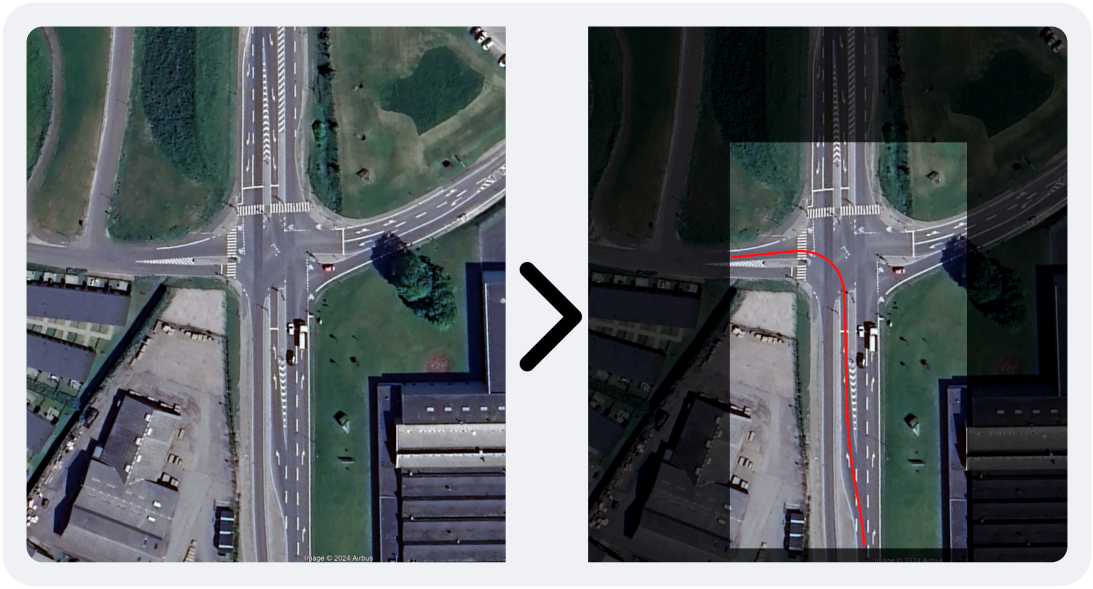
Pre-planning Intersection Traversal for Autonomous Vehicles

Master's Thesis in Computer Engineering

Ian Dahl Oliver

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

24-03-2025



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.

Acronyms Index

Acronym	Definition
AIM	Autonomous Intersection Management
APF	Artificial Potential Field
API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
AV	Autonomous Vehicle
BCE	Binary Cross-Entropy
BEV	Bird's Eye View
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DL	Deep Learning
DOF	Degrees of Freedom
FCN	Fully Convolutional Network
FL	Fuzzy Logic
GA	Genetic Algorithm
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
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
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 4
- 2.3 Satellite Imagery 4
- 2.4 Path Drawing 4
- 2.5 Pose Estimation 4

3 Related Work 5
- 3.1 Path-planning 5
- 3.2 Intersection Management 6

4 Methodology 7
- 4.1 Satellite Imagery 7
- 4.2 Loss Function Design 10
- 4.3 Dataset Creation 25
- 4.4 The Models 38

5 Results 40

6 Discussion 41
- 6.1 Integration with existing systems 41
- 6.2 Shortcomings 41
- 6.3 Future Work 41
- 6.4 Other considerations 41

7 Conclusion 42

References 43

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...

2.1 Autonomous Vehicles

2.2 Deep Learning

2.2.1 Computer Vision

2.2.2 Datasets

2.3 Satellite Imagery

2.4 Path Drawing

2.5 Pose Estimation

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 [10], 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 [11]. 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 [12]. D* has since become a very popular algorithm for path-planning in robotics, with improved alternatives like Focused D* the year after [13] and D* Lite from 2005 [14] proving use in real-world applications.

The concept of Artificial Potential Fields (APFs) was introduced in 1986 [15]. 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 [16], 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 [17], [18].

The Rapidly-exploring Random Tree (RRT) algorithm was introduced in 1998 [19], 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 [20], 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) [21], [22], despite challenges regarding the need for information about large areas [23].

Other areas of research in path-planning include Genetic Algorithms (GAs) and Fuzzy Logic (FL). GA [24] 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 [25], [26]. 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.

Neural Networks (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, Reinforcement Learning (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 Application Programming Interface (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 [27].

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 1.

```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 1. 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 Binary Cross-Entropy (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:

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

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:

$$\mathcal{L}_{\text{cold}} = C \cdot P \tag{2}$$

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 (1), 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:

$$\mathcal{L}_{\text{cold}} = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} C_{ij} P_{ij} \tag{3}$$

where H and W are the height and width of the image, respectively. The implementation of (3) 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 2. 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.

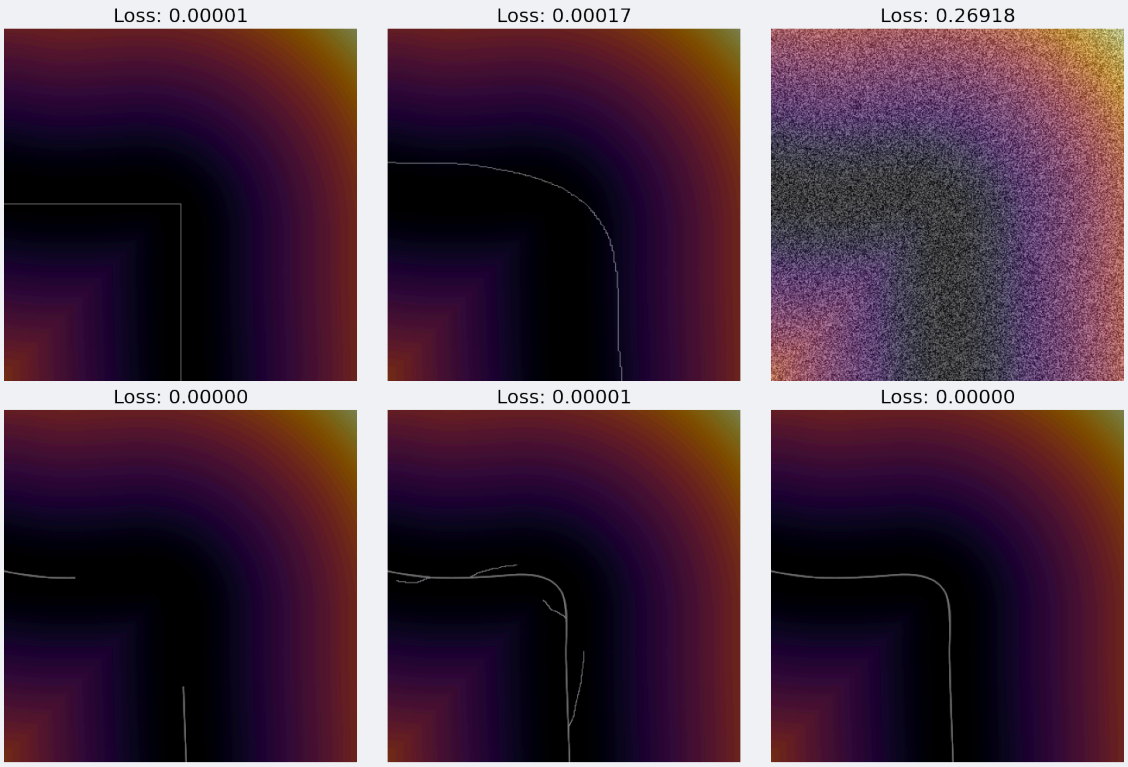


Figure 2. Paths drawn on top of a cold map with their associated loss above calculated using the function from Listing 4. The top row shows fully connected paths, while the bottom row shows paths with breaks and branches, as well as the true path.

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[1], which is defined as follows:

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

$$\ell(x, y) = L = \{l_1, ..., l_N\}^T \tag{4}$$

with

$$l_n = -w_n[y_n \log x_n + (1 - y_n) \log(1 - x_n)] \tag{5}$$

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

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'} \\ \text{sum}(L), & \text{if reduction} = \text{'sum'} \end{cases} \tag{6}$$

depending on the reduction parameter. The left-hand side of (5) 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 (5) 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.

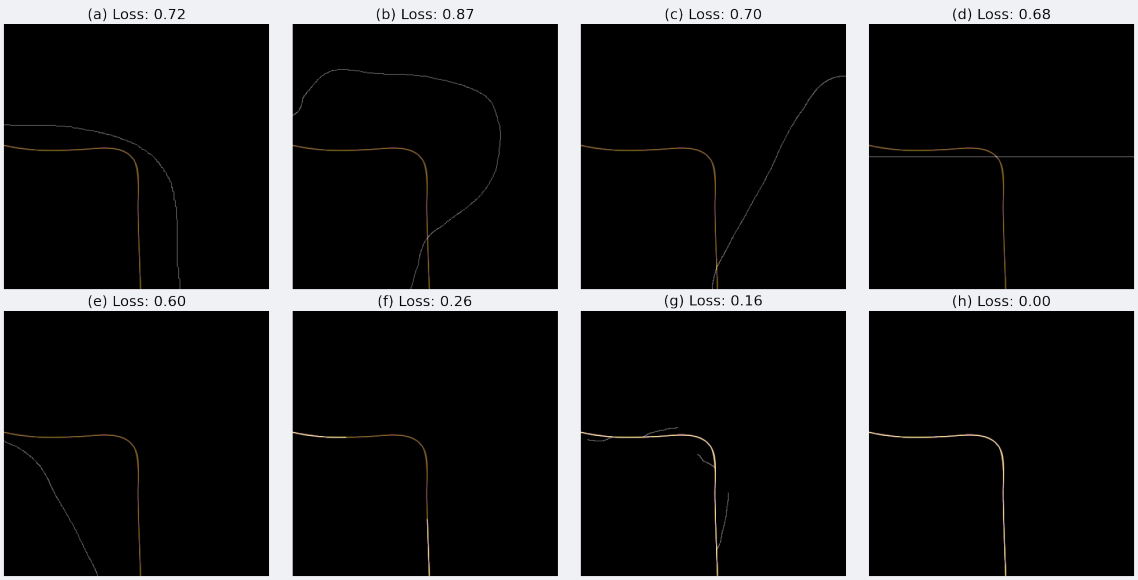


Figure 3. The ground truth● compared to some drawn path●. The losses above the plots are the BCE loss using function in Listing 5 using `mean` as the string passed as the reduction method.

Examples of the `mean`-based BCE loss is shown in Figure 3, 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 [28] similarity coefficient and Focal loss [29], 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.* [30] 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 [31] 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 4). 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.

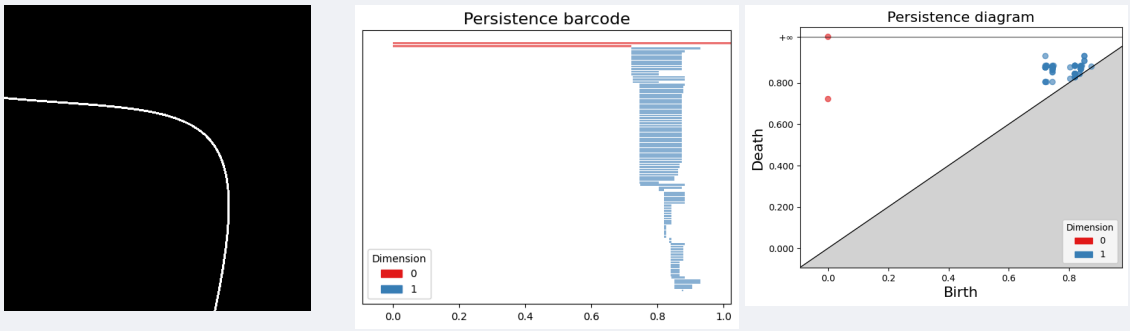


Figure 4. The top row shows a connected path along with its persistence barcode and persistence diagram, while the bottom row shows a disconnected path. The number of lines in the barcode, stems from the fact that the images are rather large in size and thus the number of built spaces are many.

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:

$$\mathcal{L}_{\text{cont}} = \sqrt{\sum_{\text{dim}} \left(\beta^{\text{dim}}_{\text{computed}} - \beta^{\text{dim}}_{\text{target}}\right)^2} \tag{7}$$

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

$$\Delta = \sum_{\text{dim}} \left(\beta^{\text{dim}}_{\text{computed}} - \beta^{\text{dim}}_{\text{target}}\right) \tag{8}$$

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

$$g = \frac{\Delta}{L} \tag{9}$$

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

$$\nabla_x L = g \cdot \text{grad\_output} \tag{10}$$

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 5. 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.

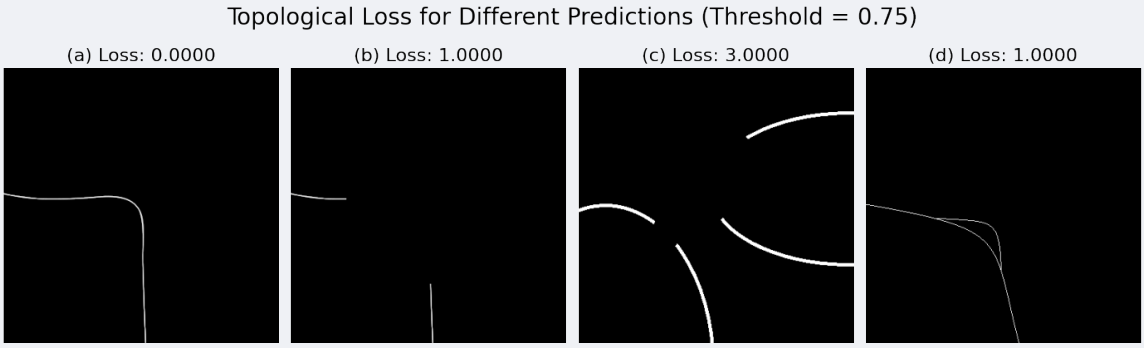


Figure 5. Results of using the continuity part of the topology loss.

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

$$k = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{11}$$

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:

$$\text{input\_tensor} = \sigma\left(\alpha\left(\frac{\text{input\_tensor}}{255} - t\right)\right) \tag{12}$$

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 (11). 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:

$$\text{neighbour\_sum}_{x,y} = \sum_{i=-a}^{a} \sum_{j=-b}^{b} k_{i,j} f_{x-i,y-j} \tag{13}$$

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

$$\text{indicator} = \exp\left(-\frac{(\text{neighbour\_sum} - 1)^2}{2\sigma^2}\right) \tag{14}$$

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

$$\text{endpoints} = \text{input\_tensor} \cdot \text{indicator} \tag{15}$$

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

$$\mathcal{L}_{\text{branch}} = (E - T)^2 \tag{16}$$

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

$$p = \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right) \tag{17}$$

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

$$I = \exp\left(-\frac{(S-1)^2}{2\sigma^2}\right) \tag{18}$$

where
- I is the indicator function,
- S is the neighbour sum obtained by convolution with k from (11),
- σ is a scalar value.

Pixel endpoint contribution

$$f(p) = p \cdot I \tag{19}$$

where
- p comes from (17),
- I is the indicator from (18).

Loss function definition

$$\mathcal{L}_{\text{branch}} = (E - T)^2 \tag{20}$$

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 (20) is simply

$$\frac{\partial \mathcal{L}_{\text{branch}}}{\partial E} = 2(E - T) \tag{21}$$

Secondly, the direct contribution from the indicator function is found by taking the derivative of (18) with respect to the pixel value p:

$$\frac{\partial}{\partial p} p \cdot I = I \tag{22}$$

directly representing the sensitivity of (19). 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 (18), its derivative with respect to S is

$$\frac{\partial I}{\partial S} = -\frac{S-1}{\sigma^2} \cdot I \tag{23}$$

To then find the contributions from neighbouring pixels, the product $p \cdot$ (23) is convolved with the kernel k from (11). Lastly, the derivative of (17) is found by taking the derivative of p with respect to x. First, the derivative of the sigmoid function is found:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{24}$$

So the derivative of p with respect to input x is

$$\frac{\partial}{\partial x} p = \frac{\alpha}{255} \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right)\left(1 - \sigma\left(\alpha\left(\frac{x}{255} - t\right)\right)\right) \tag{25}$$

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:

$$\frac{\partial}{\partial x}\mathcal{L}_{\text{branch}} = \frac{\partial \mathcal{L}_{\text{branch}}}{\partial E} \cdot \frac{\partial E}{\partial p} \cdot \frac{\partial p}{\partial x} \tag{26}$$

- $\frac{\partial \mathcal{L}_{\text{branch}}}{\partial E}$ was found in (21),
- $\frac{\partial E}{\partial p}$ is the direct and indirect contributions from the indicator function, and
- $\frac{\partial p}{\partial x}$ is (25) rewritten as $\frac{\alpha}{255}p(1-p)$

yielding the complete gradient

$$\frac{\partial}{\partial x}\mathcal{L}_{\text{branch}} = 2(E-T) \cdot \left[I + \text{Conv}\left(p \cdot \left(-\frac{S-1}{\sigma^2}\right)\right)\right] \cdot \frac{\alpha}{255}p(1-p) \tag{27}$$

as seen in Listing 8 line 48. Some results can be seen in Figure 6.

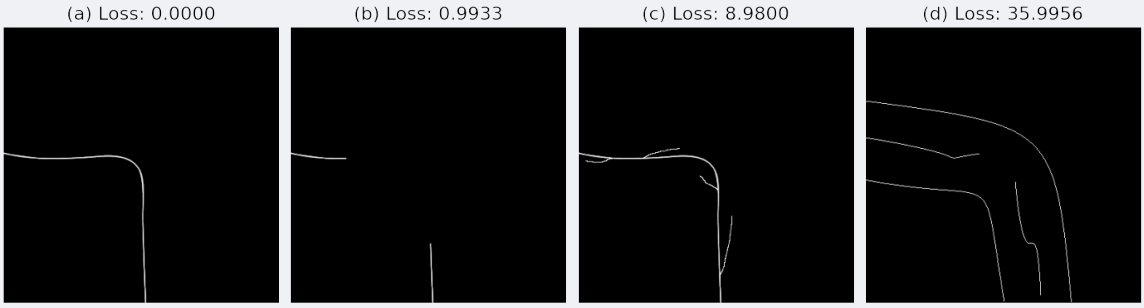


Figure 6. Results of using the branching part of the topology loss.

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 9. 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 $\mathcal{O}(n^4)$ [2]. 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:

$$d_{ij} = \sqrt{\left(i - c_{ij0}\right)^2 + \left(j - c_{ij1}\right)^2} \tag{28}$$

$$dt_{ij} = \begin{cases} d_{ij} & \text{if } d_{ij} < t \\ t + \left(d_{ij} - t\right)^e & \text{otherwise} \end{cases} \tag{29}$$

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 9.

[2] 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 7. 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.

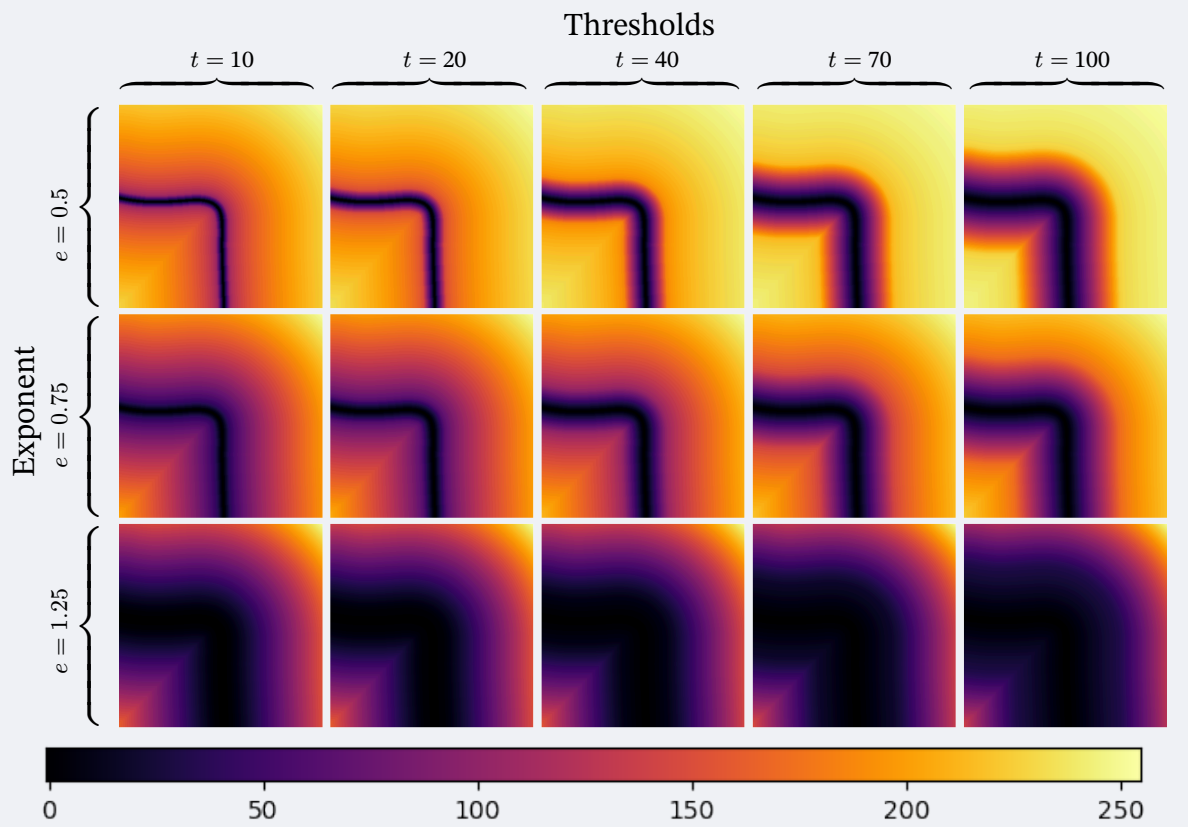


Figure 7. Results of testing threshold values $t \in \{10, 20, 40, 70, 100\}$ and exponent values $e \in \{0.5, 0.75, 1.25\}$. The colour map is shown beneath the results[3].

While pretty, these coldmaps can be difficult to understand. Therefore, Figure 8 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.

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

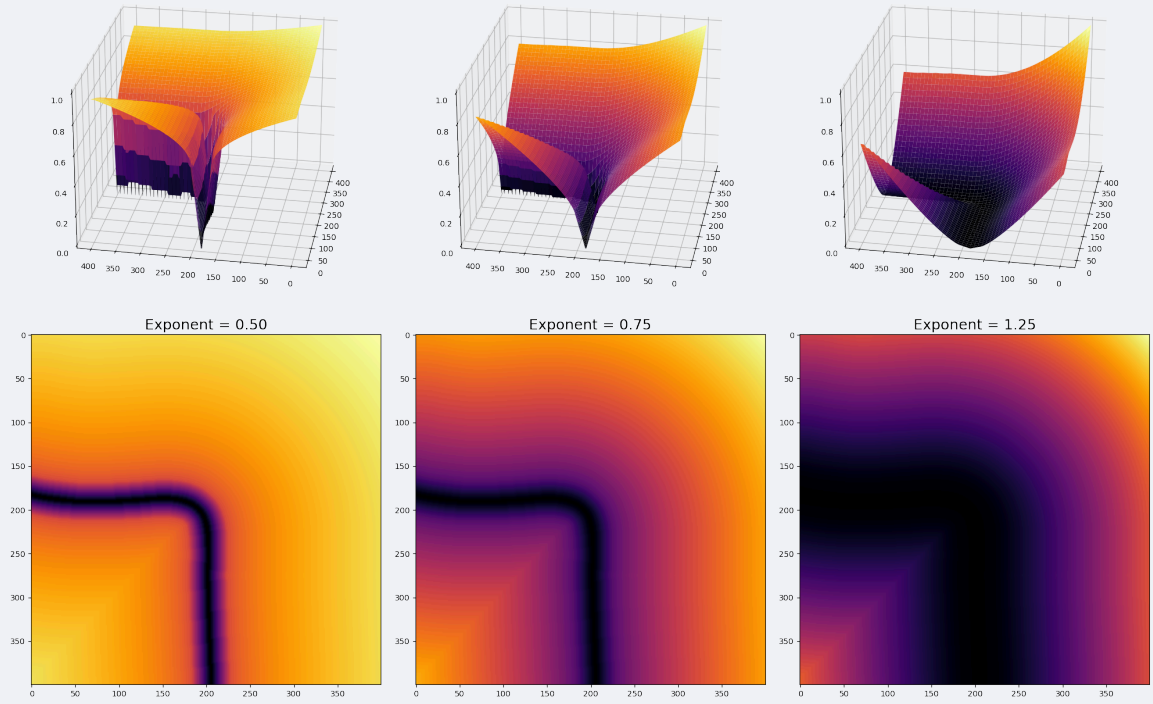


Figure 8. 3D plots of cold maps along with their 2D counterparts.

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 9. 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.

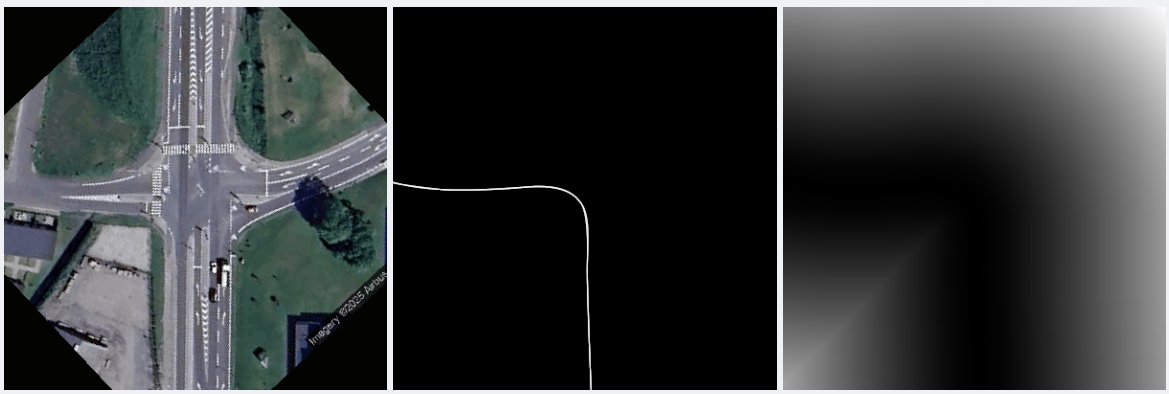


Figure 9. Example of satellite image, next to the desired path through with. To the far right, the generated cold map is shown with threshold $t = 20$ and exponent $e = 1.25$. Notice how it is only the points very close to the path that are very cold, while the rest of the map is warmer the further away 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 15. 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 10. 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.

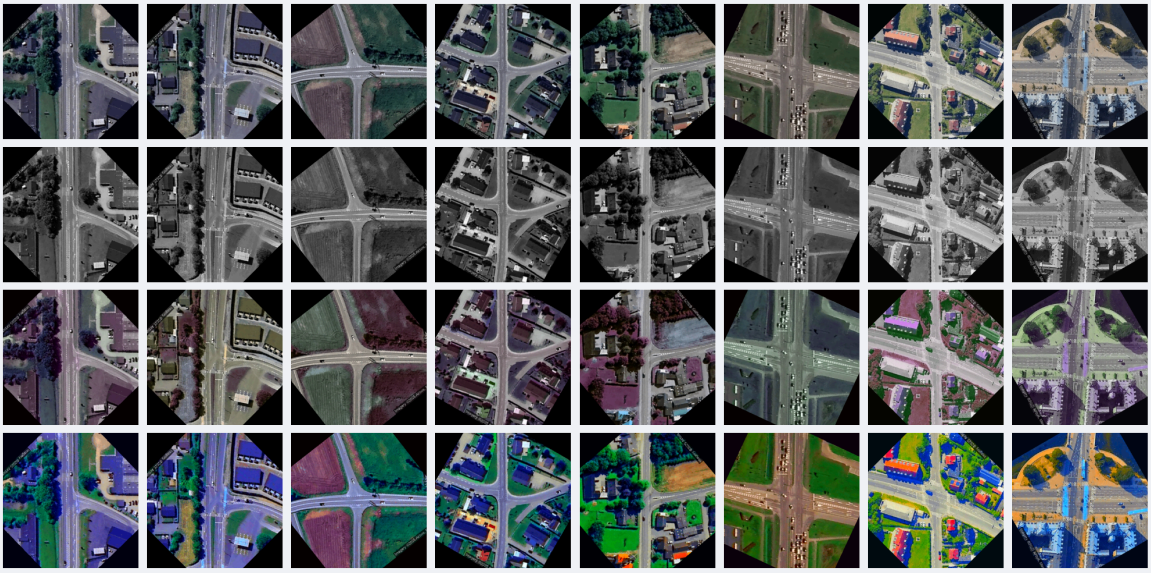


Figure 10. Example of the colouration augmentations. The top row is the original image, second row is the greyscale augmented image, third row is the hue adjusted image, and fourth row is the saturation adjusted image.

DISTORTION

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.

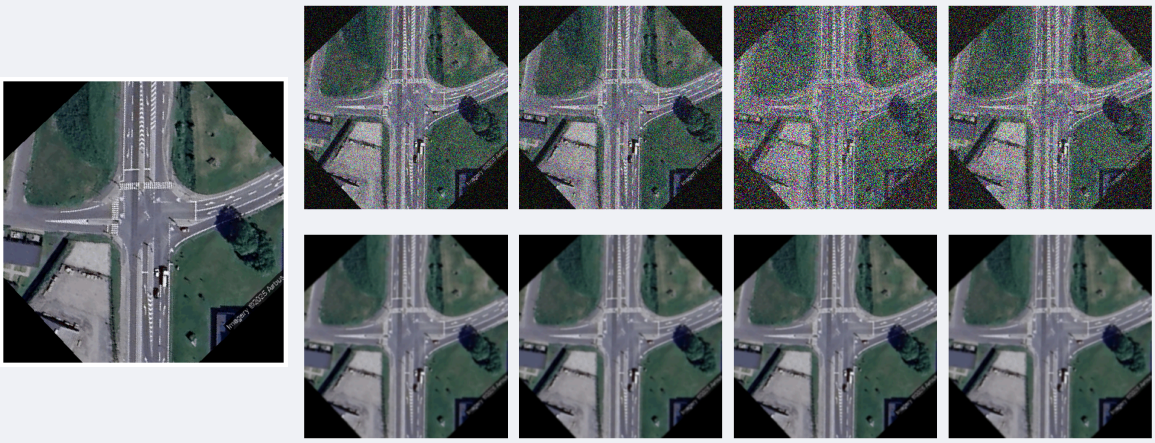


Figure 11. Example of the distortion augmentations. The far left image is the original image, the top row is the noise augmented images, and the bottom row is the blur augmented images.

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 15. 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

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 16 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 16. 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 17 below.

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

Listing 17. 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 18 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 18. 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 19. 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 20. Creating a dataloader for the dataset.

Arguments passed to the `DataLoader` initializer are the dataset from Listing 19, 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 12, where an example batch from the `DataLoader` is shown. The batch contains four paths, showcasing the path and the associated satellite image.

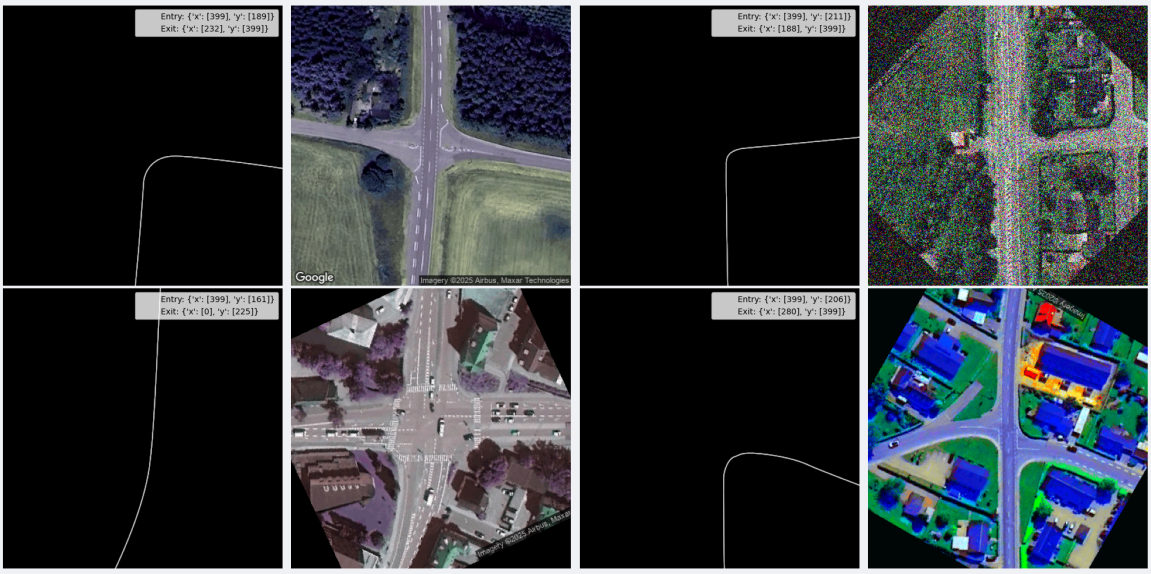


Figure 12. Example batch from the `DataLoader` with `batch_size = 4`.

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 21.

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 22.

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 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.

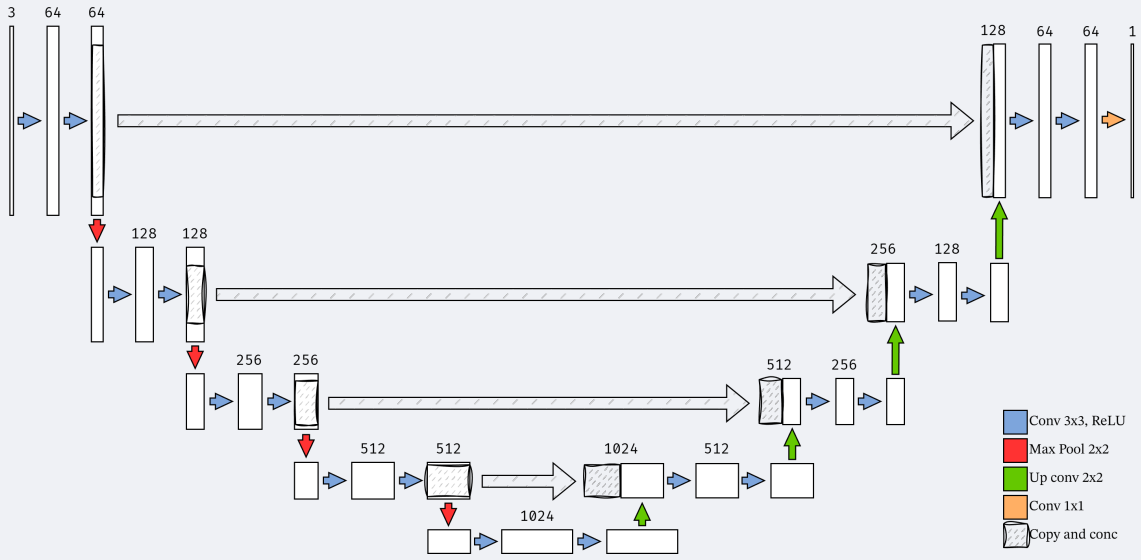


Figure 13. U-Net Architecture.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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

[30] 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

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

Appendix

A: Branch loss function tests vii

A: Branch loss function tests

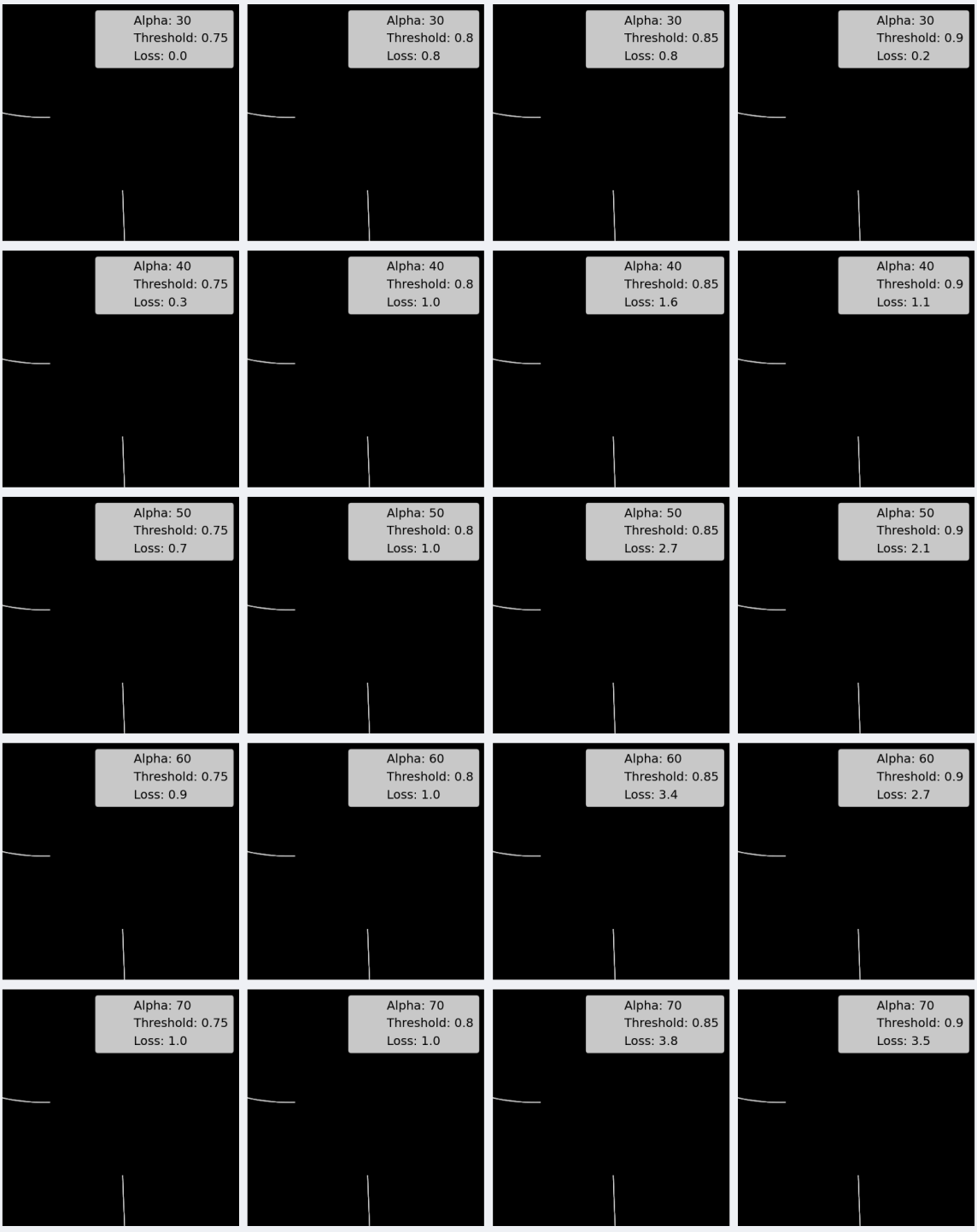


Figure 14. Showcase of various combinations of alpha and threshold values for the branch loss part of the topology loss function.