chosen for the cold maps and defined as the default for the function in Listing 10.

[4]The colour map is retrieved from the matplotlib docs: https://matplotlib.org/stable/users/explain/colors/colormaps.html

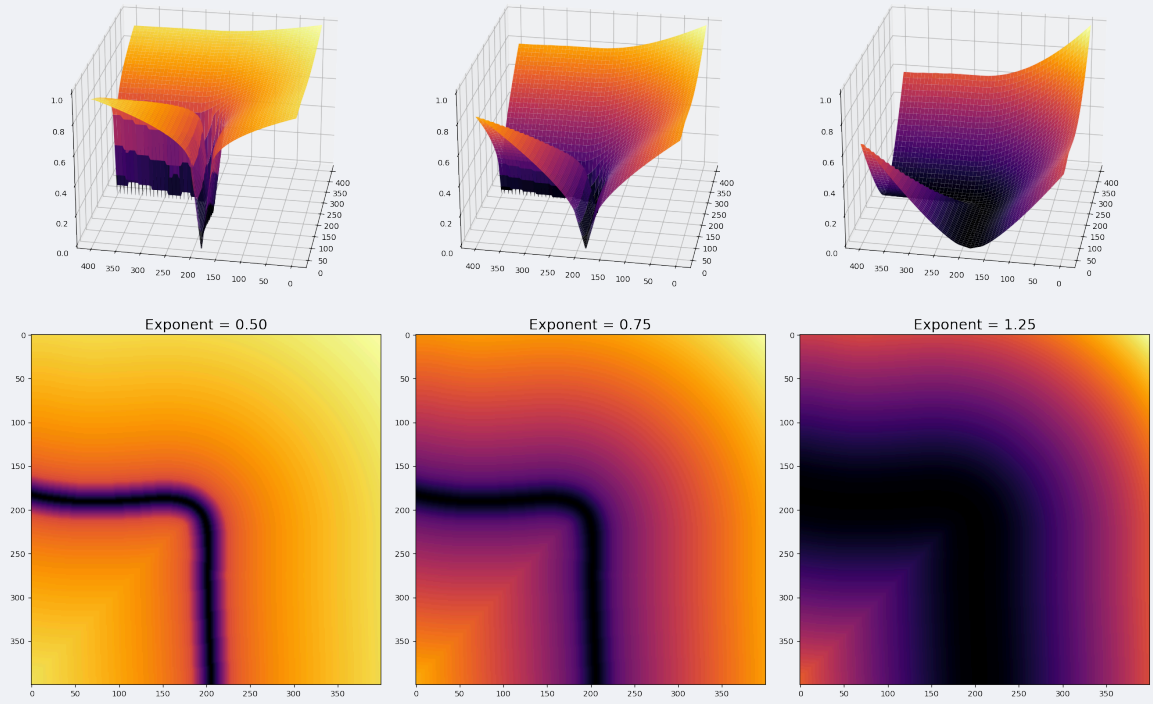


Figure 18. 3D plots of cold maps along with their 2D counterparts.

Finally, a comparison between the retrieved satellite image of an intersection, the optimal path through it, and the cold map generated by the process described above are shown in Figure 19. This highlights the importance of the cold map in the training process as opposed to the single line path. The cold map allows for a more lenient path to be generated, as the model is not penalized for deviating slightly from the path.

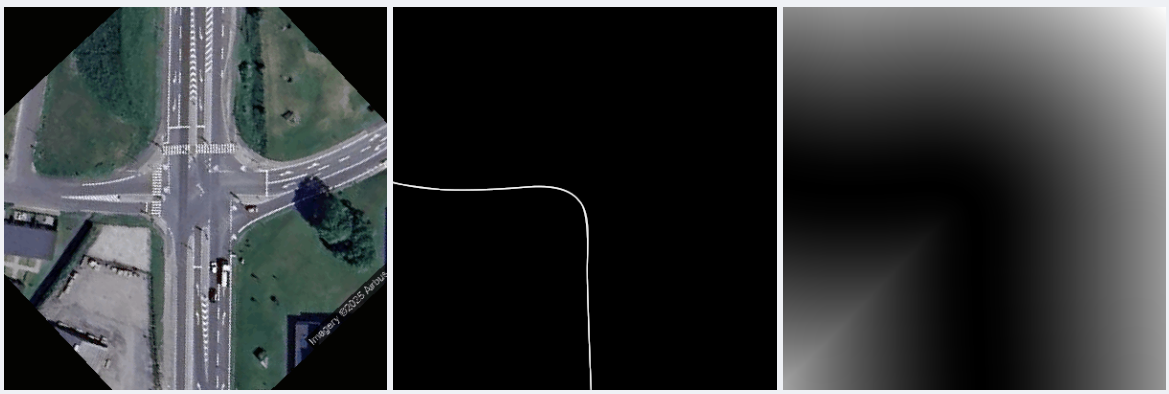


Figure 19. Example of satellite image, next to the desired path through with. To the far right, the generated cold map is shown with threshold $t = 20$ and exponent $e = 1.25$. Notice how it is only the points very close to the path that are very cold, while the rest of the map is warmer the further away it is.

4.3.2 Data Augmentation

Creating large datasets is a very time consuming tasks, scaling directly with the complexity and workflows structured around its creation. For the dataset created during this project, the workflow was as follows: Find a suitable intersection for the dataset. Copy the coordinates for the center of the intersection. Use the found coordinates in the satellite script described in Section 4.1.2 to download satellite images. Through trial and error, rotate the downloaded satellite image to align entry with bottom of the image.

Once a bunch of satellite images were downloaded, a small python script was used to automatically distribute each intersection image to their own folder and within each folder, create the structure shown in Listing 17. GNU Image Manipulation Program (GIMP) was chosen as the software to draw the paths through the intersections. First, another small script distributed a `.xcf` file to each intersection folder. `.xcf` is the file format for GIMP projects. This base `.xcf` was defined to be 400x400 pixels and contained a black background and three empty layers named “left”, “right”, and “ahead”. This massively simplified the process of creating the paths by not having to create a new project every time a new intersection was to be processed.

For each of the paths drawn in GIMP, they were saved individually as a `.png` file. Yet another small script then used these images of the path to create the corresponding JavaScript Object Notation (JSON) files containing the entry and exit coordinates of the path as well as generate the cold map. The cold map generated is stored as a `.npy` files, as the values of the cold map are simply between 0 and 1, meaning it does not make sense to store as a Portable Network Graphics (PNG) as there is not high enough values to be discernible to the human eye. The small scripts mentioned can be seen in Appendix X.

As described, this is a very time consuming process, and despite many hours of work being put into it, the training dataset only consisted of 112 intersections, some of which have very similar satellite images. To enlarge this dataset dramatically, the dataset underwent augmentation. Augmentation can be done in many ways with different methods. A variety of augmentations were chosen for this dataset, including: colouration, distortion, cropping, and zooming. The reason for choosing these will be discussed in their respective sections.

COLOURATION is the augmentation regarding the colours making up the image. In this project, this is achieved by adjusting the saturation and hue of the HSV colour space for an image. HSV stands for Hue, Saturation, and Value. Changing the hue of an image is changing the colour tone, meaning that a red image can be turned into a blue image. Changing the saturation is changing the intensity of the colours in an image, resulting in a more vibrant or dull image. Changing the value is changing the brightness of the image, meaning that a dark image can be turned much brighter and vice versa. HSV is generally more intuitive than the RGB colour space, as it is more closely related to how humans perceive colour.

Concretely, the colouration augmentation was done by randomly changing the hue and saturation of the image. This was done to help the models focus on structural features rather than specific colour cues, i.e. become better at generalizing. Colour augmentations also

help the model become more robust to changes in lighting conditions. This is especially prominent when using satellite images from all kinds of areas. Some satellite images appear to have a very low image saturation, while others are more vibrant and sharp. Therefore, teaching the model to understand these different conditions is crucial. So, these colouration augmentations help make the models more adept at generalizing to different conditions.

The code for the hue and saturation augmentation functions can be seen in Listing 11 and Listing 12, respectively. The hue augmentation function randomly changes the hue of the image by a value between the lower and upper bounds. The values are the defaults from the official documentation of the function. The saturation augmentation function randomly changes the saturation of the image by a value between 6, 8, and 10. These values were chosen as they were found to be the most visually appealing and interesting when testing the augmentations. Finally, a greyscale augmentation is also implemented, which is simply a call of the `saturation_aug` function with the value 0. This is done to further enhance the models understanding of the structural features of the image.

Hue augmentation

The hue augmentation was done by randomly changing the hue of the image. The hue was changed by a value between −0.5 and 0.5. The image was then converted to a tensor and the hue was adjusted using the `adjust_hue` function from the `torchvision.transforms.functional` module. The resulting image was then converted back to a PNG image for easier handling.

```
def hue_aug(image,
    lower = -0.5, upper = 0.5):
  v = random.uniform(lower, upper)
  img = T.ToTensor()(image)
  img = F.adjust_hue(img, v)

  return T.ToPILImage()(img)
```

Listing 11. Hue augmentation function.

Saturation augmentation

The saturation augmentation was done by randomly changing the saturation of the image. The saturation was changed by a value between 6, 8, and 10. The image was then converted to a tensor and the saturation was adjusted using the `adjust_saturation` function from the `torchvision.transforms.functional` module. The resulting image was then converted back to a PNG image for easier handling.

```
def saturation_aug(image,
    val = [6, 8, 10]):
  v = random.choice(val)
  img = T.ToTensor()(image)
  img = F.adjust_saturation(img, v)

  return T.ToPILImage()(img)
```

Listing 12. Saturation augmentation function.

Examples of the colouration augmentations can be seen in Figure 20. Each column shows an intersection and its augmented variations. The top row is the original image, the second row is the greyscale augmented image, the third row is the hue adjusted image, and the fourth row is the saturation adjusted image. The greyscale images highlights the structural features of the image. By adjusting the hue, the dominant tones of the image are altered, resulting in dominant parts like vegetation appears as a variety of colours, such as blue or even purple. Adjusting saturation then makes the colours more vibrant or muted, creating anything from intensely vivid scenes to nearly colourless landscapes as also highlighted by the greyscale image.

Seeing these colour augmentation examples, it is clear to see that a large amount of diversity has been introduced to the dataset. Rather than training on a dataset where the colours might be very similar, hence not capturing the real world, the models are trained on a dataset that encapsulates real world colour variations. Furthermore, this motivates the model to focus on structural features rather than specific colour cues, which is crucial for generalization.

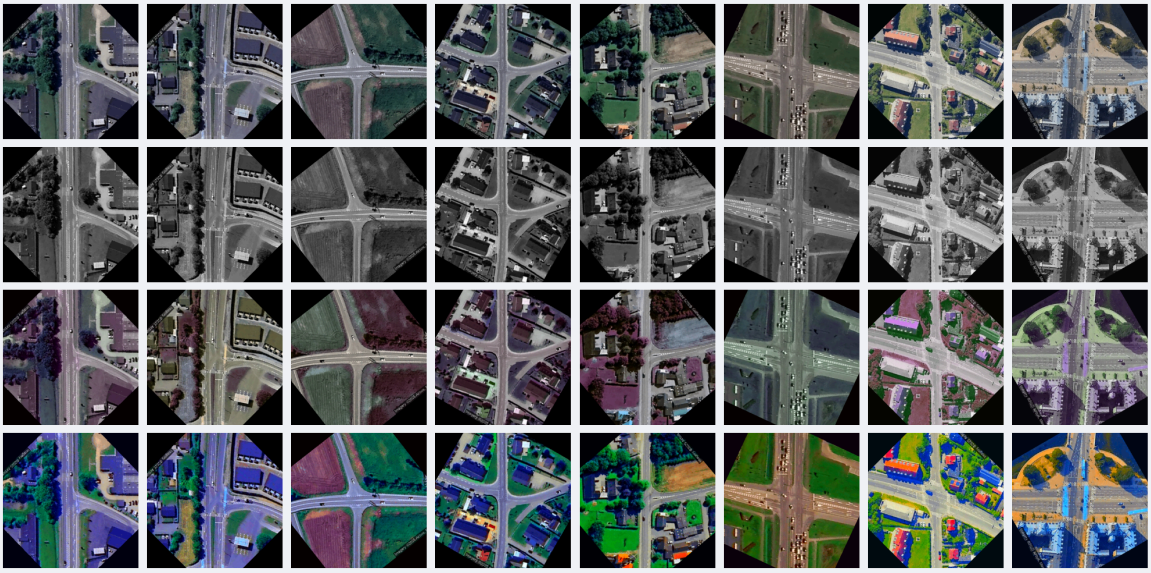


Figure 20. Example of the colouration augmentations. The top row is the original image, second row is the greyscale augmented image, third row is the hue adjusted image, and fourth row is the saturation adjusted image.

DISTORTION is the augmentation regarding the quality of the image. In this project, this is achieved by applying two methods of intentionally deteriorating the image quality. The first method is noise augmentation and the second is blurring the image. These augmentation were chosen, as they represent common downfalls when working with satellite images. Depending on the area where images are taken, the images are often more blurry in smaller towns and rural areas, while they are noticeably sharper and more pristine in larger cities like capitol cities. Thus, by incorporating distortion augmentations into the dataset, it becomes much more diverse and a better representation of the quality of images that the models are expected to work with.

The noise augmentation was chosen as it is a common issue with images in general, not just satellite images. Noise is produced in images through various outside factors, such as atmospheric conditions, sensor limitations, and environmental interferences. By adding this noise augmentation to the images, it helps the model learn to ignore these artifacts and focus on the relevant features of the image. The noise augmentation was done by generating Gaussian noise through the `randn` function from the PyTorch library. Furthermore, this kind of augmentation has been noted to act as a form of regularization on it own [58], as it prevents the model from overfitting to these clean, ideal, and pristine images that are common for very populated areas.

The blur augmentation was chosen as blur is a common issue with satellite images, particularly when using images from smaller cities, rural areas, older images, or far out of the cities on the roads where population density is significantly lower. Applying a Gaussian blur simulates these conditions by intentionally deteriorating the image quality, making it more representative of real-world scenarios. This helps the model learn to extract structural features from the image, despite the images being obscured. This augmentation is particularly useful because it forces the model to focus on the structural features of the image, rather than the fine details. This is crucial for the model to generalize well to unseen data, as it is not expected to see the same images during inference as it did during training.

The code for the noise and blur augmentations can be seen in Listing 13 and Listing 14, respectively. The noise augmentation function generates random noise using a normal distribution with a mean of 0 and a standard deviation between 0.1 and 0.5. The noise is then added to the image and the resulting image is clamped to a value between 0 and 1 before being returned as an image. The blur augmentation function applies a Gaussian blur to the image using a kernel size of 5, 7, or 9 and a sigma value of 1.5, 2, or 2.5. The kernel size and sigma values were chosen through testing to find the most visually appealing results, i.e. distortions great enough to distort the details of the image, but not so much that the image becomes unrecognizable.

Noise augmentation

The noise augmentation was done by generating random noise and adding it to the image. The noise was generated using a normal distribution with a mean of 0 and a standard deviation between 0.1 and 0.5. The noise was then added to the image and the resulting image was clamped to a value between 0 and 1. The resulting image was then converted back to a PNG image for easier handling.

```
def noise_aug(image, mean = 0, std_l =
0.1, std_u = 0.5):
  img = T.ToTensor()(image)
  std = random.uniform(std_l, std_u)
  noise = randn(img.size()) * std + mean
  img = img + noise
  img = torch.clamp(img, 0, 1)

  return T.ToPILImage()(img)
```

Listing 13. Noise augmentation function.

Blur augmentation

The blur augmentation was done by applying a Gaussian blur to the image. Through testing, the kernel sizes of 5, 7, and 9 were chosen, as well as the sigma values of 1.5, 2, and 2.5. After randomly selecting the combination of kernel size and sigma, the image is converted to a tensor and the blur is applied. The resulting image is then converted back to a PNG image for easier handling.

```
def blur_aug(image,
    kernel_size = [5, 7, 9],
    sigma = [1.5, 2, 2.5]):
  kernel_size = choice(kernel_size)
  sigma = choice(sigma)
  img = T.ToTensor()(image)
 img = F.gaussian_blur(img, kernel_size,
sigma)

  return T.ToPILImage()(img)
```

Listing 14. Blur augmentation function.

Examples of the distortion augmentations can be seen in Figure 21. The image to the far left-hand side is the original image, the top row is the noise augmented images, and the bottom row is the blur augmented images. The noise augmented images show the image with added noise, making certain features difficult to see and the image more obscured. The blur augmented images show the image with a Gaussian blur applied, making the

image more obscured and less sharp. Seeing these examples, it is clear to see that, again, a large amount of diversity has been introduced to the dataset. The introduction of noise and blur further help the model generalize better by focusing on structural features rather than specific details. This addition to the dataset broadens the ability of the models and teaches them to perform better on suboptimal images.

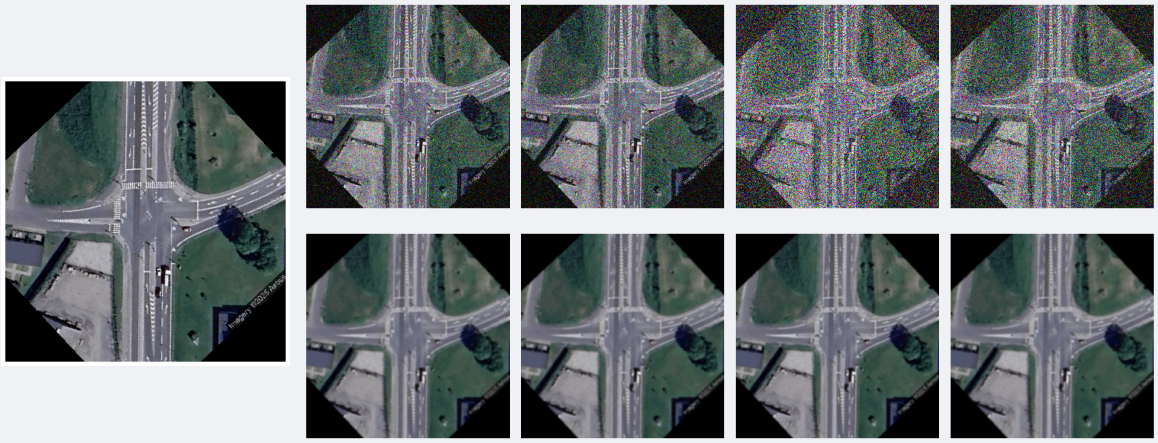


Figure 21. Example of the distortion augmentations. The far left image is the original image, the top row is the noise augmented images, and the bottom row is the blur augmented images.

CROP AND ZOOM is the last augmentation adopted to expand the dataset. They are very common spatial augmentations used on image datasets. They are typically selected to make the models trained more robust against non-centred images and to make the models more adept at generalizing to different scales. If centring is not an issue, then cropping is still a common method for classification tasks, as it helps the model focus on the relevant features of the image. Zooming is also a common augmentation, as it helps the model learn different spatial scales of whatever it is being taught to handle. Also, as discussed earlier in Section 4.1, the zoom level of images in the dataset is set to a value of 18, but these images still appear to consist of vastly different sized intersection, thanks to the fact that some intersection are simply larger than others. Thus, using a zoom augmentation, helps the models generalize better to different scales of intersections. Furthermore, this should also help the model gain a better understanding of road width and not overfit to any particular width present in the dataset.

So, both of these augmentations help the trained models become scale invariant, which is a desired trait of models set to perform the task at hand, as the scale and size of intersection images gotten through satellite images can vary greatly. The crop augmentation was done by randomly cropping the satellite image and its corresponding paths. The crop size is determined by a factor of the original image size, with a default value of 0.8. The cropped image and paths are then resized back to the original dimensions to maintain consistency. This augmentation helps the model focus on different parts of the image and improves its robustness to non-centred features, as is highly relevant for this project. The zoom augmentation was done by resizing the image to a larger size and then cropping it back to the original size. Through testing, the zoom factors of 1.4 to 1.9 were chosen. After randomly selecting a zoom factor, the image is resized and cropped to simulate zooming in.

Crop augmentation

The crop augmentation was done by randomly cropping the satellite image and its corresponding paths. The crop size is determined by a factor of the original image size, with a default value of 0.8. The cropped image and paths are then resized back to the original dimensions to maintain consistency. This augmentation helps the model focus on different parts of the image and improves its robustness to non-centred features.

```
def crop_aug(sat_image, paths, factor =
0.8):
    h, w = sat_image.size
    new_h = int(h*factor)
    new_w = int(w*factor)
    top = random.randint(0, h - new_h)
    left = random.randint(0, w - new_w)
    img = F.crop(sat_image, top, left,
                 new_h, new_w)
    scaled_img = F.resize(img, (h, w))

    path_images = []
    for p in paths:
        path = p["path_line"]
        path_img = F.crop(path, top, left,
                          new_h, new_w)
        scaled_path = F.resize(path_img,
            (h, w), interpolation=LANCZOS)
        path_images.append(scaled_path)

    return scaled_img, path_images
```

Listing 15. Crop augmentation function.

Zoom augmentation

The zoom augmentation was done by resizing the image to a larger size and then cropping it back to the original size. Through testing, the zoom factors of 1.4 to 1.9 were chosen. After randomly selecting a zoom factor, the image is resized and cropped to simulate zooming in. The resulting image is then converted back to a PNG image for easier handling.

```
def zoom_aug(image, paths,
             zoom_range = (1.4, 1.9)):
  zoom_factor = uniform(*zoom_range)
  tmp_h = int(image.size[0]*zoom_factor)
  tmp_w = int(image.size[1]*zoom_factor)
  img = F.resize(image, (tmp_h, tmp_w))
  img = F.center_crop(img, image.size)

  path_images = []
  for p in paths:
    path = p["path_line"]
    path_img = F.resize(path,
                        (tmp_h, tmp_w))
    path_img = F.center_crop(path_img,
                             image.size)
    path_images.append(path_img)

  return img, path_images
```

Listing 16. Zoom augmentation function.

Examples of the crop and zoom augmentations can be seen in Figure 22. The far left image is the original image, the top row is the crop augmented images, and the bottom row is the zoom augmented images. The crop augmented images show the image cropped to a different part of the image, with the associated paths being cropped by the same factors. This was a necessary step, separating these augmentations from the previous, as the paths through the image were not the exact same as when the image underwent colouration or distortion augmentations. The same is true for the zoom augmented image. Here, the paths also needed to undergo the same augmentation as the satellite image. Once again, these augmentation have clearly impacted the diversity of the dataset, as the models are now trained on images that are not centred and images that are zoomed in, which is important for generalization.

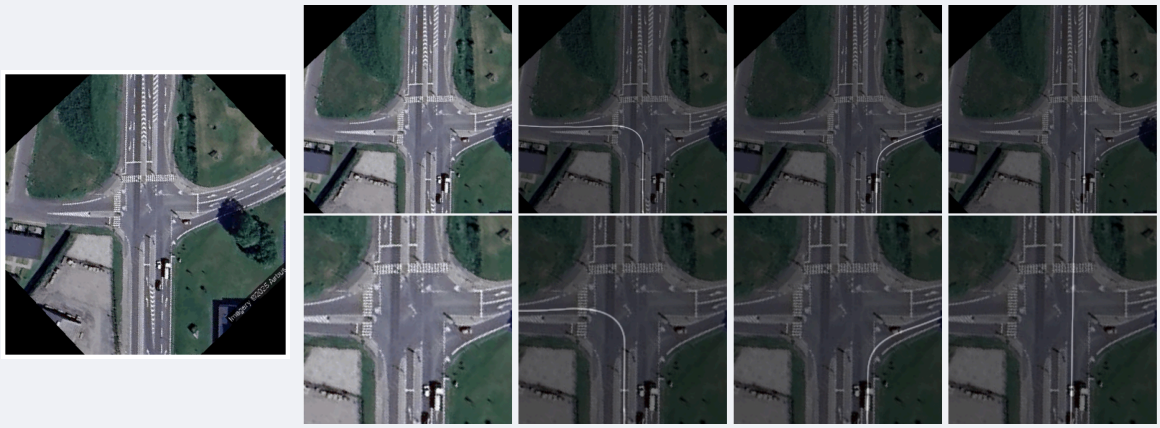


Figure 22. Example of the crop and zoom augmentations. The far left image is the original image, the top row is the crop augmented images, and the bottom row is the zoom augmented images.

All in all, these augmentations were selected to expand the dataset by introducing real-world inspired variations of the satellite images. These augmentation mimic the unpredictable conditions posed by using satellite imagery. By employing techniques such as hue and saturation adjustments, greyscale conversions, noise injection, blur filtering, and spatial transformations like cropping and zooming, the augmented dataset now contains diverse lighting, quality, and scale entries. This strategy of employing such a variety of augmentations not only ballooned the size of the dataset immensely, but also made the dataset more representative of the real-world conditions the models are expected to work with, meaning that it should be more robust and better at generalizing to unseen data.

Other augmentation techniques were considered for this project. Rotation, for example, was considered to further increase the diversity of the dataset by introducing variations in orientation, which could help the model learn to recognize paths from different angles. However, early on in this project, it was decided that the entry for each path should be at the bottom of the satellite image. Thus, rotation the images could have undesired consequences for the models' ability to generalize with this constraint. Flipping was also considered, but was ultimately left out, as it was deemed to have too little of a potential impact since many intersections already go in all directions. Translation was also considered, but was left out, as it would require the paths to be redrawn as they would no longer reach the edges of the image, which was a factor this entire set out to combat. Finally, a very common augmentation used in segmentation, is the act of using cutmix and its constituent parts, namely cutout and mixup. These were, however, also left out, as it is assumed that the satellite images do not contain holes or other artefacts that would be introduced by these augmentations. Furthermore, cutmix might disrupt the spatial features understood by the model. These may be introduced in future work, as they may ultimately increase the robustness of the models in environments that have blocking features.

4.3.3 Dataset Structure

UPDATE TO INCLUDE CLASS_LABELS

To maintain an ease-of-use principle for this project, the dataset was structured in a way that allows for easy loading of the data. This includes building the dataset in a logical way, and creating a class that can load the dataset gracefully. This is especially important as the paths in a dataset can vary in number, so custom loading is necessary. Thus, the dataset is structured like shown in the listing below

```
dataset/
├── train/
│   ├── intersection_001/
│   │   ├── satellite.png
│   │   ├── class_labels.npy
│   │   ├── paths/
│   │   │   ├── path_1/
│   │   │   │   ├── path_line.png
│   │   │   │   ├── path_line_ee.json
│   │   │   │   ├── cold_map.npy
│   │   │   ├── path_2/
│   │   │   │   ├── path_line.png
│   │   │   │   ├── path_line_ee.json
│   │   │   │   ├── cold_map.npy
│   │   │   ├── path_3/
│   │   │   │   ├── path_line.png
│   │   │   │   ├── path_line_ee.json
│   │   │   │   ├── cold_map.npy
├── test/
│   ├── intersection_001/
│   │   ...
```

Listing 17. Folder structure of the dataset. Each `intersection_XXX` folder contains a satellite image of an intersection and a `paths` folder containing the paths through the intersection. Each path folder contains the path line, the path's entry and exit points, and the cold map for the path in a `.npy` format.

Firstly, the dataset is split into two separate parts, `train` and `test`. This is done to ensure that the model is not overfitting to the training data. This is achieved by training the model on the `train` dataset and testing/validating it on the `test` dataset. To ensure that the models generalize well to the task as hand, the `test` dataset should contain intersections that are completely absent from the `train` dataset. This is done to ensure it does not fall into the simple trap of memorizing the training data and created really good results that can be considered false positives as it supposedly has never seen the data before.

This `train`/`test` split in the dataset is created in the folder structure instead of using the simpler functionalities offered by PyTorch. PyTorch offers a `random_split` function from its utility sub-library. This function takes in some dataset declared as a PyTorch `Dataset` object, as shown below in Section 4.3.3.1, and splits it based on a given ratio. This is a simple way to split the dataset, but, as is the case of the created dataset, some images are very similar, meaning that the split does not achieve the desired effect and the model overfits to the training data. Thus, a completely different set of intersections is used for the `test` dataset.

Each `intersection_XXX` folder contains a satellite image saved as a PNG. Accompanying this image, is the `paths` folder, which contains a folder for each path through the intersection. Each path folder contains the path line image, currently saved as a PNG as well, a JSON file containing the entry and exit points of the path in relation to the image, not the global coordinates, and the corresponding cold map saved as a `.npy` file.

4.3.3.1 Dataset class

UPDATE ALL TO REFLECT LATEST CHANGES. MAYBE ALSO DIFFERENT TYPES.

To be able to easily load a satellite image and its corresponding paths, entry/exit points, and cold maps, a `IntersectionDataset` class was created, built on top of the PyTorch `Dataset` class. To implement this class, three functions must be created, namely `__init__`, `__len__`, and `__getitem__`.

`__init__` is the function called when the class is instantiated. It initializes the dataset with the root directory of the dataset, a transform function, and a path transform function. The root directory is the directory where the dataset is stored, the transform function is a function that can be applied to the satellite image, and the path transform function is a function that can be applied to the path line. The root directory passed to the instantiation should that of either the training or test dataset within the dataset root folder. The transforms are simply `ToTensor` functions provided by PyTorch. The `__init__` function also creates a list of all the paths in the dataset by listing all directories found in the `paths` folders. The code for the `__init__` function can be seen in Listing 18 below.

```
def __init__(self, root_dir, transform = None, path_transform = None):
  self.root_dir = root_dir
  self.transform = transform
  self.path_transform = path_transform

  self.path_dirs = glob.glob(f'{root_dir}/*/paths/*')
```

Listing 18. Code snippet of the `__init__` function for the dataset.

Here the library `glob` is used to list all directories found in the `paths` folders. It is a useful library that allows for the use of wildcards in the path, making it easy to find all directories in the `paths` folders. This approach was used as it was not certain that all `paths` folders contained three subfolders, meaning that there might be inconsistencies in how the data is structured across different intersections. This approach ensures that all paths are found, regardless of the structure of the `paths` folder.

`__len__` is another function required by the PyTorch `Dataset` class. It returns the length of the dataset. Thanks to the initialization of the dataset in the `__init__` function, the length of the dataset is simply the number of paths in the dataset. The code for the `__len__` function can be seen in Listing 19 below.

```
def __len__(self):
  return len(self.path_dirs)
```

Listing 19. Code snippet of the `__len__` function for the dataset.

`__getitem__` is one of the most crucial functions of the dataset class. The signature of the function is simply `__getitem__(self, idx)`, where `idx` is the index of the path to be loaded. First, the function retrieves the directory of the path at the given index. Then, it loads the satellite image from the intersection directory and applies the

transform function to it. This is achieved by moving up by two directories, i.e. getting the satellite image from the `intersection_XXX` folder. It then loads the path itself and applies the path transform function to it. These transforms are simply `ToTensor` as provided by PyTorch. Then, for the path being indexed, the function loads the JSON file containing the entry and exit points of the path, and the cold map. The entry/exit data is loaded from a JSON file, and the cold map is loaded from a `.npy` file. All of this data is then stored in a dictionary and returned as the sample. The code for the `__getitem__` function can be seen in Listing 20 below.

```
def __getitem__(self, idx):
  path_dir = self.path_dirs[idx]

  # Load satellite image (../../satellite.png)
        satellite_path    =    os.path.join(os.path.dirname(os.path.dirname(path_dir)),
'satellite.png')
  satellite_img = Image.open(satellite_path).convert('RGB')

  if self.transform:
    satellite_img = self.transform(satellite_img)

  # load path line image (./path_line.png)
  path_line_path = os.path.join(path_dir, 'path_line.png')
  path_line_img = Image.open(path_line_path).convert('L')

  if self.path_transform:
    path_line_img = self.path_transform(path_line_img)[0]

  # load E/E json file (./path_line_ee.json)
  json_path = os.path.join(path_dir, 'path_line_ee.json')
  with open(json_path) as f:
    ee_data = json.load(f)

  # load cold map npy (./cold_map.npy)
  cold_map_path = os.path.join(path_dir, 'cold_map.npy')
  cold_map = np.load(cold_map_path)

  # return sample
  sample = {
    'satellite': satellite_img,
    'path_line': path_line_img,
    'ee_data': ee_data,
    'cold_map': cold_map
  }
  return sample
```

Listing 20. Code snippet of the `__getitem__` function for the dataset.

The dataset is then simply instantiated as such:

```
dataset = IntersectionDataset(root_dir=dataset_dir,
                              transform=ToTensor(),
                              path_transform=ToTensor())
```

Listing 21. Instantiation of the dataset.

where `dataset_dir` is the path to either the training or test dataset folders. Creating the dataloader is as simple as:

```
dataloader = DataLoader(dataset,
                        batch_size=b,
                        shuffle=True,
                        num_workers=num_workers)
```

Listing 22. Creating a dataloader for the dataset.

Arguments passed to the `DataLoader` initializer are the dataset from Listing 21, the batch size, whether the dataset should be shuffled, the number of workers to use for loading the data, and the ability to give it a custom collate function, which is not necessary in this case as the default function `default_collate` handles the data gracefully. `num_workers` is found using the `multiprocessing` library as it easily finds the number of available computation cores. An example of the dataloader in action is shown in Figure 23, where an example batch from the `DataLoader` is shown. The batch contains four paths, showcasing the path and the associated satellite image.

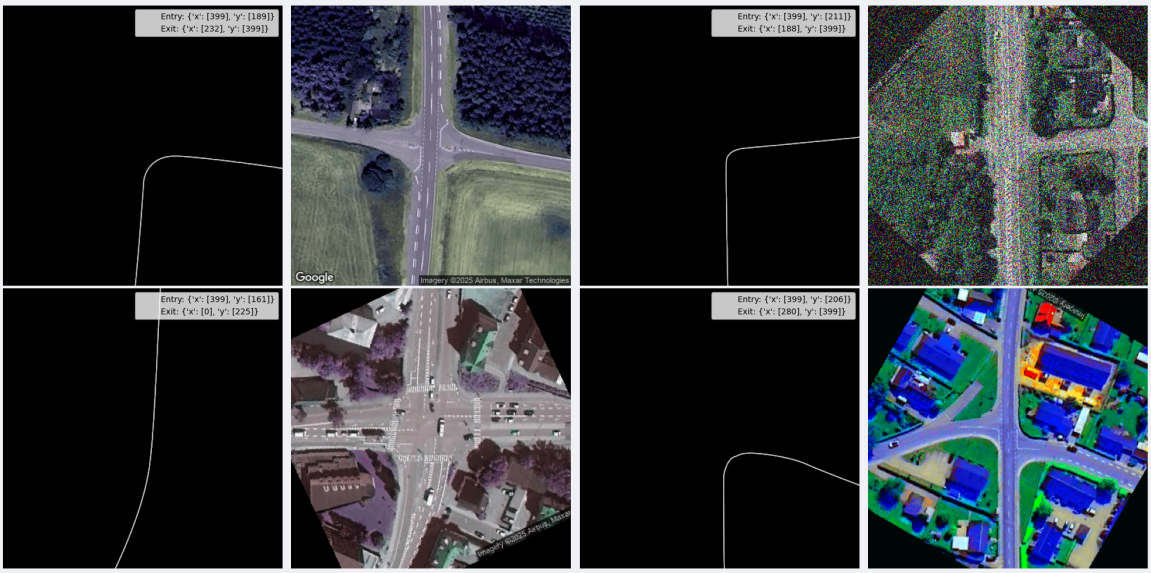


Figure 23. Example batch from the `DataLoader` with `batch_size = 4`.

4.4 The Models

With all the previous sections covered, the next step is to define and discuss the models that will be used to predict the path through intersection. The goal is to create and test various models that do this task well, evaluating their performance and comparing them to each other. The models used are the very bare-bones version of the models, with little to no modifications applied to them. This approach was chosen as it may provide a better understanding of what kind of backbone for a model might yield the greatest results in the context of path-planning.

4.4.1 U-Net

Hello

DoubleConv

```
class DoubleConv(nn.Module):
def __init__(self, in_c, out_c):
  super(DoubleConv, self).__init__()
  self.double_conv = nn.Sequential(
    nn.Conv2d(...),
    nn.BatchNorm2d(...),
    nn.ReLU(...),
    nn.Conv2d(...),
    nn.BatchNorm2d(...),
    nn.ReLU(...)
  )
def forward(self, x):
  return self.double_conv(x)
```

Listing 23.

Down

```
1  class Down(nn.Module):
2    def __init__(self, in_c, out_c):
3      super(Down, self).__init__()
4      self.maxpool_conv = nn.Sequential(
5        nn.MaxPool2d(...),
6        DoubleConv(...)
7      )
8
9    def forward(self, x):
10     return self.maxpool_conv(x)
```

Listing 24.

Up

```
class Up(nn.Module):
  def __init__(self, in_c, out_c,
                     bilinear=True):
      super(Up, self).__init__()
    if bilinear:
      self.up = nn.Upsample(...)
    else:
      self.up = nn.ConvTranspose2d(...)
    self.conv = DoubleConv(...)

  def forward(self, x1, x2):
    x1 = self.up(x1)
    x1 = F.pad(x1, ...)
    x = torch.cat([x2, x1], dim=1)
    return self.conv(x)
```

Listing 25.

OutConv

```
class OutConv(nn.Module):
def __init__(self, in_c, out_c):
  super(OutConv, self).__init__()
  self.conv = nn.Conv2d(...)
def forward(self, x):
  return self.conv(x)
```

Listing 26.

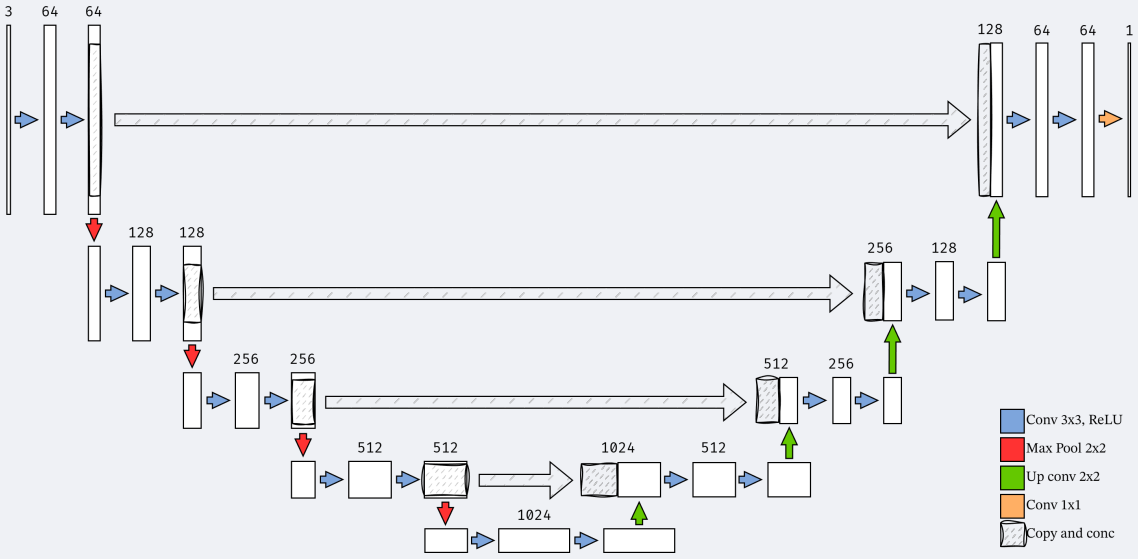


Figure 24. U-Net Architecture.

5 Results

This section details the experiments conducted

6 Discussion

In this section...

6.1 Integration with existing systems

6.2 Shortcomings

6.3 Future Work

6.4 Other considerations

7 Conclusion

References

[1] C. M. University, “The Carnegie Mellon University Autonomous Land Vehicle Project.” [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/alv/www/index.html

[2] C. Thorpe, M. Hebert, T. Kanade, and S. Shafer, “Vision and navigation for the Carnegie-Mellon Navlab,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 3, pp. 362–373, 1988, doi: 10.1109/34.3900.

[3] J. Billington, “The Prometheus project: The story behind one of AV's greatest developments.” [Online]. Available: https://www.autonomousvehicleinternational.com/features/the-prometheus-project.html

[4] A. S. Francisco, [Online]. Available: https://abc7news.com/post/wrong-waymo-driverless-car-goes-oncoming-traffic-tempe-arizona/15238556/

[5] “Technical milestone in road safety: experts praise Volkswagen's Car2X technology.” [Online]. Available: https://www.volkswagen-newsroom.com/en/press-releases/technical-milestone-in-road-safety-experts-praise-volkswagens-car2x-technology-5914

[6] K. Dresner and P. Stone, “A multiagent approach to autonomous intersection management,” *Journal of artificial intelligence research*, vol. 31, pp. 591–656, 2008.

[7] A. P. Chouhan and G. Banda, “Autonomous Intersection Management: A Heuristic Approach,” *IEEE Access*, vol. 6, no. , pp. 53287–53295, 2018, doi: 10.1109/ACCESS.2018.2871337.

[8] Z. Zhong, M. Nejad, and E. E. Lee, “Autonomous and Semiautonomous Intersection Management: A Survey,” *IEEE Intelligent Transportation Systems Magazine*, vol. 13, no. 2, pp. 53–70, 2021, doi: 10.1109/MITS.2020.3014074.

[9] M. Cederle, M. Fabris, and G. A. Susto, “A Distributed Approach to Autonomous Intersection Management via Multi-Agent Reinforcement Learning.” [Online]. Available: https://arxiv.org/abs/2405.08655

[10] On-Road Automated Driving (ORAD) committee, “Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems,” SAE International, 400 Commonwealth Drive, Warrendale, PA, United States, 2017.

[11] On-Road Automated Driving (ORAD) committee, “Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles,” SAE International, 400 Commonwealth Drive, Warrendale, PA, United States, 2021.

[12] S. E. Staff, “The 6 Levels of Vehicle Autonomy Explained.” [Online]. Available: https://www.synopsys.com/blogs/chip-design/autonomous-driving-levels.html

[13] K. Buchholz, “Infographic: Cars Increasingly Ready for Autonomous Driving.” [Online]. Available: https://www.statista.com/chart/25754/newly-registered-cars-by-autonomous-driving-level/

[14] K. Buchholz, “Autonomous driving is taking over.” [Online]. Available: https://www.weforum.org/stories/2023/02/charted-autonomous-driving-accelerating-mobility/

[15] F. A. Mall, “The History of Cruise Control | Folsom Auto Mall.” [Online]. Available: https://www.folsomautomall.com/blog/2022/november/21/the-history-of-cruise-control.htm

[16] F. Danmark, “Ford BlueCruise | Håndfri kørsel i Blue Zones | Ford DK.” [Online]. Available: https://www.ford.dk/om-os/foererassistancesystemer/ford-bluecruise

[17] C. Hoffmann, [Online]. Available: https://www.shop4tesla.com/en/blogs/news/tesla-fsd-supervised-europa-2025

[18] P. Davies, [Online]. Available: https://autoseu.com/you-can-now-drive-partially-hands-free-in-france-this-is-whats-changing/

[19] C. Murray, [Online]. Available: https://www.designnews.com/autonomous-vehicles/here-s-why-level-5-autonomous-cars-may-still-be-a-decade-away

[20] J. Marie, “The Power of Perception: Cameras in Self-Driving Cars.” [Online]. Available: https://supplybridge.com/the-power-of-perception-cameras-in-self-driving-cars/

[21] J. Cohen, “Sensor Fusion - Fusing LiDARs & RADARs in Self-Driving Cars.” [Online]. Available: https://www.thinkautonomous.ai/blog/sensor-fusion/#:~:text=A%20Kalman%20filter%20can%20be,prediction

[22] D. J. Yeong, K. Panduru, and J. Walsh, “Exploring the Unseen: A Survey of Multi-Sensor Fusion and the Role of Explainable AI (XAI) in Autonomous Vehicles,” *Sensors*, vol. 25, no. 3, 2025, doi: 10.3390/s25030856.

[23] J. Cohen, “9 Types of Sensor Fusion Algorithms.” [Online]. Available: https://www.thinkautonomous.ai/blog/9-types-of-sensor-fusion-algorithms/

[24] W. Franklin, “Kalman Filter Explained Simply - The Kalman Filter.” [Online]. Available: https://thekalmanfilter.com/kalman-filter-explained-simply/

[25] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[26] Z. Parvez, “The Pioneers of AI: Marvin Minsky and the SNARC.” [Online]. Available: https://zahid-parvez.medium.com/history-of-ai-the-first-neural-network-computer-marvin-minsky-231c8bd58409

[27] [Online]. Available: https://www.ibm.com/history/early-games

[28] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain,” *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, Nov. 1958.

[29] R. Karjian, “History and evolution of machine learning: A timeline.” [Online]. Available: https://www.techtarget.com/whatis/feature/History-and-evolution-of-machine-learning-A-timeline

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., Curran Associates, Inc., 2012, p. . [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[32] [Online]. Available: https://www.wikiwand.com/en/articles/Gradient_descent

[33] G. Boesch, "Computer Vision Tasks (Comprehensive 2025 Guide) - viso.ai." [Online]. Available: https://viso.ai/deep-learning/computer-vision-tasks/

[34] [Online]. Available: https://openai.com/index/introducing-4o-image-generation/

[35] D. Jagula, "Satellite Imagery for Everyone." [Online]. Available: https://spectrum.ieee.org/commercial-satellite-imagery#:~:text=The%20best%20commercially%20available%20spatial,Moderate

[36] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math. (Heidelb.)*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[37] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.

[38] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp. 3310–3317. doi: 10.1109/ROBOT.1994.351061.

[39] A. (Tony) Stentz, "The Focussed D* Algorithm for Real-Time Replanning," in *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, Aug. 1995, pp. 1652–1659.

[40] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005, doi: 10.1109/TRO.2004.838026.

[41] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," in *Autonomous Robot Vehicles*, I. J. Cox and G. T. Wilfong, Eds., New York, NY: Springer New York, 1990, pp. 396–404. doi: 10.1007/978-1-4613-8997-2_29.

[42] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, 1991, pp. 1398–1404. doi: 10.1109/ROBOT.1991.131810.

[43] V. Gazi, "Swarm aggregations using artificial potentials and sliding-mode control," *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1208–1214, 2005, doi: 10.1109/TRO.2005.853487.

[44] N. Leonard and E. Fiorelli, "Virtual leaders, artificial potentials and coordinated control of groups," in *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, 2001, pp. 2968–2973. doi: 10.1109/CDC.2001.980728.

[45] S. M. LaValle, "Rapidly-exploring random trees : a new tool for path planning," *The annual research report*, 1998, [Online]. Available: https://api.semanticscholar.org/CorpusID:14744621

[46] S. Karaman and E. Frazzoli, “Sampling-based Algorithms for Optimal Motion Planning.” [Online]. Available: https://arxiv.org/abs/1105.1186

[47] R. Cui, Y. Li, and W. Yan, “Mutual Information-Based Multi-AUV Path Planning for Scalar Field Sampling Using Multidimensional RRT*,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 7, pp. 993–1004, 2016, doi: 10.1109/TSMC.2015.2500027.

[48] M. Xanthidis *et al.*, “Navigation in the Presence of Obstacles for an Agile Autonomous Underwater Vehicle,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 892–899. doi: 10.1109/ICRA40945.2020.9197558.

[49] C. S. TAN, “A Collision Avoidance System for Autonomous Underwater Vehicles.” [Online]. Available: https://pearl.plymouth.ac.uk/secam-theses/302/

[50] C. Lamini, S. Benhlima, and A. Elbekri, “Genetic Algorithm Based Approach for Autonomous Mobile Robot Path Planning,” *Procedia Computer Science*, vol. 127, pp. 180–189, 2018, doi: https://doi.org/10.1016/j.procs.2018.01.113.

[51] L. Zadeh, “Fuzzy sets,” *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965, doi: https://doi.org/10.1016/S0019-9958(65)90241-X.

[52] L. Zadeh, “Fuzzy algorithms,” *Information and Control*, vol. 12, no. 2, pp. 94–102, 1968, doi: https://doi.org/10.1016/S0019-9958(68)90211-8.

[53] “Use a Digital Signature.” [Online]. Available: https://developers.google.com/maps/documentation/maps-static/digital-signature

[54] C. H. Sudre, W. Li, T. Vercauteren, S. Ourselin, and M. Jorge Cardoso, “Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations,” in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, Springer International Publishing, 2017, pp. 240–248. doi: 10.1007/978-3-319-67558-9_28.

[55] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection.” [Online]. Available: https://arxiv.org/abs/1708.02002

[56] M. Deb, M. Deb, and A. R. Murty, “TopoNets: High performing vision and language models with brain-like topography,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=THqWPzL00e

[57] [Online]. Available: https://mathworld.wolfram.com/BettiNumber.html

[58] H. Noh, T. You, J. Mun, and B. Han, “Regularizing Deep Neural Networks by Noise: Its Interpretation and Optimization.” 2017.

Appendix

A: Branch loss function tests vii

A: Branch loss function tests

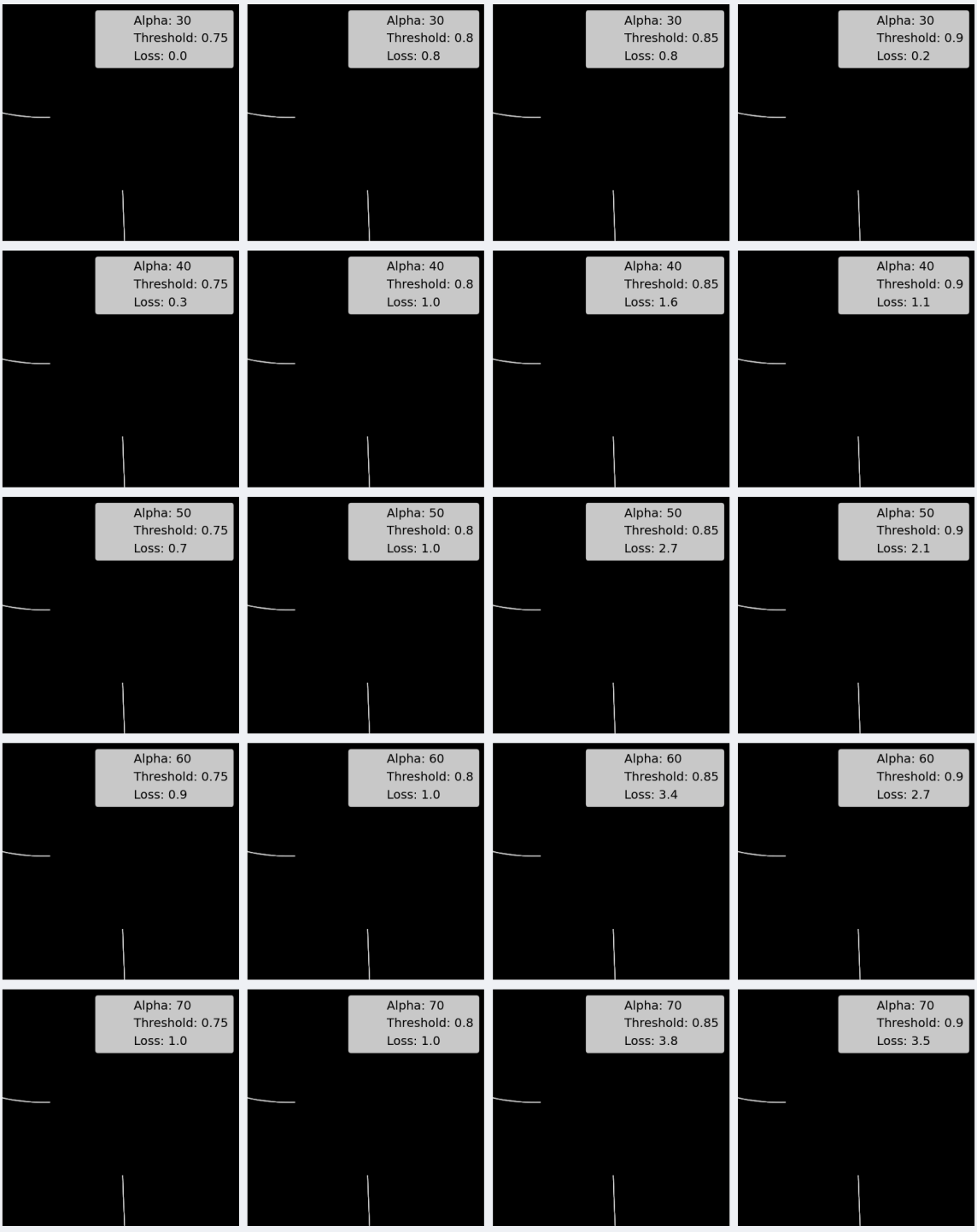


Figure 25. Showcase of various combinations of alpha and threshold values for the branch loss part of the topology loss function.